

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 r-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 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.</p> <p>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.</p> <p>Figura 8 – digrafo com custos λ-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).</p> <p>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 saíam 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.</p> <p>Figura 10 – Reexpansão da r-arborescência ótima T' em D' para obter a r-arborescência T em D</p> <p>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.</p> <p>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).</p> <p>Figura 13 – digrafo D com custos λ-reduzidos. Todos os arcos do ciclo custo mínimo agora custo zero.</p> <p>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.</p>	<p>16</p> <p>16</p> <p>16</p> <p>17</p> <p>18</p> <p>19</p> <p>20</p> <p>20</p> <p>20</p> <p>21</p> <p>21</p> <p>21</p> <p>22</p> <p>22</p>
---	---

Figura 15 – Duas r -arborescências ótimas distintas em D com custos c_λ -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.

23

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

30

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.

32

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. .

34

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.

37

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. .

39

Figura 21 – digrafo direcionado ponderado inicial com raiz no vértice 0. O digrafo 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 digrafo.

43

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. . .

44

Figura 23 – digrafo contraído após 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. . . .

44

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.

45

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.

45

Figura 26 – Dígrafo D com custos originais. Este exemplo ilustrará todas as etapas do algoritmo de András Frank, incluindo formação de ciclos e contração. 49

Figura 27 – Exemplo de redução de custo para o vértice a no dígrafo completo. À esquerda, os arcos entrando em a estão destacados em laranja com custos originais 2 e 5. Calculamos $\delta(\{a\}) = 2$ e subtraímos esse valor de ambos os arcos. À direita, após a redução: (r, a) tem custo zero (arco justo, em azul) e (b, a) tem custo $5 - 2 = 3$ (em laranja). Os demais arcos permanecem inalterados.

50

Figura 28 – Dígrafo após a primeira iteração. Os arcos justos (custo 0) são: (r, a) , (r, b) , (b, e) , (c, d) e (d, c) . Todos os vértices não-raiz possuem arcos justos entrando: a tem (r, a) , b tem (r, b) , e tem (b, e) , e o conjunto $\{c, d\}$ tem (a, c) (além do ciclo interno). Os arcos (c, d) e (d, c) formam um ciclo justo.

50

Figura 29 – Identificação de componentes fortemente conexas nos arcos justos após a primeira iteração. As componentes triviais $\{r, a\}$, $\{b\}$ e $\{e\}$ estão em verde. A componente não-trivial $\{c, d\}$ (em laranja) forma um ciclo justo com os arcos (c, d) e (d, c) , e é identificada como **minimal** para a próxima iteração. Os arcos justos internos ao ciclo estão destacados, indicando que $\{c, d\}$ deve ser tratado como uma unidade no processo de contração.

51

Figura 30 – Redução de custos para o subconjunto minimal $\{c, d\}$. À esquerda, antes da redução: os arcos entrando em $\{c, d\}$ vindos de fora são (r, c) com custo 6, (a, d) com custo 3 e (a, c) com custo 4, destacados em laranja. Calculamos $\delta(\{c, d\}) = 3$ e subtraímos esse valor. À direita, após a redução: (a, d) torna-se justo (custo 0), (r, c) tem custo reduzido para 3 e (a, c) tem custo reduzido para 1. O conjunto $\{c, d\}$ está destacado em laranja para enfatizar que é tratado como uma unidade.

52

Figura 31 – Contração do ciclo justo $\{c, d\}$. À esquerda, o dígrafo após as reduções de custo mostra o ciclo justo formado pelos arcos (c, d) e (d, c) (em vermelho). À direita, o dígrafo contraído onde os vértices c e d são substituídos pelo supervértice x_C . Os arcos que entravam ou saíam do ciclo são redirecionados para x_C . Note que os arcos justos agora formam uma r -arborescência no dígrafo contraído.

53

Figura 32 – Exemplo de múltiplas fontes disponíveis para processamento. À esquerda, o estado inicial possui três componentes $\{a\}$, $\{b\}$, $\{c\}$ (em laranja) que são fontes no grafo de condensação. Qualquer uma pode ser escolhida. À direita, após escolher e processar $\{a\}$ (elevando seu potencial por $\Delta(\{a\}) = 2$), o arco (r_0, a) torna-se justo (em azul). Agora $\{a\}$ deixa de ser fonte e as fontes restantes são $\{b\}$ e $\{c\}$. A ordem de processamento não afeta a arborescência ótima final.

55

59

67

Figura 35 – Ilustração da função `get_minimum_weight_cut` em D_{32} . Considerando os arcos em vermelho que entram em X (identificados pela função anterior), esta função calcula o peso mínimo entre eles. O arco em verde possui o menor peso (2), correspondendo ao valor $\Delta(X) = 2$.

68

Figura 36 – Ilustração da função `update_weights_in_X` em D_{32} . À esquerda, o dígrafo antes da atualização, com os arcos em vermelho entrando em X e $\Delta(X) = 2$. À direita, após subtrair $\Delta(X)$ de cada arco entrando em X : o peso (u_1, v_1) reduz de 3 para 1, (u_2, v_2) de 2 para 0 (torna-se justo), (u_3, v_3) de 4 para 2, e (u_1, v_2) de 5 para 3. O arco justo é adicionado a A_0 e D_0 . Note que os arcos da raiz e arcos internos/externos a X permanecem inalterados.

69

Figura 37 – Execução da função `phase1` em D_{32} . À esquerda, o dígrafo original D com pesos dos arcos. Ao centro, D_0 inicial após linha 5 (`build_D_zero`), contendo apenas vértices sem arcos. À direita, D_0 final após o loop de elevação de potenciais, contendo apenas os arcos justos (custo reduzido zero) que formam uma r-arborescência. Durante as iterações (linhas 7-22), os potenciais são elevados para cada componente sem arcos justos entrando, até que todos os vértices sejam alcançáveis a partir de r_0 através de arcos em D_0 .

72

Figura 38 – Execução da função phase2 em D_{32} . Superior esquerdo: conjunto A_0 com 6 arcos justos (custo reduzido zero) obtidos da Fase 1. Superior direito: Iteração 1 — adiciona arco (r_0, u_1) pois $r_0 \in \text{Arb}$ e $u_1 \notin \text{Arb}$. Inferior esquerdo: Iteração 2 — adiciona (r_0, u_3) . Inferior direito: Iteração 4 — após adicionar (u_1, u_2) na iteração 3, adiciona (u_1, v_1) . Vértices verdes foram recém-adicionados, vértices cinzas ainda não pertencem a Arb. O processo continua até que todos os 6 arcos de A_0 sejam incluídos, formando uma r-arborescência com pesos originais de D .	73
Figura 39 – Dígrafo inicial D com raiz r_0 e 6 vértices adicionais. O dígrafo contém múltiplos arcos com pesos variados. A Fase 1 do algoritmo iniciará com potenciais $y(v) = 0$ para todos os vértices, correspondendo aos custos reduzidos iniciais $c_y(u, v) = c(u, v)$.	80
Figura 40 – Fase 1, Iteração 1: Elevação de potenciais para $X_1 = \{u_1, u_2, u_3\}$. À esquerda, o conjunto X_1 (em laranja) sem arcos justos entrando. Calcula-se $\Delta(X_1) = \min\{1, 2, 1\} = 1$. À direita, após subtrair $\Delta(X_1)$, os arcos (r_0, u_1) e (r_0, u_3) tornam-se justos (custo reduzido 0) e são adicionados a A_0 .	81
Figura 41 – Fase 1, Iteração 2: Para o conjunto $X = \{v_1, v_2, v_3\}$ (em laranja), calcula-se $\Delta(X) = \min\{3, 2, 4, 5\} = 2$. Após subtrair esse valor dos arcos entrando em X , o arco (u_2, v_2) atinge custo reduzido 0 e é adicionado a A_0 . Note que (u_1, u_2) também se tornou justo na mesma iteração ao processar $X = \{u_2\}$.	81
Figura 42 – Conjunto final A_0 de arcos justos após conclusão da Fase 1. Todos os 6 arcos destacados possuem custo reduzido zero: (r_0, u_1) , (r_0, u_3) , (u_1, u_2) , (u_1, v_1) , (u_2, v_2) , (v_2, v_3) . O dígrafo formado por esses arcos é acíclico e alcança todos os vértices a partir de r_0 , satisfazendo as condições para prosseguir à Fase 2.	82
Figura 43 – Arborescência de custo mínimo final obtida pela Fase 2. Os pesos mostrados são os <i>originais</i> de D , restaurados após a construção. Cada vértice não-raiz possui exatamente um arco de entrada, não há ciclos, e todos os vértices são alcançáveis a partir de r_0 . A verificação de otimalidade dual confirma que cada conjunto X que teve potenciais elevados possui exatamente um arco da arborescência cruzando sua fronteira, garantindo que a solução é ótima.	82

Sumário

1	ALGORITMO DE CHU–LIU–EDMONDS	15
1.1	O algoritmo	15
1.2	Descrição do algoritmo	24
1.2.1	Corretude	25
1.2.2	Complexidade	26
1.3	Implementação em Python	26
1.3.1	Representação de digrafos e detecção de ciclos	27
1.3.2	Remoção de arcos que entram na raiz:	28
1.3.3	Redução de custos por vértice (normalização):	29
1.3.4	Construção de A_0 :	31
1.3.5	Detecção de ciclo:	32
1.3.6	Contração de ciclo:	34
1.3.7	Remoção de arco interno:	38
1.3.8	Procedimento principal (recursivo):	39
1.3.9	Correspondência entre teoria e implementação	45
1.3.10	Transição para a abordagem primal-dual	47
2	ALGORITMO DE ANDRÁS FRANK	48
2.1	O algoritmo	48
2.1.1	Identificação de conjuntos minimais	53
2.1.1.0.1	Componentes fortemente conexas e fontes.	54
2.1.1.0.2	Múltiplas escolhas possíveis.	54
2.1.2	Por que a escolha do conjunto minimal é importante	55
2.1.3	Arcos justos podem conter “sujeira”	56
2.1.3.0.1	Contração de ciclos.	56
2.1.3.0.2	Construção incremental.	57
2.1.4	Evolução das componentes fortemente conexas	57
2.1.4.0.1	Estado inicial.	57
2.1.4.0.2	Após cada elevação de potenciais.	57
2.1.4.0.3	Progressão monotônica.	57
2.1.4.0.4	Condição de término.	58
2.1.4.0.5	Exemplo visual.	58
2.2	Descrição do algoritmo	59
2.2.0.0.1	Observação importante sobre a Fase 1.	61
2.2.0.0.2	Observação sobre a Fase 2.	61

2.2.1	Corretude	62
2.2.2	Complexidade	62
2.2.3	Observações finais sobre o algoritmo	63
2.2.3.0.1	1. O algoritmo sempre termina.	63
2.2.3.0.2	2. Conjuntos minimais garantem otimalidade.	63
2.2.3.0.3	3. A Fase 1 constrói um certificado, não a arborescência.	63
2.2.3.0.4	4. Duas abordagens para a Fase 2.	63
2.2.3.0.5	5. Relação com Chu–Liu–Edmonds.	64
2.2.3.0.6	6. Eficiência através de componentes fortemente conexas.	64
2.3	Implementação em Python	65
2.3.1	Construção do dígrafo D_0 inicial	65
2.3.2	Identificação de arcos entrando em conjunto X	66
2.3.3	Cálculo do peso mínimo de corte	67
2.3.4	Atualização de pesos em X	68
2.3.5	Verificação de arborescência	69
2.3.6	Fase 1: Elevação de potenciais e construção de A_0	70
2.3.7	Fase 2: Construção da arborescência	75
2.3.8	Verificação de otimalidade dual	78
2.3.9	O algoritmo completo de András Frank	79
2.3.10	Correspondência entre teoria e implementação	83
REFERÊNCIAS		86
ANEXOS		87
ANEXO A – ANEXO A		88

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,

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

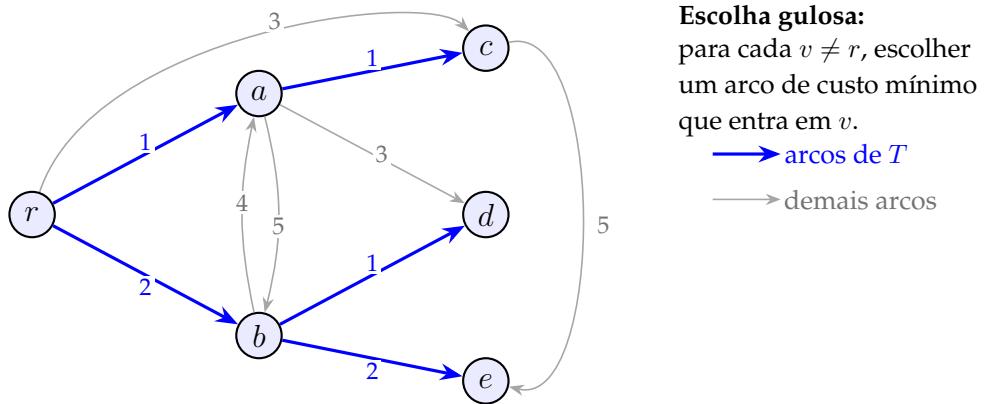


Figura 1 – A figura ilustra a escolha gulosa quando esta produz uma r -arborescência. Os arcos em azul são os escolhidos; os cinza são os demais arcos do digrafo.

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

A seguinte figura ilustra que podemos não ter tanta sorte.

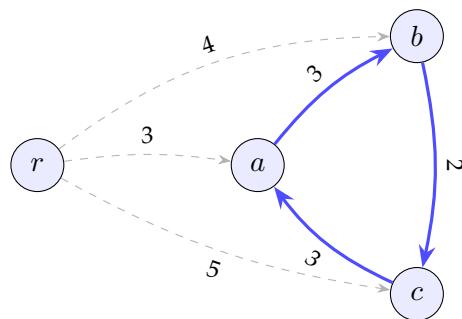


Figura 2 – Os arcos azuis são os da escolha gulosa.

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

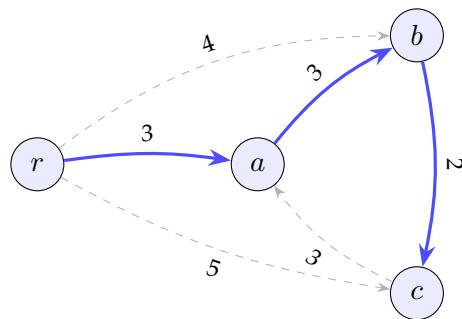


Figura 3 – Os arcos azuis são os da escolha gulosa.

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.

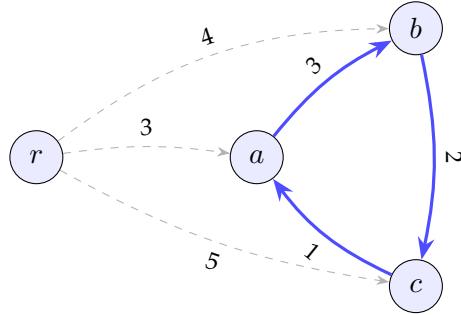


Figura 4 – Os arcos azuis são os da escolha gulosa.

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 λ 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_λ :

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

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

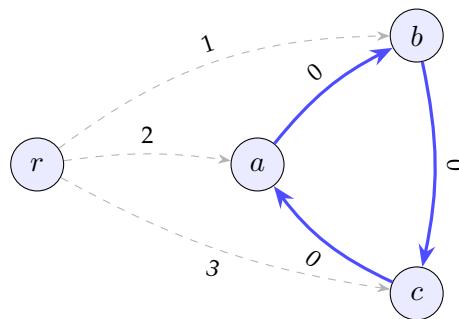


Figura 5 – Os arcos azuis são os da escolha gulosa.

Podemos agora retomar o caso no qual o subdigrafo gerador H de D , cujos arcos são aqueles em que o custo λ -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 maximal¹ 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

¹ 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.

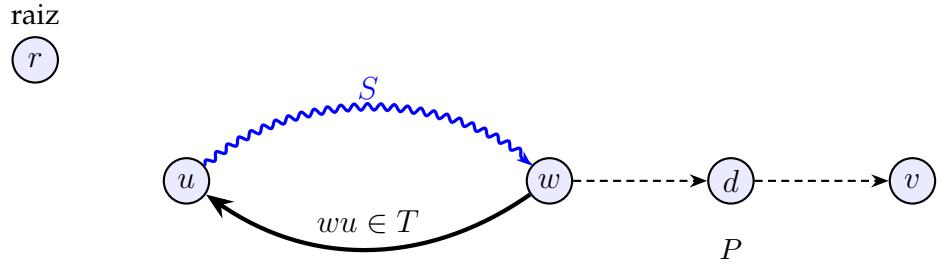


Figura 6 – O caminho simples maximal P inicia em u e termina em v . A porção S de P entre u e w é indicada pelo arco ondulado azul; o caminho $S \cdot u$ é um ciclo.

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 λ -reduzido:

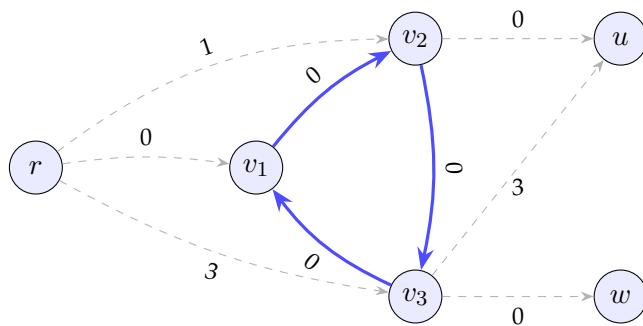


Figura 8 – digrafo com custos λ -reduzidos. Os arcos internos do ciclo C têm custo zero (em azul). Os arcos da raiz para o ciclo têm custos 1, 0 e 3 (tracejados).

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

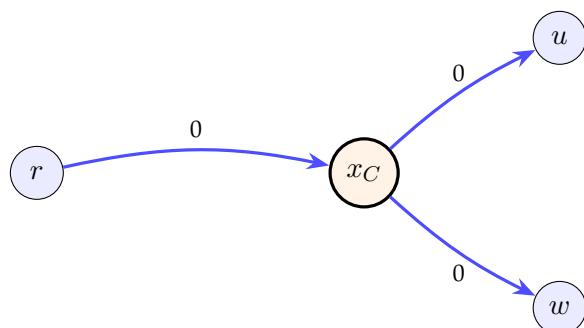


Figura 9 – digrafo D' após a contração do ciclo C . O supervértice x_C substitui todos os vértices do ciclo. Originalmente, havia três arcos paralelos de r para o ciclo: (r, v_1) , (r, v_2) e (r, v_3) com custos reduzidos 1, 0 e 3; mantemos apenas o de menor custo 0. Os arcos que saíram do ciclo agora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

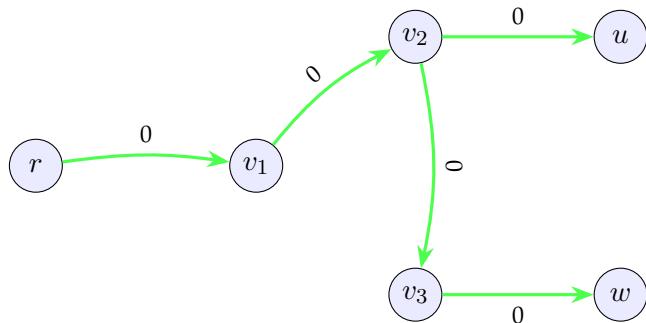


Figura 10 – Reexpansão da r -arborescência ótima T' em D' para obter a r -arborescência T em D

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

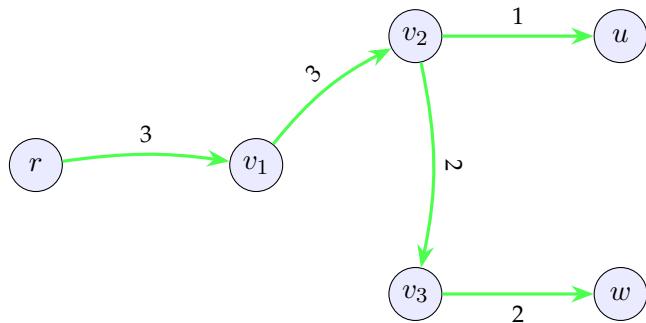


Figura 11 – Reexpansão da r -arborescência ótima T' em D' para obter a r -arborescência T em D . Os arcos selecionados em verde fazem parte de T .

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:

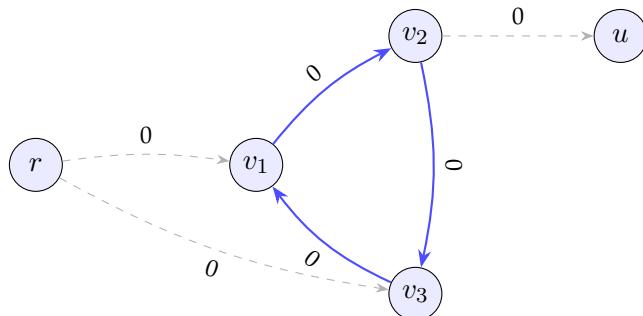


Figura 13 – digrafo D com custos λ -reduzidos. Todos os arcos do ciclo custo mínimo têm agora custo zero.

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'

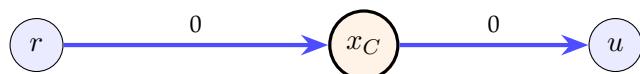


Figura 14 – Arborescência T' no digrafo contraído D' . O arco (r, x_C) pode corresponder ou ao arco (r, v_1) ou ao (r, v_3) em D e o arco (x_C, u) corresponde ao arco normalizado (v_2, u) em D .

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

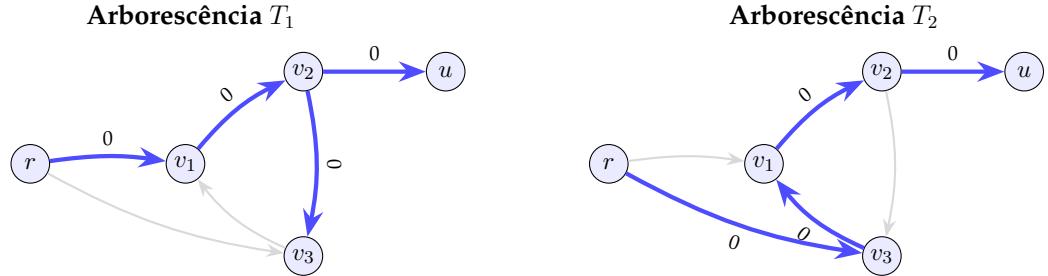


Figura 15 – Duas r -arborescências ótimas distintas em D com custos c_λ -reduzidos. T_1 usa o arco (r, v_1) e os arcos do ciclo (v_1, v_2) e (v_2, v_3) . T_2 usa o arco (r, v_3) e os arcos do ciclo (v_3, v_1) e (v_1, v_2) . Ambas incluem o arco (v_2, u) e têm custo total zero.

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 λ -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_λ 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_λ -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

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

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).

^a 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 NetworkX², 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.

² 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

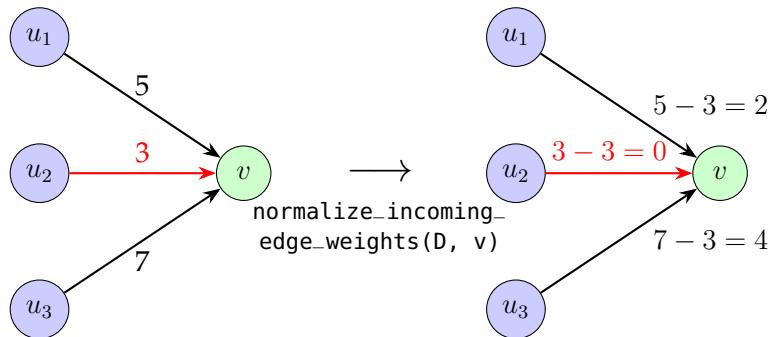


Figura 16 – Exemplo de normalização de custos reduzidos. À esquerda, vértice v com três arcos de entrada (pesos 5, 3 e 7). À direita, após aplicar $reduce_weights(D, v)$: o menor peso $y(v) = 3$ é subtraído de todas as entradas, resultando em custos reduzidos 2, 0 e 4. O arco (u_2, v) (em vermelho) tem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

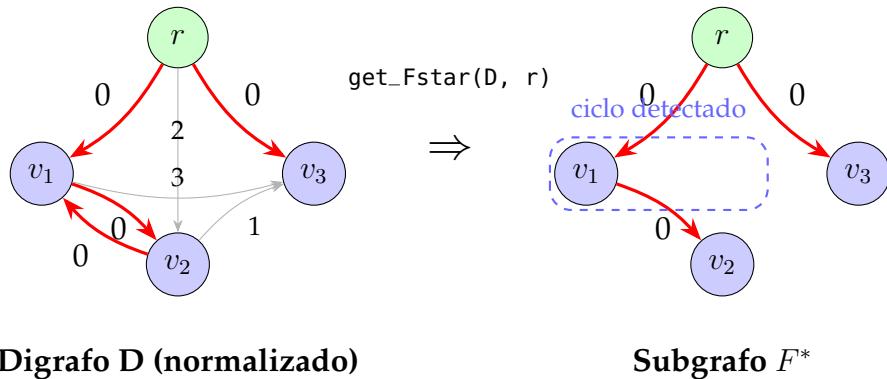


Figura 17 – Exemplo de construção de A_0 a partir de um digrafo normalizado. À esquerda, o digrafo D após normalização, onde cada vértice não-raiz possui ao menos um arco de entrada com custo zero (em vermelho). À direita, o afó A_0 resultante contém apenas os arcos de custo zero selecionados, um por 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_{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_{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:

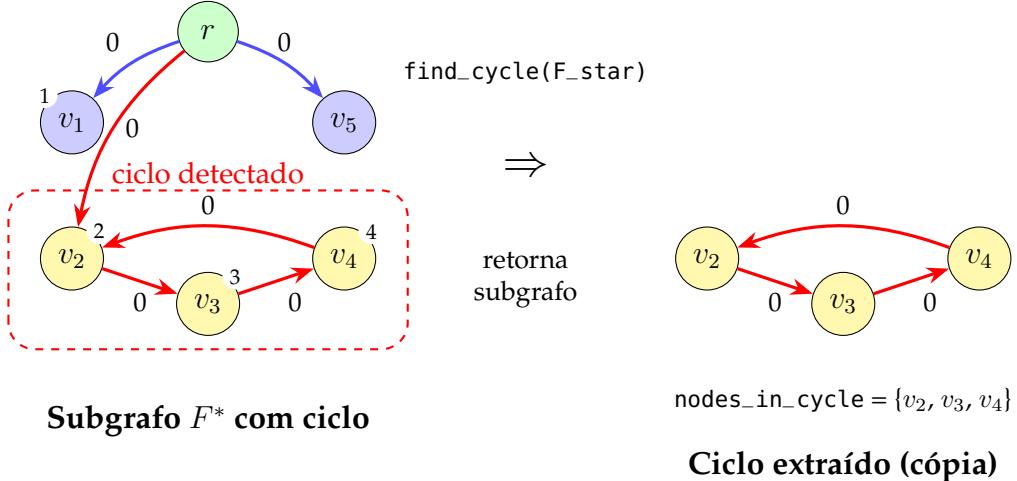


Figura 18 – Exemplo de detecção de ciclo em A_0 . À esquerda, o subdigrafo A_0 contém um ciclo formado pelos vértices $\{v_2, v_3, v_4\}$ (destacados em amarelo). A DFS percorre o digrafo e detecta o ciclo ao encontrar o arco (v_4, v_2) , onde v_2 já está na pilha de recursão. À direita, a função devolve 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 `extttin_to_cycle[u]=(w,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 `out_from_cycle[v]=(w,weight)`. Depois de coletar estas informações (para evitar mutações durante iterações), o código cria no digrafo D arcos redirecionados (u, label) e (label, v) com os pesos mínimos correspondentes. Por fim, a função devolve os dicionários `in_to_cycle` e `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:

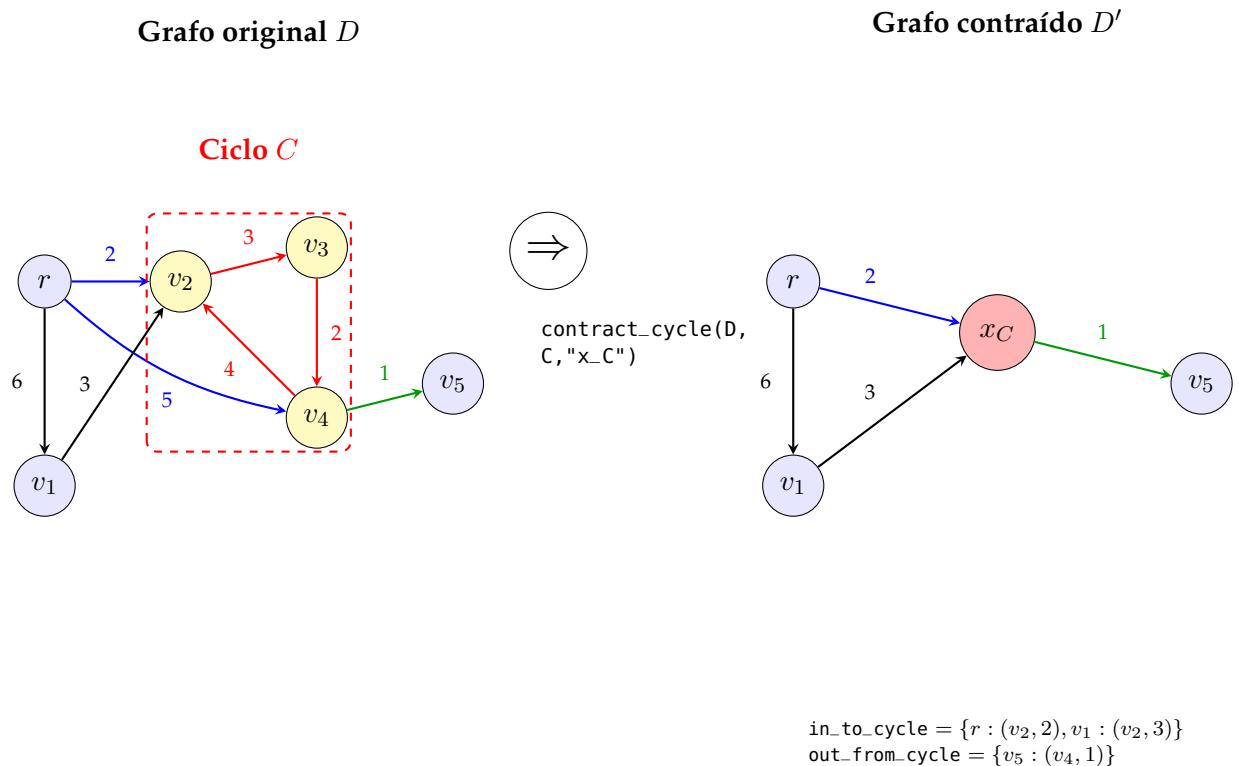


Figura 19 – Exemplo de contração de ciclo. À esquerda, digrafo original D com ciclo $C = \{v_2, v_3, v_4\}$ (em amarelo). Vértices externos r , v_1 e v_5 têm arcos conectando ao ciclo: r envia arco para v_2 (peso 2) e v_4 (peso 5); v_4 envia arco para v_5 (peso 1). À direita, após a contração: o ciclo 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:

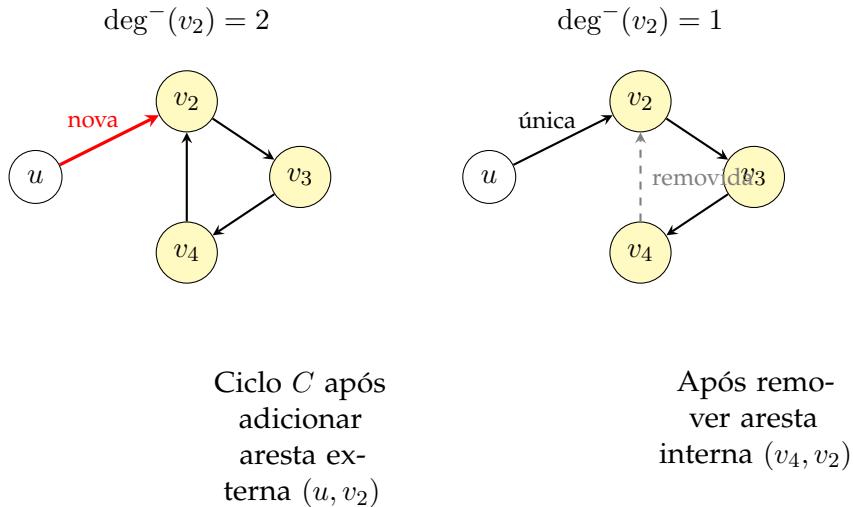


Figura 20 – Remoção de arco interno durante reexpansão. À esquerda, ciclo $C = \{v_2, v_3, v_4\}$ após adicionar arco externo (u, v_2) vindouro da arborescência T' : o vértice v_2 tem grau de entrada 2 (arco externo vermelho de u e arco interno do ciclo vindo de v_4), violando a propriedade de arborescência. 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):

Agora apresentaremos a função principal que orquestra todas as funções auxiliares descritas anteriormente no fluxo completo do algoritmo descrito por Chuliu–Edmonds. A função recebe como entrada um digrafo ponderado D , o vértice raiz $r\theta$, e um parâmetro `level` (padrão 0) usado para rotular supervértices de acordo os distintos níveis recursivos.

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 e chamamos uma função de `cast()` para garantir que o tipo seja `nx.DiGraph`, pois o método `copy()`

devolve um tipo `nx.Graph`. A cópia é necessária porque as operações de normalização e contração modificam os pesos dos arcos diretamente e precisamos preservar os dados originais para restaurar os custos corretos na arborescência final. A complexidade é $O(m + n)$ para copiar o digrafo, onde m é o número de arcos e n o número de vértices.

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). Caso sim, restaura os pesos originais de D para cada arco de A_0 (linhas 8–9) e devolve A_0 como solução (linha 10). A função `nx.is_arborescence` testa conectividade, aciclicidade e o grau de entrada correto simultaneamente. Precisamos verificar essa condição porque, se A_0 for acíclico, já encontramos a arborescência ótima e não há necessidade de contração ou recursão. Essa verificação é portanto o caso base da recursão.

Contração e resolução recursiva — caso recursivo (linhas 11–14):

Essa operação começa detectando-se o ciclo C ao chamar `find_cycle(A_zero)` (linha 11). Em seguida é criado um rótulo `cl = f"contracted_{level}"` para o supervértice (linha 12) para identificar o ciclo contraído. Na linha 13, a função `contract_cycle(D_copy, C, contracted_label)` é chamada para contrair o ciclo. e modifica `D_copy` diretamente criando o digrafo contraído D' e devolve os dicionários `in_to_cycle` e `out_from_cycle` que serão usados na reexpansão. Na linha 14 chamamos recursivamente `chuliu_edmonds(D_copy, r0, level + 1)` para resolver o problema no digrafo contraído, incrementando o nível recursivo. A arborescência ótima F' do digrafo contraído é devolvida e armazenada em `F_prime`.

Reexpansão do ciclo contraído (linhas 15–28):

Esse processo começa identificando o arco externo que entra no supervértice (linha 15) usando `next(iter(F_prime.in_edges(cl, data=True)))` para obter o primeiro arco de entrada em `cl`, desempacotando na forma `(u, _, _)` onde `u` é o vértice externo que conecta ao ciclo, é necessário chamar a função `cast()` para garantir o tipo correto, pois

`in_edge` devolve um iterador de tuplas genéricas. O vértice externo u é extraído (linha 16). Na linha 17, extrai-se v com $v = \text{in_to_cycle}[u]$ para identificar qual vértice do ciclo recebe a conexão externa; em seguida, chama-se `remove_edge_cycle(C, v)` (linha 18) para eliminar o arco interno que entrava em v , quebrando o ciclo no ponto de entrada e restaurando a propriedade de arborescência.

Na linha 19, o arco externo (u, v) é adicionado a F' . E nas linhas 20–21, todos os arcos internos do ciclo C são adicionados a F' para reconstruir a estrutura interna do ciclo. Em seguida, itera-se sobre os arcos que saem do supervértice cl em F' e para cada arco (cl, z) , identifica-se o vértice original do ciclo `u_cycle` usando `out_from_cycle[z]` e adiciona-se o arco (u_cycle, z) a F' , restaurando as conexões de saída do ciclo (linhas 22–24).

Finalmente, o supervértice cl é removido de F' (linha 25) e os pesos originais de D são restaurados para todos os arcos em F' (linhas 26–27). No final, a arborescência resultante F' é devolvida (linha 28).

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 r-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_edmonds(D: nx.DiGraph, rθ: str, level=0):
2     D_copy = cast(nx.DiGraph, 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 C = find_cycle(A_zero)
12 cl = f"\n {level}" # contracted label
13 in_to_cycle, out_from_cycle = contract_cycle(D_copy, C, cl)
14 F_prime = chuliu_edmonds(D_copy, rθ, level + 1)
15 in_edge = next(iterator(F_prime.in_edges(cl, data=True)))

```

```

16     u, _, _ = cast(tuple, in_edge)
17     v, _ = in_to_cycle[u]
18     remove_edge_cycle(C, v)
19     F_prime.add_edge(u, v)
20     for u_c, v_c in C.edges:
21         F_prime.add_edge(u_c, v_c)
22     for _, z, _ in F_prime.out_edges(cl, data=True):
23         u_cycle, _ = out_from_cycle[z]
24         F_prime.add_edge(u_cycle, z)
25     F_prime.remove_node(cl)
26     for u2, v2 in F_prime.edges:
27         F_prime[u2][v2]["w"] = D[u2][v2]["w"]
28     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: $F_{\text{prime}} = \text{chuliu}(D_{\text{copy}}, r_0, \text{level}+1)$ (linha 14 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_{prime} 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 $\text{remove_internal_edge_to_cycle_entry}(C, v)$ —, 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 15–28).
- O resultado é uma arborescência em D com pesos originais restaurados.

Aqui finalizamos a descrição detalhada da implementação do algoritmo de

Chu–Liu/Edmonds. A seguir, apresentamos um exemplo completo de execução do algoritmo em um digrafo específico, ilustrando cada etapa do processo.

Exemplo de execução do algoritmo

Aqui ilustraremos cada fase do processo: normalização, construção de A_0 , detecção de ciclos, contração, resolução recursiva e reexpansão a partir do digrafo inicial mostrado na Figura 21.

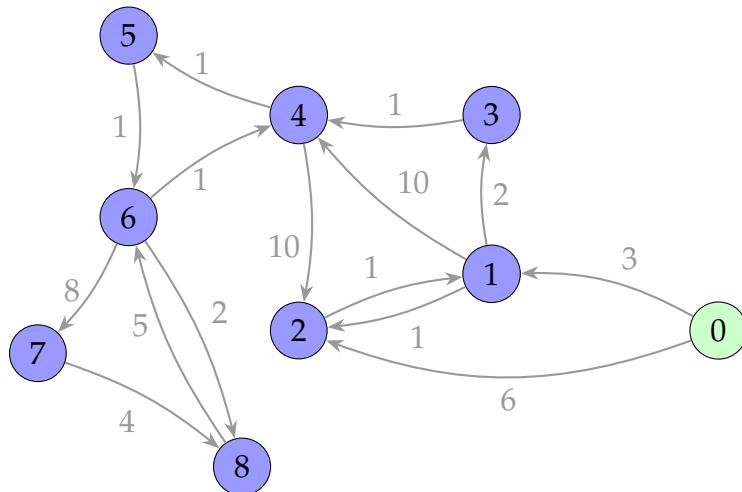


Figura 21 – digrafo direcionado ponderado inicial com raiz no vértice 0. O digrafo contém 9 vértices e múltiplos arcos com pesos variados. O primeiro passo do algoritmo seria remover arcos que entram na raiz, porém não há nenhum neste caso, logo não existe necessidade de alterar o digrafo.

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 digrafo.

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 $n * 0$.

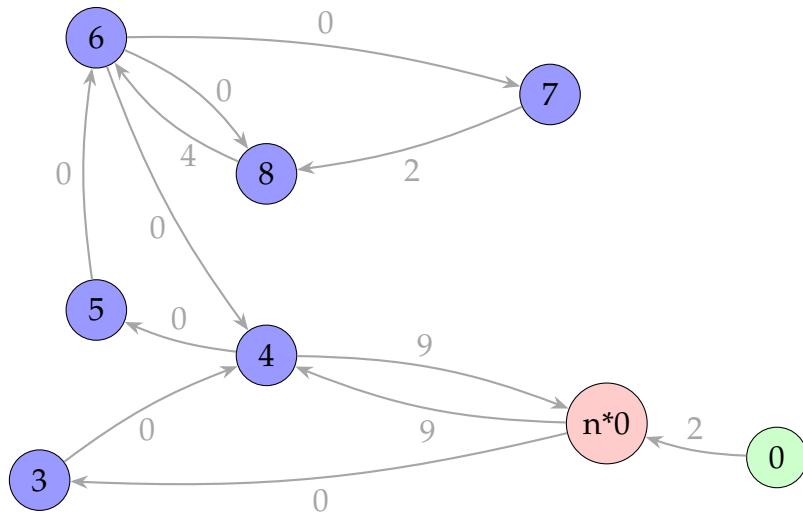


Figura 23 – digrafo contraído apóis detecção do ciclo $C = \{1, 2\}$ em A_0 . O ciclo foi contraído no supervértice $n * 0$ (destacado em vermelho). Os arcos que entravam ou saíam do ciclo foram redirecionados para o supervértice, com custos ajustados segundo as fórmulas $c'(u, x_C) := c(u, w) - y(w)$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

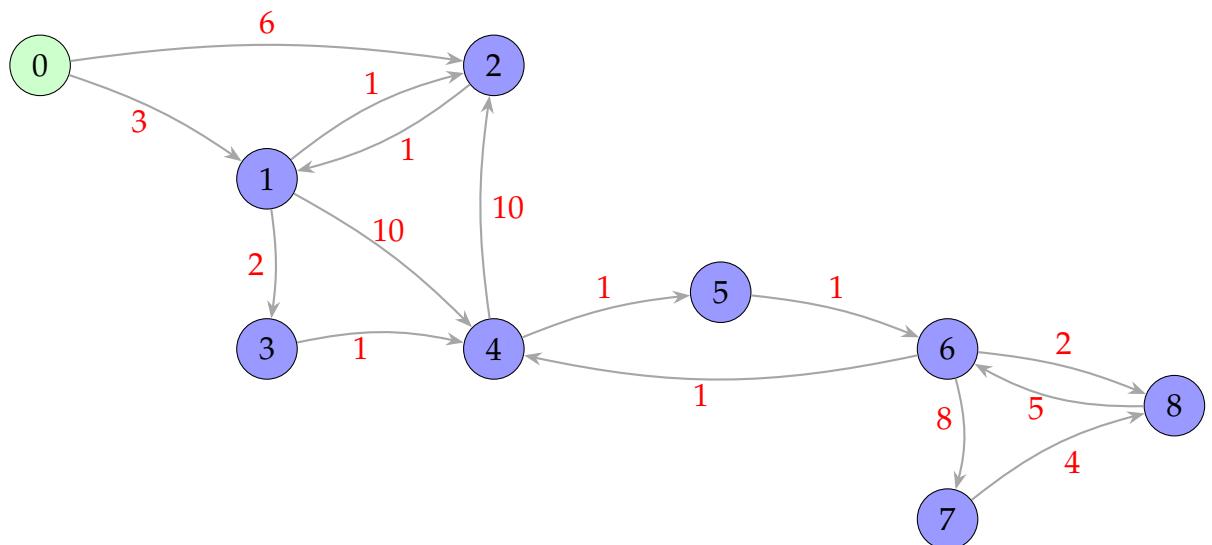


Figura 25 – Arborescência ótima final no digrafo original com pesos restaurados. O supervértice $n * 0$ foi expandido de volta para os vértices 1 e 2, com o arco externo $(0, 1)$ escolhido pela solução recursiva conectando ao ciclo. O arco interno $(2, 1)$ do ciclo original foi removido para manter a 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>Passo 1: Normalização e construção de A_0. 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 $A_0 := \{a_v : v \neq r\}$.</p>	<p>Linhas 3–6: <code>for v in D_copy.nodes: reduce_weights(D_copy, v) A_zero = get_Azero(D_copy, r0)</code> Calcula $y(v)$ e cria custos reduzidos, depois constrói A_0 selecionando arcos de custo zero.</p>
<p>Passo 2: Verificação de aciclicidade (caso base). Se (V, A_0) é acíclico, devolva A_0. Por Obs. 4.36 de (KLEINBERG; TARDOS, 2006), trata-se de uma r-arborescência de custo mínimo.</p>	<p>Linhas 7–10: <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>Passo 3: Contração de ciclo. Caso contrário, seja C um ciclo dirigido de A_0 (com $r \notin C$). Contraia C em supervértice x_C e defina custos c' por: $c'(u, x_C) := c(u, w) - y(w)$ $c'(x_C, v) := c(w, v)$ Denote o digrafo contraído por $D' = (V', A')$.</p>	<p>Linhas 11–13: <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 D_{copy} para criar D'.</p>
<p>Passo 4: Resolução recursiva. Resolva recursivamente em D', obtendo arborescência ótima F'.</p>	<p>Linha 14: <code>F_prime = chuliu(D_copy, r0, level+1)</code> Chamada recursiva resolve o problema no digrafo contraído.</p>
<p>Passo 5: Reexpansão. Expanda x_C para o ciclo original C. Se $(u, x_C) \in F'$, adicione (u, v) onde v é o vértice do ciclo mapeado por u, remova o arco interno entrando em v, e reintegre demais arcos de C. Restaure custos originais.</p>	<p>Linhas 15–28: <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_edge_cycle`) 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 ser melhorada em termos de intuição e generalização.

No capítulo seguinte, revisitaremos o mesmo problema sob uma ótica gulosa–dual em duas fases, proposta por András Frank. Essa perspectiva organiza a normalização via potenciais³ $y(\cdot)$, 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.

³ 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.

2 Algoritmo de András Frank

Neste capítulo, apresentaremos o algoritmo de András Frank, que também 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 através da identificação de subconjuntos minimais 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. A operação de redução é essencialmente a mesma do algoritmo de Chu–Liu–Edmonds — subtrair o menor custo de arco entrando em cada conjunto — mas enquanto Chu–Liu–Edmonds opera vértice a vértice, o algoritmo de Frank identifica *subconjuntos minimais* através de componentes fortemente conexas, processando todos simultaneamente a cada iteração. O propósito deste capítulo é fornecer tanto uma descrição teórica do algoritmo quanto detalhes da implementação desenvolvida neste trabalho.

2.1 O algoritmo

O algoritmo de András Frank também 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 e devolve uma r -arborescência c -mínima de D .

Assim como no capítulo anterior, adotamos a terminologia de r -digrafo ponderado para uma tripla (D, c, r) em que (D, c) é um digrafo ponderado, r é um vértice de D , $\delta^-(r) = \emptyset$ e D possui uma r -arborescência.

O algoritmo de Frank opera em duas fases principais: (i) redução de custos e (ii) construção da arborescência a partir dos arcos que se tornaram justos (custo reduzido zero).

Na primeira fase, identificamos iterativamente os *subconjuntos minimais* — subconjuntos $X \subseteq V \setminus \{r\}$ tais que nenhum arco justo entra em X , mas todo subconjunto próprio de X possui ao menos um arco justo entrando. Para cada conjunto minimal X , calculamos $\delta(X)$, o menor custo entre todos os arcos que entram em X , e subtraímos esse valor de todos esses arcos, criando ao menos um novo arco justo. Esse processo é repetido até que exista exatamente uma *fonte* no grafo de condensação.

Vamos desenvolver essas ideias utilizando um exemplo: considere o dígrafo D da Figura 26 com seis vértices $\{r, a, b, c, d, e\}$ e custos nos arcos para ilustrar o comportamento completo do algoritmo.

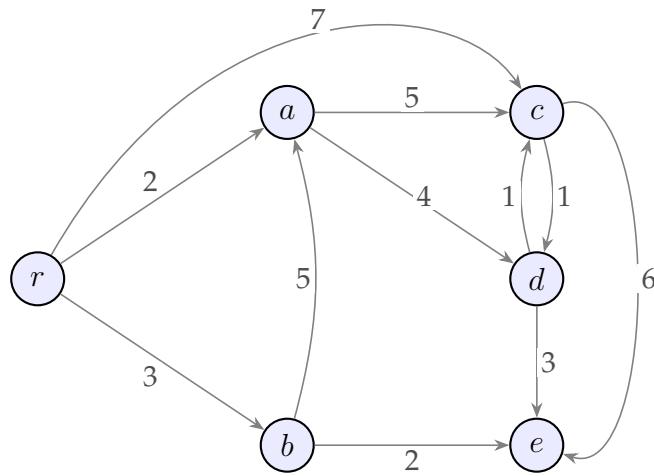


Figura 26 – Dígrafo D com custos originais. Este exemplo ilustrará todas as etapas do algoritmo de András Frank, incluindo formação de ciclos e contração.

Vamos agora executar o algoritmo de Frank sobre este dígrafo. Inicialmente, nenhum arco tem custo reduzido zero, cada vértice não-raiz $v \neq r$ forma seu próprio subconjunto minimal $\{v\}$. No dígrafo da Figura 26, os subconjuntos minimais iniciais são portanto $\{a\}$, $\{b\}$, $\{c\}$, $\{d\}$ e $\{e\}$. Para cada um desses conjuntos unitários, encontramos $\delta(\{v\})$, o menor custo entre todos os arcos entrando em v , e subtraímos esse valor de todos esses arcos.

Por exemplo, consideremos o vértice a . Os arcos entrando em a são (r, a) com custo 2 e (b, a) com custo 5. Calculamos $\delta(\{a\}) = \min\{2, 5\} = 2$ e subtraímos este valor de ambos os arcos. Assim, (r, a) passa a ter custo $2 - 2 = 0$ (tornando-se um *arco justo*) e (b, a) passa a ter custo $5 - 2 = 3$. A Figura 27 ilustra essa operação de redução aplicada ao vértice a .

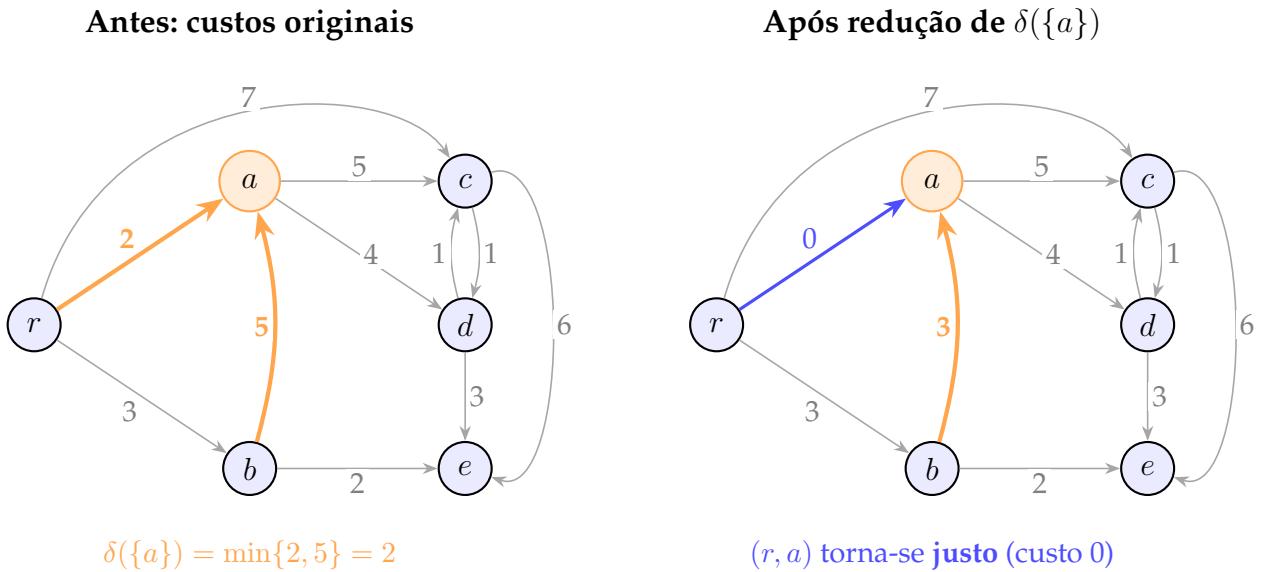


Figura 27 – Exemplo de redução de custo para o vértice a no dígrafo completo. À esquerda, os arcos entrando em a estão destacados em laranja com custos originais 2 e 5. Calculamos $\delta(\{a\}) = 2$ e subtraímos esse valor de ambos os arcos. À direita, após a redução: (r, a) tem custo zero (arco justo, em azul) e (b, a) tem custo $5 - 2 = 3$ (em laranja). Os demais arcos permanecem inalterados.

Essa operação vai sendo repetida para todos os conjuntos minimais iniciais. A Figura 28 mostra o resultado dessa primeira iteração no exemplo da Figura 26, onde aplicamos a redução apenas para os vértices isolados.

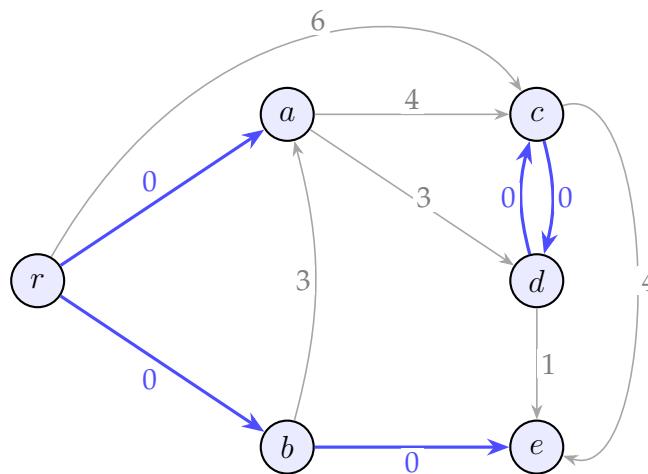


Figura 28 – Dígrafo após a primeira iteração. Os arcos justos (custo 0) são: (r, a) , (r, b) , (b, e) , (c, d) e (d, c) . Todos os vértices não-raiz possuem arcos justos entrando: a tem (r, a) , b tem (r, b) , e tem (b, e) , e o conjunto $\{c, d\}$ tem (a, c) (além do ciclo interno). Os arcos (c, d) e (d, c) formam um ciclo justo.

Se esses arcos de custo zero formam uma r -arborescência, o algoritmo termina. Caso contrário, identificamos novamente os subconjuntos minimais sem arcos de custo

zero entrando neles.

No exemplo da Figura 28, os arcos justos não formam uma r -arborescência, pois nem todos os vértices são alcançáveis a partir de r pelos arcos justos e o conjunto $\{c, d\}$ forma um ciclo com dois arcos justos entre c e d . Como não existem arcos justos entrando em $\{c, d\}$, esse conjunto é um subconjunto minimal. Além disso, nesse caso forma-se uma componente fortemente conexa entre c e d . Uma componente fortemente conexa (SCC) é um subconjunto maximal de vértices em que cada vértice é alcançável a partir de qualquer outro vértice do mesmo subconjunto.

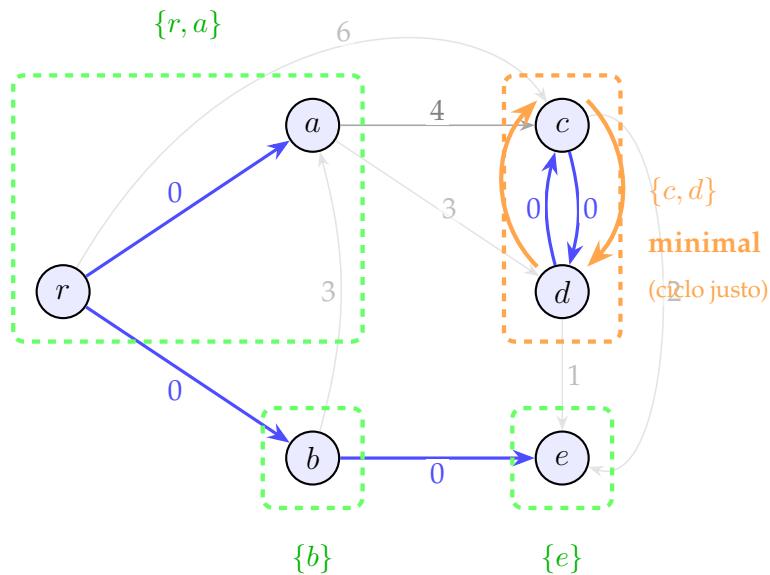


Figura 29 – Identificação de componentes fortemente conexas nos arcos justos após a primeira iteração. As componentes triviais $\{r, a\}$, $\{b\}$ e $\{e\}$ estão em verde. A componente não-trivial $\{c, d\}$ (em laranja) forma um ciclo justo com os arcos (c, d) e (d, c) , e é identificada como **minimal** para a próxima iteração. Os arcos justos internos ao ciclo estão destacados, indicando que $\{c, d\}$ deve ser tratado como uma unidade no processo de contração.

O algoritmo então procede escolhendo um subconjunto minimal (neste caso, $\{c, d\}$) e repetindo o processo de redução de custos para esse conjunto. Calculamos $\delta(\{c, d\})$ como o menor custo entre os arcos que entram em c ou d vindos de fora do conjunto $\{c, d\}$. No exemplo, os arcos que entram em $\{c, d\}$ são (r, c) com custo 6, (a, c) com custo 4 e (a, d) com custo 3. Portanto, $\delta(\{c, d\}) = \min\{6, 4, 3\} = 3$. Subtraímos esse valor dos arcos que entram em c e d , tornando (a, d) um arco justo.

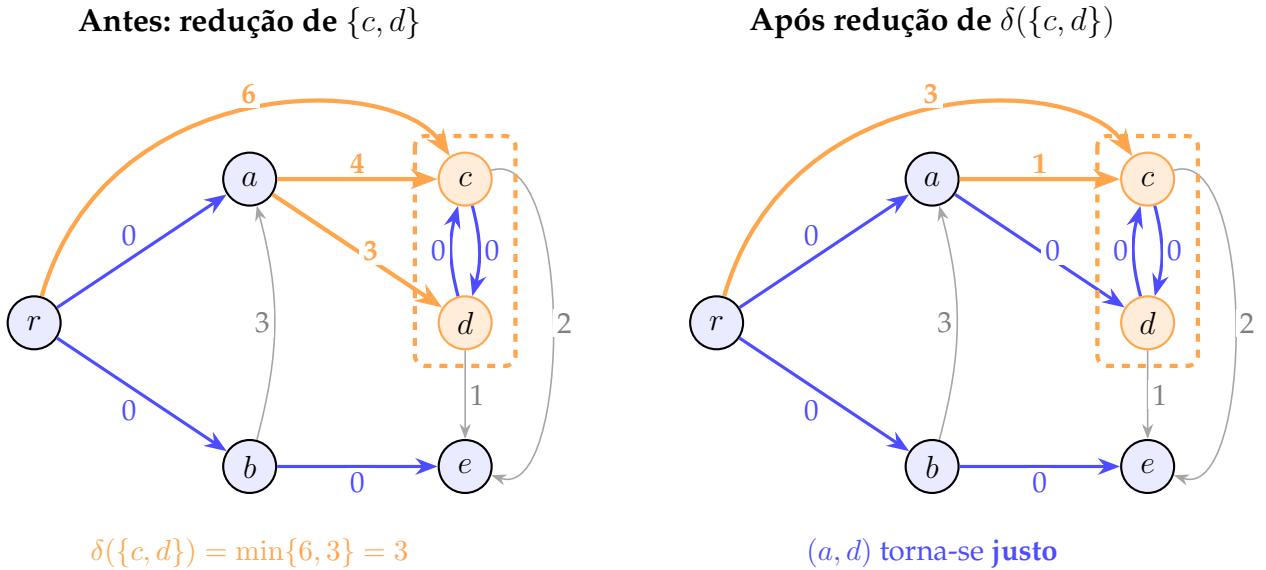
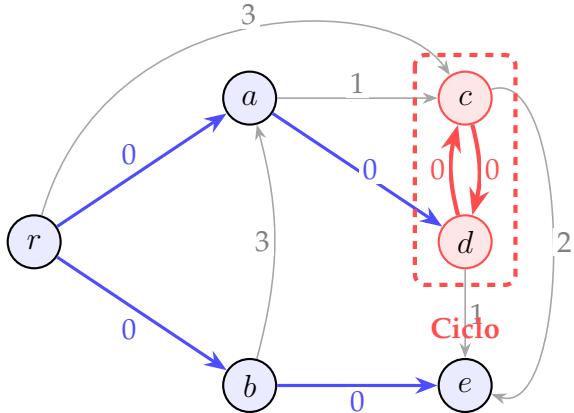


Figura 30 – Redução de custos para o subconjunto minimal $\{c, d\}$. À esquerda, antes da redução: os arcos entrando em $\{c, d\}$ vindos de fora são (r, c) com custo 6, (a, d) com custo 3 e (a, c) com custo 4, destacados em laranja. Calculamos $\delta(\{c, d\}) = 3$ e subtraímos esse valor. À direita, após a redução: (a, d) torna-se justo (custo 0), (r, c) tem custo reduzido para 3 e (a, c) tem custo reduzido para 1. O conjunto $\{c, d\}$ está destacado em laranja para enfatizar que é tratado como uma unidade.

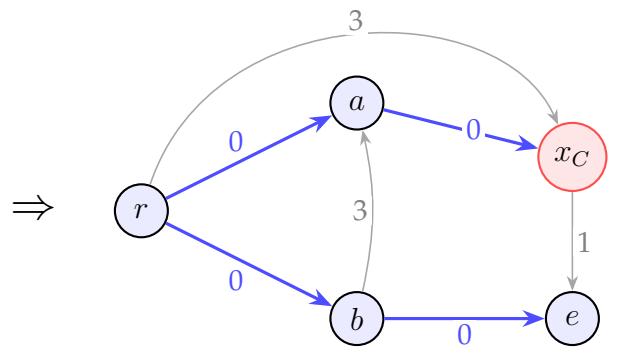
Agora, não existem mais subconjuntos minimais sem arcos justos entrando neles, porém os arcos justos não formam uma r -arborescência, pois existe um ciclo justo entre c e d . Nesse ponto, o algoritmo procede para a fase de contração, onde o ciclo justo formado pelos arcos (c, d) e (d, c) é contraído em um único vértice. Podemos estender essa ideia de contração de ciclos para abranger a contração de qualquer componente fortemente conexa (SCC) formada por arcos justos. No exemplo, a SCC $\{c, d\}$ é contraída em um novo vértice x_C . Todos os arcos que entravam em c ou d agora entram em x_C , e todos os arcos que saíam de c ou d agora saem de x_C . Os custos dos arcos são mantidos conforme estavam antes da contração.

A Figura 31 ilustra essa contração.

Antes: dígrafo com ciclo justo



Após contração: $\{c, d\} \rightarrow x_C$



Supervértice x_C
representa $\{c, d\}$

Figura 31 – Contração do ciclo justo $\{c, d\}$. À esquerda, o dígrafo após as reduções de custo mostra o ciclo justo formado pelos arcos (c, d) e (d, c) (em vermelho). À direita, o dígrafo contraído onde os vértices c e d são substituídos pelo supervértice x_C . Os arcos que entravam ou saíam do ciclo são redirecionados para x_C . Note que os arcos justos agora formam uma r -arborescência no dígrafo contraído.

Após a contração, os arcos justos formam uma r -arborescência no dígrafo contraído.

Assim termina a fase 1 do algoritmo de Frank. A fase 2 envolve a expansão do supervértice x_C de volta para o ciclo $\{c, d\}$ e a reconstrução da r -arborescência ótima no dígrafo original. Isso é feito selecionando o arco que entra em x_C na arborescência do dígrafo contraído e substituindo-o pelo arco correspondente que entra em c ou d no dígrafo original, além dos arcos justos internos ao ciclo, exceto o arco que entra no vértice escolhido.

2.1.1 Identificação de conjuntos minimais

Um aspecto fundamental do algoritmo de Frank é a identificação correta de *conjuntos minimais*. Um conjunto $X \subseteq V \setminus \{r\}$ é dito **minimal** se:

1. Não há arcos justos entrando em X (ou seja, nenhum arco (u, v) com $u \notin X$ e $v \in X$ possui custo reduzido zero).
2. Para qualquer subconjunto próprio $Y \subset X$, existe ao menos um arco justo entrando em Y .

A segunda condição garante a *minimalidade*: não podemos encontrar um subconjunto menor que também não tenha arcos justos entrando. Essa propriedade é crucial

para a corretude do algoritmo, pois escolher um conjunto não-minimal poderia levar a uma solução subótima.

2.1.1.0.1 Componentes fortemente conexas e fontes.

O algoritmo utiliza componentes fortemente conexas (SCCs) formadas pelos arcos justos para identificar eficientemente os conjuntos minimais. Quando construímos o grafo de condensação das SCCs:

- Cada nó do grafo de condensação representa uma SCC do grafo original formado pelos arcos justos.
- Um arco no grafo de condensação conecta duas SCCs se existe um arco justo no grafo original indo de um vértice da primeira SCC para um vértice da segunda.
- O grafo de condensação é sempre acíclico (DAG).

Uma **fonte** no grafo de condensação é uma SCC sem arcos justos entrando. Essas fontes correspondem exatamente aos conjuntos minimais que buscamos:

- **Sem arcos justos entrando:** Por definição de fonte, não há arcos no grafo de condensação entrando naquela SCC, o que significa que não há arcos justos do grafo original entrando em nenhum vértice da SCC.
- **Maximalidade da componente:** Como trabalhamos com SCCs, qualquer vértice dentro da componente é alcançável a partir de qualquer outro usando arcos justos. Isso garante que não há subconjunto próprio sem arcos justos entrando (pois dentro da componente, todos os vértices são conectados por arcos justos).

2.1.1.0.2 Múltiplas escolhas possíveis.

Em uma iteração do algoritmo, pode haver múltiplas fontes no grafo de condensação (excluindo a componente que contém a raiz). Qualquer uma dessas fontes pode ser escolhida como conjunto minimal para elevação de potenciais naquela iteração. A escolha específica não afeta a corretude do algoritmo, apenas a ordem em que os arcos se tornam justos.

Para ilustrar, considere um dígrafo simples com raiz r_0 conectada a três vértices a , b , e c através de arcos com custos 2, 3, e 2 respectivamente. Após a inicialização, todas as três componentes $\{a\}$, $\{b\}$, $\{c\}$ são fontes (cada uma sem arcos justos entrando). O algoritmo pode escolher qualquer uma delas:

- Se escolhermos $\{a\}$: elevamos seu potencial por $\Delta(\{a\}) = 2$, tornando (r_0, a) justo.

- Se escolhermos $\{b\}$: elevamos seu potencial por $\Delta(\{b\}) = 3$, tornando (r_0, b) justo.
- Se escolhermos $\{c\}$: elevamos seu potencial por $\Delta(\{c\}) = 2$, tornando (r_0, c) justo.

Em cada caso, após a elevação, a componente escolhida deixa de ser uma fonte (pois agora possui um arco justo entrando). O algoritmo continua processando as fontes restantes nas iterações subsequentes. A ordem de processamento não afeta o custo total da arborescência final, pois todas as elevações são necessárias e determinadas univocamente pelos custos dos arcos.

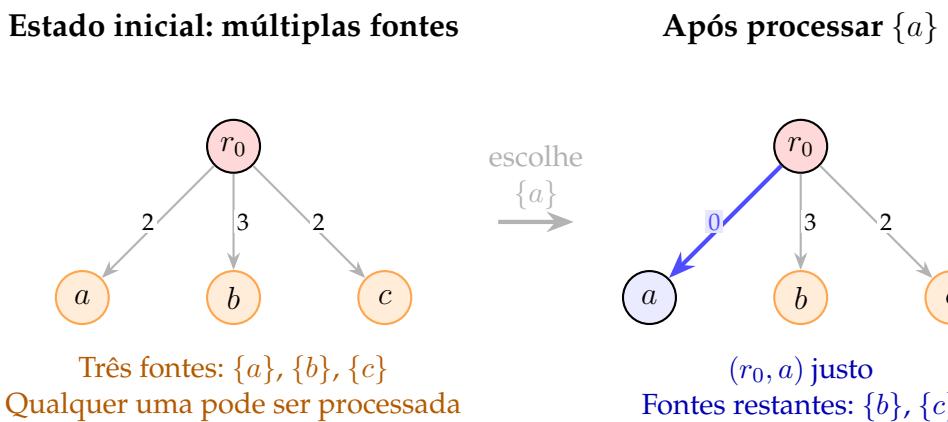


Figura 32 – Exemplo de múltiplas fontes disponíveis para processamento. À esquerda, o estado inicial possui três componentes $\{a\}$, $\{b\}$, $\{c\}$ (em laranja) que são fontes no grafo de condensação. Qualquer uma pode ser escolhida. À direita, após escolher e processar $\{a\}$ (elevando seu potencial por $\Delta(\{a\}) = 2$), o arco (r_0, a) torna-se justo (em azul). Agora $\{a\}$ deixa de ser fonte e as fontes restantes são $\{b\}$ e $\{c\}$. A ordem de processamento não afeta a arborescência ótima final.

No exemplo da Figura 28, após a primeira iteração de redução, temos os vértices $\{a\}$, $\{b\}$, $\{e\}$ que são componentes triviais (cada um forma sua própria SCC) e todos possuem arcos justos entrando. Porém, o conjunto $\{c, d\}$ forma uma SCC não-trivial (um ciclo) sem arcos justos entrando, constituindo uma fonte no grafo de condensação e, portanto, um conjunto minimal. Não poderíamos escolher apenas $\{c\}$ ou apenas $\{d\}$ isoladamente, pois esses subconjuntos já possuem arcos justos entrando (os arcos do ciclo interno).

2.1.2 Por que a escolha do conjunto minimal é importante

Escolher um conjunto que não seja minimal pode violar a otimalidade da solução. Para entender isso, considere a interpretação dual do algoritmo: cada vez que elevamos o potencial de um conjunto X por $\Delta(X)$, estamos aumentando a variável dual correspondente. As condições de folga complementar da programação linear dual exigem que:

- Se um arco (u, v) está na arborescência ótima, então seu custo reduzido deve ser zero.
- Se um conjunto X teve seu potencial elevado ($\Delta(X) > 0$), então exatamente um arco da arborescência deve cruzar a fronteira de X .

Se escolhêssemos um conjunto não-minimal X (isto é, um conjunto que contém um subconjunto próprio $Y \subset X$ que também não tem arcos justos entrando), poderíamos elevar os potenciais de forma redundante, criando folga nas restrições duais. Isso pode levar a uma solução que satisfaz as restrições mas não atinge o valor ótimo.

Ao garantir que sempre escolhemos conjuntos minimais, mantemos a propriedade de que a sequência de elevações de potenciais constrói progressivamente uma solução dual ótima, que por sua vez guia a construção de uma arborescência primal ótima através dos arcos justos.

2.1.3 Arcos justos podem conter “sujeira”

Um aspecto sutil mas importante do algoritmo é que o conjunto A_0 de arcos justos obtido na Fase 1 pode conter **mais arcos do que necessário** para formar uma arborescência. Especificamente:

- Uma r -arborescência em um dígrafo com n vértices contém exatamente $n - 1$ arcos.
- O conjunto A_0 pode conter ciclos formados por arcos justos.
- Mesmo sem ciclos, A_0 pode conter múltiplos arcos entrando em um mesmo vértice.

Essa “sujeira” não é um problema — é uma característica do algoritmo. Durante a Fase 1, todos os arcos em A_0 são garantidamente parte de *alguma* arborescência ótima (não necessariamente a mesma), mas apenas um subconjunto de A_0 formará a arborescência específica que o algoritmo constrói.

2.1.3.0.1 Contração de ciclos.

Quando A_0 contém um ciclo, a operação de contração (ilustrada na Figura 31) é necessária para eliminar essa ambiguidade. Ao contrair o ciclo em um supervértice e resolver recursivamente, o algoritmo efetivamente escolhe quais arcos do ciclo fazer parte da solução final. Durante a expansão, apenas $|C| - 1$ arcos do ciclo contraído C são incluídos na arborescência (onde $|C|$ é o número de vértices no ciclo), descartando exatamente um arco — aquele que entra no vértice por onde a arborescência alcança o ciclo.

2.1.3.0.2 Construção incremental.

Mesmo quando A_0 não contém ciclos, a Fase 2 do algoritmo constrói a arborescência *incrementalmente*, adicionando arcos de A_0 um por vez, garantindo que cada vértice não-raiz receba exatamente um arco de entrada. O algoritmo sempre escolhe um arco $(u, v) \in A_0$ tal que u já está na arborescência parcial e v ainda não está. Esse processo naturalmente seleciona um subconjunto de A_0 sem ambiguidades, descartando arcos "extras".

A presença dessa sujeira demonstra que a Fase 1 do algoritmo de Frank não constrói diretamente a arborescência, mas sim um *certificado de otimalidade* através dos arcos justos e potenciais duais. A Fase 2 então utiliza esse certificado para extrair uma arborescência ótima específica.

2.1.4 Evolução das componentes fortemente conexas

Durante a execução da Fase 1, as componentes fortemente conexas (SCCs) formadas pelos arcos justos evoluem de forma dinâmica e sistemática. Compreender essa evolução é essencial para entender por que o algoritmo termina corretamente.

2.1.4.0.1 Estado inicial.

No início do algoritmo, $A_0 = \emptyset$ (nenhum arco justo), portanto cada vértice forma sua própria SCC trivial. O grafo de condensação tem n nós (um para cada vértice original), e todas as SCCs exceto a raiz são fontes.

2.1.4.0.2 Após cada elevação de potenciais.

Quando escolhemos uma fonte (conjunto minimal) X e elevamos seus potenciais por $\Delta(X)$:

1. Ao menos um arco entrando em X torna-se justo (aquele com custo reduzido original igual a $\Delta(X)$).
2. Esse novo arco justo conecta X a alguma outra componente pré-existente no grafo de arcos justos.
3. As SCCs podem se **fundir**: se o novo arco justo cria um caminho bidirecional entre duas SCCs anteriormente separadas, elas se tornam uma única SCC maior.

2.1.4.0.3 Progressão monotônica.

A cada iteração, o número de SCCs no grafo de arcos justos diminui (ou permanece igual):

- No melhor caso, múltiplas SCCs se fundem em uma única componente maior.
- No pior caso, uma SCC deixa de ser fonte (passa a ter arcos justos entrando) mas não se funde com outras.
- **Importante:** O número de SCCs nunca aumenta, pois adicionamos arcos mas nunca os removemos.

2.1.4.0.4 Condição de término.

O algoritmo termina quando existe exatamente **uma fonte** no grafo de condensação das SCCs formadas pelos arcos justos. Essa condição significa:

- A fonte única é a SCC que contém a raiz r .
- Todas as outras SCCs possuem arcos justos entrando (não são fontes).
- Todo vértice $v \neq r$ possui ao menos um arco justo entrando nele.

Quando essa condição é satisfeita, o grafo (V, A_0) formado pelos arcos justos contém uma r -arborescência, pois:

1. Todo vértice não-raiz é alcançável a partir de r através de arcos justos (consequência de haver apenas uma fonte).
2. Existe uma seleção de $n - 1$ arcos de A_0 que formam uma arborescência (possivelmente com sujeira adicional em A_0).

2.1.4.0.5 Exemplo visual.

Na Figura 29, após a primeira iteração de reduções, o grafo de condensação possui as seguintes SCCs:

- $\{r, a\}$: componente contendo a raiz, alcançável via arco justo (r, a) .
- $\{b\}$: componente trivial, alcançável via arco justo (r, b) .
- $\{e\}$: componente trivial, alcançável via arco justo (b, e) .
- $\{c, d\}$: componente não-trivial (ciclo justo interno), **fonte** (sem arcos justos entrando).

O grafo de condensação tem um arco de $\{r, a\}$ para $\{b\}$, de $\{b\}$ para $\{e\}$, mas nenhum arco entrando em $\{c, d\}$, que portanto é uma fonte e representa o conjunto minimal a ser processado na próxima iteração. Após elevar os potenciais de $\{c, d\}$ (Figura 30), um novo arco justo (a, d) é criado, conectando $\{c, d\}$ à componente $\{r, a\}$.

Nesse momento, todas as SCCs exceto a que contém r possuem arcos justos entrando, satisfazendo a condição de término.

Essa evolução garante que o algoritmo progride de forma finita: como cada iteração processa ao menos uma fonte (exceto a raiz) e o número de SCCs é limitado por n , o algoritmo termina em no máximo $O(n)$ iterações da Fase 1.

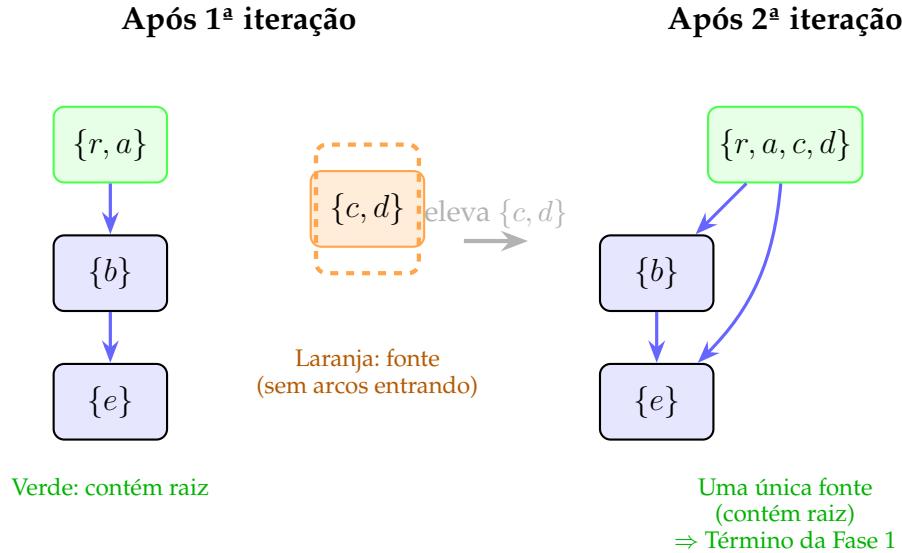


Figura 33 – Evolução do grafo de condensação durante a Fase 1 do algoritmo de Frank. À esquerda: após a primeira iteração, temos 4 SCCs formadas pelos arcos justos. A SCC $\{r, a\}$ (em verde) contém a raiz. A SCC $\{c, d\}$ (em laranja) é uma fonte (sem arcos justos entrando) e portanto um conjunto minimal. À direita: após elevar os potenciais de $\{c, d\}$, o arco (a, d) torna-se justo, conectando $\{c, d\}$ a $\{r, a\}$. As SCCs se fundem em $\{r, a, c, d\}$. Agora existe apenas uma fonte (a que contém r), satisfazendo a condição de término da Fase 1.

2.2 Descrição do algoritmo

Apresentamos agora uma descrição formal do algoritmo de András Frank.

O algoritmo opera em duas fases. A Fase 1 reduz progressivamente os custos dos arcos, criando arcos justos (de custo zero) através da identificação de subconjuntos minimais. A Fase 2 constrói a arborescência a partir dos arcos justos, usando contração e expansão quando necessário.

A diferença fundamental em relação ao Chu–Liu–Edmonds está na Fase 1:

- **Chu–Liu:** Para cada vértice v , subtrai o menor custo entrando em v . Processa vértice a vértice, uma vez.
- **Frank:** A cada iteração, identifica componentes fortemente conexas no grafo de arcos justos. Para cada componente sem arcos justos entrando (subconjunto

minimal X), subtrai $\delta(X)$ de todos os arcos entrando em X . Repete até todos os vértices terem arcos justos entrando.

A Fase 2 é idêntica em ambos os algoritmos: se há ciclos nos arcos justos, contrai, resolve recursivamente, e expande.

Algoritmo 2.1: András Frank

Entrada: dígrafo $D = (V, A)$, custos $c : A \rightarrow \mathbb{R}_{\geq 0}$, raiz r .

Fase 1: Redução de custos e construção de A_0

1. **Inicialização:** defina $A_0 := \emptyset$ (conjunto de arcos justos).
2. **Iteração:** enquanto existir subconjunto minimal $X \subseteq V \setminus \{r\}$:
 - Calcule as componentes fortemente conexas de (V, A_0) .
 - Para cada componente X que não contém r e não possui arcos de A_0 entrando:
 - Calcule $\delta(X) := \min\{c(u, v) : u \notin X, v \in X\}$.
 - Para todo arco (u, v) com $u \notin X$ e $v \in X$, atualize $c(u, v) := c(u, v) - \delta(X)$.
 - Adicione a A_0 todos os arcos (u, v) com $u \notin X, v \in X$ que atingiram custo zero.

Ao final, todo vértice $v \neq r$ possui ao menos um arco justo entrando.

Fase 2: Construção da arborescência

3. Se (V, A_0) forma uma r-arborescência, devolva A_0 . Por construção, todos os arcos têm custo reduzido zero e os demais arcos têm custo não negativo, garantindo otimalidade.
4. Caso contrário, identifique um ciclo dirigido C em A_0 (com $r \notin C$). **Contração:** contraia C em um supervértice x_C e ajuste os custos dos arcos incidentes, descartando laços em x_C e permitindo paralelos. Denote o dígrafo contraído por $D' = (V', A')$.
5. **Recursão:** compute uma r-arborescência ótima T' de D' com os custos ajustados.
6. **Expansão:** seja $(u, x_C) \in T'$ o único arco que entra em x_C . No dígrafo original, ele corresponde a (u, w) com $w \in C$. Forme

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

onde a_w é o arco de C que entra em w . Então T é uma r-arborescência ótima de D .

^a Se algum $v \neq r$ não possui arco de entrada, não existe r-arborescência.

2.2.0.0.1 Observação importante sobre a Fase 1.

A descrição acima enfatiza que em cada iteração devemos escolher um **subconjunto minimal**, não um subconjunto arbitrário sem arcos justos entrando. Esta distinção é crucial para a corretude do algoritmo:

- Se escolhermos um conjunto X que não seja minimal (ou seja, que contenha um subconjunto próprio $Y \subset X$ também sem arcos justos entrando), podemos elevar potenciais de forma redundante.
- Elevações redundantes podem criar folga desnecessária nas restrições duais, potencialmente levando a uma solução dual que não é ótima.
- A implementação através de componentes fortemente conexas e fontes no grafo de condensação garante automaticamente que apenas conjuntos minimais são processados.

Como discutido anteriormente nas Seções 2.1.1 e 2.1.2 (onde `\label{sec:...}` será adicionado nas subseções anteriores), as fontes no grafo de condensação correspondem exatamente aos conjuntos minimais que devemos processar.

2.2.0.0.2 Observação sobre a Fase 2.

A descrição teórica acima apresenta a Fase 2 com contração e recursão (Passos 4–6), seguindo a estrutura clássica do algoritmo de Chu–Liu–Edmonds. No entanto, na implementação apresentada neste capítulo, adotamos uma abordagem alternativa:

- A Fase 1 continua elevando potenciais iterativamente até que o grafo de arcos justos (V, A_0) contenha uma r -arborescência (possivelmente com arcos extras).
- A Fase 2 constrói a arborescência de forma **incremental** a partir de A_0 , sem recursão explícita.
- Apresentamos duas versões da construção incremental: uma versão ingênua $O(nm)$ e uma versão otimizada $O(m \log m)$ usando fila de prioridade.

Ambas as abordagens (recursiva vs. incremental) são corretas e produzem arborescências ótimas. A escolha entre elas é uma questão de implementação e eficiência prática.

2.2.1 Corretude

A corretude do algoritmo baseia-se em três ideias principais:

1. *Equivalência com potenciais duais:* A operação de subtrair $\delta(X)$ dos arcos entrando em X equivale a aumentar um potencial dual $y(v)$ em $\delta(X)$ para cada $v \in X$. Trabalhar com custos reduzidos $c(u, v)$ é equivalente a trabalhar com $c_{\text{original}}(u, v) - y(v)$.
2. *Condições de otimalidade:* Uma r -arborescência T é ótima se, e somente se:
 - Todos os arcos de T são justos (custo reduzido zero).
 - Todos os arcos do dígrafo têm custo reduzido não negativo.

Isso porque, para qualquer r -arborescência F ,

$$c(F) = \sum_{v \neq r} c_y(a_v) + \sum_{v \neq r} y(v),$$

onde a_v é o arco de F entrando em v . Como $\sum_{v \neq r} y(v)$ é constante, minimizar $c(F)$ equivale a minimizar $\sum_{v \neq r} c_y(a_v)$.

3. *Identificação de subconjuntos mínimos:* O algoritmo usa componentes fortemente conexas para identificar quais conjuntos ainda precisam de arcos justos entrando. Inicialmente cada vértice é sua própria componente. Após criar arcos justos, se formarem ciclos, as componentes agrupam vértices e o processo continua sobre esses conjuntos maiores.

Conclusão: A operação é a mesma do Chu–Liu–Edmonds (subtrair o menor custo entrando em cada conjunto), mas organizada diferentemente: Chu–Liu opera vértice a vértice; Frank opera sobre componentes fortemente conexas.

2.2.2 Complexidade

A implementação, baseada em componentes fortemente conexas, detecta em cada iteração, quais conjuntos X necessitam elevação de potenciais. Calcular componentes fortemente conexas custa $O(n + m)$ usando algoritmos como Tarjan ou Kosaraju. Para cada componente (exceto a raiz), eleva-se o potencial calculando $\Delta(X)$ em $O(m)$, atualizando os custos reduzidos.

No pior caso, cada iteração reduz o número de componentes em pelo menos uma unidade, resultando em $O(n)$ iterações. Cada iteração processa todos os arcos para atualizar custos reduzidos e recalcular componentes, resultando em $O(nm)$ no total para a Fase 1. A Fase 2 constrói a arborescência percorrendo A_0 uma vez, custando $O(n)$.

O uso de memória é $O(n + m)$, incluindo as estruturas para armazenar o dígrafo, potenciais e componentes. A implementação a seguir adota a versão $O(nm)$

por simplicidade e está disponível no repositório do projeto ([<https://github.com/lorenypsum/GraphVisualizer>](https://github.com/lorenypsum/GraphVisualizer)).

2.2.3 Observações finais sobre o algoritmo

Antes de apresentar a implementação detalhada, é útil consolidar alguns pontos-chave sobre o comportamento e as propriedades do algoritmo de András Frank:

2.2.3.0.1 1. O algoritmo sempre termina.

A Fase 1 progride monotonicamente: a cada iteração, ao menos uma componente fortemente conexa (exceto a raiz) recebe um arco justo entrando e deixa de ser fonte. Como o número de componentes é limitado superiormente por n (o número de vértices) e cada iteração reduz o número de fontes em pelo menos uma unidade, o algoritmo termina em no máximo $O(n)$ iterações.

2.2.3.0.2 2. Conjuntos minimais garantem otimalidade.

A escolha de conjuntos minimais (fontes no grafo de condensação) não é apenas uma conveniência algorítmica, mas uma exigência para a corretude. Escolher um conjunto não-minimal poderia levar a elevações redundantes de potenciais, violando as condições de folga complementar da programação linear dual e potencialmente produzindo uma solução subótima.

2.2.3.0.3 3. A Fase 1 constrói um certificado, não a arborescência.

O conjunto A_0 de arcos justos produzido pela Fase 1 geralmente contém mais arcos do que os $n - 1$ necessários para uma arborescência. Essa "sujeira" é intencional: A_0 representa o conjunto de *todos* os arcos que podem fazer parte de *alguma* arborescência ótima, não necessariamente a mesma. A Fase 2 seleciona um subconjunto específico de A_0 que forma uma arborescência.

2.2.3.0.4 4. Duas abordagens para a Fase 2.

A literatura apresenta duas estratégias equivalentes para a Fase 2:

- **Abordagem recursiva (clássica):** Se A_0 contém ciclos, contrai cada ciclo em um supervértice, resolve recursivamente o problema no dígrafo contraído, e expande a solução de volta ao dígrafo original. Esta é a abordagem apresentada no pseudocódigo formal (Passos 4–6).

- **Abordagem incremental (implementação):** Continua a Fase 1 até que (V, A_0) contenha uma r -arborescência (possivelmente com arcos extras), e então constrói a arborescência de forma gulosa e incremental, adicionando arcos de A_0 um por vez. Esta é a abordagem adotada na implementação Python deste capítulo.

Ambas as abordagens são corretas e produzem arborescências de custo mínimo. A escolha entre elas é uma questão de preferência de implementação e eficiência prática em diferentes cenários.

2.2.3.0.5 5. Relação com Chu–Liu–Edmonds.

O algoritmo de Frank pode ser visto como uma generalização do algoritmo de Chu–Liu–Edmonds:

- **Chu–Liu–Edmonds:** Processa vértices individualmente ($X = \{v\}$ para cada $v \neq r$), elevando potenciais uma vez por vértice.
- **Frank:** Processa componentes fortemente conexas formadas por arcos justos, permitindo elevações para conjuntos X que podem conter múltiplos vértices (quando formam ciclos).

Essa generalização permite ao algoritmo de Frank lidar de forma mais natural com estruturas cíclicas que aparecem durante a elevação de potenciais, mantendo a mesma operação fundamental de redução de custos.

2.2.3.0.6 6. Eficiência através de componentes fortemente conexas.

O uso de componentes fortemente conexas e grafos de condensação é mais do que uma ferramenta teórica — é uma estratégia de implementação eficiente. Ao trabalhar com o grafo de condensação:

- Evitamos enumerar explicitamente todos os subconjuntos de vértices.
- Identificamos automaticamente apenas os conjuntos minimais (fontes).
- Detectamos quando o algoritmo deve terminar (uma única fonte contendo a raiz).

Bibliotecas modernas como NetworkX fornecem implementações eficientes de algoritmos de SCC (Tarjan, Kosaraju), tornando esta abordagem prática e elegante.

Com essas observações em mente, passamos agora à descrição detalhada da implementação em Python, que materializa todos esses conceitos em código executável.

2.3 Implementação em Python

Esta seção descreve a implementação do algoritmo de András Frank em Python, estruturada para refletir com precisão as duas fases formais discutidas anteriormente. A Fase 1 realiza a elevação de potenciais e identifica os arcos justos, enquanto a Fase 2 constrói a arborescência de custo mínimo a partir desses arcos. Utilizamos a biblioteca NetworkX para manipulação de digrafos, aproveitando suas funcionalidades para representar grafos, calcular componentes fortemente conexas e gerenciar atributos de arcos.

A entrada consiste em um dígrafo orientado $D = (V, A)$, com custos dos arcos registrados no atributo "w", e uma raiz $r \in V$. As hipóteses adotadas são: (i) o dígrafo 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 ; e (iii) todos os custos são não negativos.

A saída é um subdígagrafo 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.

A estrutura do código é modular: funções auxiliares tratam cada etapa do algoritmo — cálculo de componentes fortemente conexas, elevação de potenciais, construção do subdígagrafo A_0 e construção da arborescência final. Todas operam sobre objetos `nx.DiGraph` e são coordenadas por uma função principal que gerencia o fluxo das duas fases. As subseções seguintes detalham cada função auxiliar, abordando lógica, parâmetros, saídas e complexidade.

2.3.1 Construção do dígrafo D_0 inicial

Começamos escrevendo uma função que constrói o dígrafo inicial D_0 que será utilizado na Fase 1 do algoritmo. O dígrafo D_0 é inicializado como um grafo vazio contendo apenas os vértices do dígrafo original, sem arcos. Essa estrutura será gradualmente populada com arcos de custo reduzido zero à medida que os potenciais são elevados.

Recebe como entrada um dígrafo D (objeto `nx.DiGraph`). A implementação cria um novo dígrafo vazio D_zero (linha 2) e adiciona todos os vértices de D a D_zero (linhas 3-4), preservando a estrutura de vértices sem incluir arcos inicialmente.

A função devolve o dígrafo D_zero contendo todos os vértices de D mas nenhum arco. O dígrafo original D não é modificado. A complexidade é $O(n)$, onde $n = |V|$, pois itera sobre todos os vértices uma única vez.

Construção do dígrafo D_0 inicial

Constrói um dígrafo vazio contendo apenas os vértices de D , sem arcos. Este dígrafo será populado com arcos justos durante a elevação de potenciais.

```

1 def build_D_zero(D):
2     D_zero = nx.DiGraph()
3     for v in D.nodes():
4         D_zero.add_node(v)
5     return D_zero

```

2.3.2 Identificação de arcos entrando em conjunto X

Esta função auxiliar identifica todos os arcos que entram em um conjunto $X \subseteq V$, isto é, arcos (u, v) tais que $u \notin X$ e $v \in X$. Essa operação é fundamental para calcular o mínimo custo de entrada em X durante a elevação de potenciais.

Recebe como entrada um dígrafo D e um conjunto de vértices X . A implementação cria uma lista vazia arcs (linha 2) e itera sobre todos os arcos do dígrafo com seus dados (linha 3), incluindo o peso. Para cada arco (u, v, data) , verifica se $u \notin X$ e $v \in X$ (linha 4), adicionando à lista apenas os arcos que cruzam a fronteira de X (linha 5).

A função devolve uma lista de tuplas (u, v, data) representando os arcos que entram em X , onde data contém o atributo "w" com o peso do arco. A complexidade é $O(m)$, onde $m = |A|$, pois examina cada arco uma vez.

Identificação de arcos entrando em conjunto X

Identifica todos os arcos (u, v) do dígrafo D tais que $u \notin X$ e $v \in X$, devolvendo uma lista com as tuplas (u, v, data) onde data contém o peso do arco.

```

1 def get_arcs_entering_X(D, X):
2     arcs = []
3     for u, v, data in D.edges(data=True):
4         if u not in X and v in X:
5             arcs.append((u, v, data))
6     return arcs

```

A figura a seguir ilustra o funcionamento da função `get_arcs_entering_X` em um digrafo que vamos denotar por D_{32} . O dígrafo possui uma raiz r_0 conectada aos vértices u_1, u_2, u_3 . Os vértices em laranja pertencem ao conjunto $X = \{v_1, v_2, v_3\}$, e a função identifica apenas os arcos em vermelho, que saem de vértices fora de X e entram

em vértices dentro de X . Arcos da raiz, arcos internos a X , externos a X , ou saindo de X não são retornados.

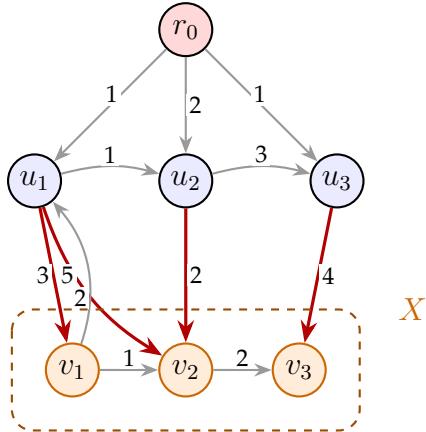


Figura 34 – Ilustração da função `get_arcs_entering_X` em D_{32} . A raiz r_0 (em vermelho claro) conecta-se aos vértices u_1, u_2, u_3 . Os vértices em laranja pertencem ao conjunto $X = \{v_1, v_2, v_3\}$. A função identifica apenas os arcos em vermelho: aqueles que saem de vértices fora de X e entram em vértices dentro de X . Arcos da raiz, arcos internos a X , externos a X , ou saindo de X não são retornados.

2.3.3 Cálculo do peso mínimo de corte

Esta função calcula o peso mínimo entre todos os arcos fornecidos, correspondendo ao valor $\Delta(X)$ necessário para elevar os potenciais dos vértices em X .

Recebe como entrada uma lista `arcos` de tuplas $(u, v, data)$. A implementação usa a função `min` com uma compreensão de gerador (linha 2) que extrai o atributo "w" de cada tupla em `data`.

A função devolve o peso mínimo encontrado entre todos os arcos da lista. A complexidade é $O(k)$, onde k é o número de arcos na lista, pois examina cada arco uma vez para encontrar o mínimo.

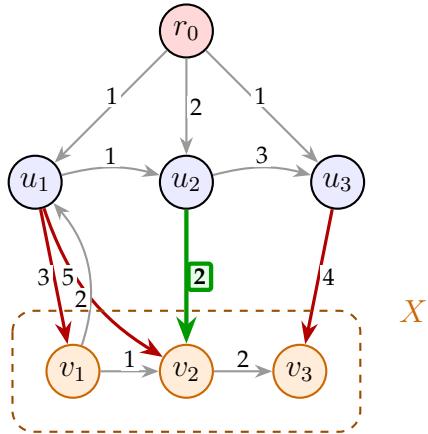
Cálculo do peso mínimo de corte

Calcula o peso mínimo entre todos os arcos fornecidos, correspondendo ao valor $\Delta(X)$ usado na elevação de potenciais.

```
1 def get_minimum_weight_cut(arcos):
2     return min(data["w"] for _, _, data in arcos)
```

A seguir temos uma ilustração do funcionamento da função `get_minimum_weight_cut` em D_{32} . Considerando os arcos em vermelho que entram em X (identificados pela

função anterior), esta função calcula o peso mínimo entre eles. O arco em verde possui o menor peso (2), correspondendo ao valor $\Delta(X) = 2$ que será devolvido pela função.



$$\Delta(X) = \min\{3, 2, 4, 5\} = 2$$

Figura 35 – Ilustração da função `get_minimum_weight_cut` em D_{32} . Considerando os arcos **em vermelho** que entram em X (identificados pela função anterior), esta função calcula o peso mínimo entre eles. O arco **em verde** possui o menor peso (2), correspondendo ao valor $\Delta(X) = 2$.

2.3.4 Atualização de pesos em X

Esta função auxiliar atualiza os pesos dos arcos que entram em um conjunto X , subtraindo o valor $\Delta(X)$ de cada peso. Arcos que atingem peso zero são adicionados a A_0 e a D_0 .

Recebe como entrada um dígrafo D , lista de `arcs` entrando em X , o valor `min_weight` a ser subtraído, uma lista `A_zero` para armazenar arcos de peso zero, e o dígrafo `D_zero` para adicionar os arcos justos.

A implementação itera sobre cada arco $(u, v, _)$ da lista (linha 2), subtrai `min_weight` do peso armazenado em $D[u][v][w]$ (linha 3), e verifica se o peso resultante é zero (linha 4). Caso sim, adiciona-se (u, v) à lista `A_zero` (linha 5) e ao dígrafo `D_zero` (linha 6).

A função não devolve valor, pois modifica diretamente as estruturas passadas como parâmetros: o dígrafo D tem seus pesos atualizados, `A_zero` acumula arcos justos, e `D_zero` é populado com esses arcos. A complexidade é $O(k)$, onde k é o número de arcos em `arcs`.

Atualização de pesos em X

Atualiza os pesos dos arcos que entram em X , subtraindo o valor mínimo. Arcos que atingem peso zero são registrados em A_0 e adicionados a D_0 .

```

1 def update_weights_in_X(D, arcs, min_weight, A_zero, D_zero):
2     for u, v, _ in arcs:
3         D[u][v]["w"] -= min_weight
4         if D[u][v]["w"] == 0:
5             A_zero.append((u, v))
6             D_zero.add_edge(u, v)

```

A seguir ilustramos o funcionamento da função `update_weights_in_X` em D_{32} . A figura mostra o dígrafo antes e depois da atualização dos pesos. No estado inicial (esquerda), temos $\Delta(X) = 2$. A função subtrai esse valor de todos os arcos que entram em X . No estado final (direita), os arcos que atingiram peso zero (destacados em azul) são adicionados a A_0 e D_0 .

Antes: pesos originais



$$\Delta(X) = 2$$

Depois: pesos atualizados



$$\text{Arco justo} \rightarrow A_0$$

Figura 36 – Ilustração da função `update_weights_in_X` em D_{32} . À esquerda, o dígrafo antes da atualização, com os arcos **em vermelho** entrando em X e $\Delta(X) = 2$. À direita, após subtrair $\Delta(X)$ de cada arco entrando em X : o peso (u_1, v_1) reduz de 3 para 1, (u_2, v_2) de 2 para 0 (torna-se justo), (u_3, v_3) de 4 para 2, e (u_1, v_2) de 5 para 3. O arco justo é adicionado a A_0 e D_0 . Note que os arcos da raiz e arcos internos/externos a X permanecem inalterados.

2.3.5 Verificação de arborescência

Esta função verifica se um dígrafo D contém uma r -arborescência com raiz r_0 . Utiliza busca em profundidade (DFS) a partir da raiz para verificar se todos os vértices são alcançáveis.

Recebe como entrada um dígrafo D e a raiz r_0 . A implementação constrói uma árvore DFS a partir de r_0 usando `nx.dfs_tree` (linha 2), que devolve um subdíggrafo contendo apenas os vértices alcançáveis a partir da raiz seguindo arcos. Em seguida, compara o número de vértices da árvore DFS com o número total de vértices de D (linha 3).

A função devolve `True` se todos os vértices são alcançáveis (indicando presença de r -arborescência), `False` caso contrário. A complexidade é $O(n + m)$, onde $n = |V|$ e $m = |A|$, devido à busca em profundidade. Precisamos dessa verificação para garantir que a Fase 1 produza um dígrafo D_0 que contenha uma r -arborescência antes de prosseguir para a Fase 2.

Verificação de arborescência

Verifica se o dígrafo D contém uma r -arborescência com raiz r_0 , usando busca em profundidade para testar alcançabilidade de todos os vértices.

```
1 def has_arborescence(D, r0):
2     tree = nx.dfs_tree(D, r0)
3     return tree.number_of_nodes() == D.number_of_nodes()
```

2.3.6 Fase 1: Elevação de potenciais e construção de A_0

A seguir apresentamos a função principal da Fase 1, responsável por elevar os potenciais dos vértices iterativamente até que cada conjunto de vértices possua ao menos um arco justo entrando. O processo utiliza componentes fortemente conexas para identificar quais conjuntos necessitam elevação.

Recebe como entrada um dígrafo D_{original} e a raiz r_0 . A implementação cria uma cópia do dígrafo original (linha 2) para preservar a entrada, inicializa estruturas auxiliares A_{zero} (lista de arcos justos), Dual_list (lista de pares $(X, \Delta(X))$ para fins de validação dual), e D_{zero} (dígrafo de arcos justos) (linhas 3-5). Um contador de iterações é inicializado na linha 6.

O loop principal (linhas 7-22) itera enquanto houver conjuntos sem arcos justos entrando. Em cada iteração, incrementa-se o contador (linha 8), calcula-se as componentes fortemente conexas de D_0 usando `nx.condensation` (linha 9), que devolve um grafo acíclico dirigido (DAG, do inglês *directed acyclic graph*) onde cada vértice representa uma componente e contém o atributo "members" com os vértices originais. Utilizamos componentes fortemente conexas porque elas identificam naturalmente os conjuntos maiores de vértices que ainda não possuem arcos justos entrando, evitando a necessidade de rastrear manualmente quais conjuntos já foram processados.

Em seguida, identificam-se as fontes (componentes sem arcos entrando) no grafo de condensação (linha 10). Se há apenas uma fonte, significa que todos os vértices estão em uma única componente alcançável pela raiz através de arcos justos, garantindo que D_0 contém uma r-arborescência e encerrando o loop (linhas 11-12).

Para cada fonte u no grafo de condensação (linha 13), obtém-se o conjunto X de vértices da componente (linha 14). Se $r_0 \in X$, a fonte é ignorada (linhas 15-16), pois a componente contendo a raiz não necessita elevação. Caso contrário, identificam-se os arcos entrando em X usando `get_arcs_entering_X` (linha 17), calcula-se o peso mínimo $\Delta(X)$ usando `get_minimum_weight_cut` (linha 18), e atualiza-se os pesos com `update_weights_in_X`, registrando novos arcos justos (linha 19). A elevação simultânea de potenciais para todos os vértices de X mantém a propriedade de que arcos internos a X permanecem com o mesmo custo reduzido relativo, preservando a correção do algoritmo. Finalmente, adiciona-se $(X, \Delta(X))$ à lista dual se $\Delta(X) > 0$ (linhas 20-21), permitindo verificação posterior das condições de otimalidade dual.

A função devolve `A_zero` (lista de arcos justos) e `Dual_list` (pares $(X, \Delta(X))$ para validação). A complexidade é $O(nm)$ no pior caso, com $O(n)$ iterações, cada uma custando $O(m)$ para calcular componentes e atualizar pesos.

Fase 1: Elevação de potenciais e construção de A_0

Eleva iterativamente os potenciais dos vértices até que cada conjunto possua ao menos um arco justo entrando. Devolve a lista A_0 de arcos justos e a lista de pares $(X, \Delta(X))$ para validação dual.

```

1 def phase1(D_original, r0):
2     D_copy = D_original.copy()
3     A_zero = []
4     Dual_list = []
5     D_zero = build_D_zero(D_copy)
6     iteration = 0
7     while True:
8         iteration += 1
9         C = nx.condensation(D_zero)
10        sources = [x for x in C.nodes() if C.in_degree(x) == 0]
11        if len(sources) == 1:
12            break
13        for u in sources:
14            X = C.nodes[u]["members"]
15            if r0 in X:
16                continue
17            arcs = get_arcs_entering_X(D_copy, X)
18            min_weight = get_minimum_weight_cut(arcs)
19            update_weights_in_X(D_copy, arcs, min_weight, A_zero, D_zero)
20            if min_weight != 0:
21                Dual_list.append((X, min_weight))
22    return A_zero, Dual_list

```

A seguir ilustramos a execução da função `phase1` no dígrafo D_{32} estendido com

a raiz r_0 . A figura mostra três estados principais: o dígrafo original D com seus pesos, o dígrafo D_0 inicial (vazio, sem arcos), e o dígrafo D_0 final após as iterações de elevação de potenciais, contendo apenas os arcos justos (em azul) que formam uma r-arborescência.

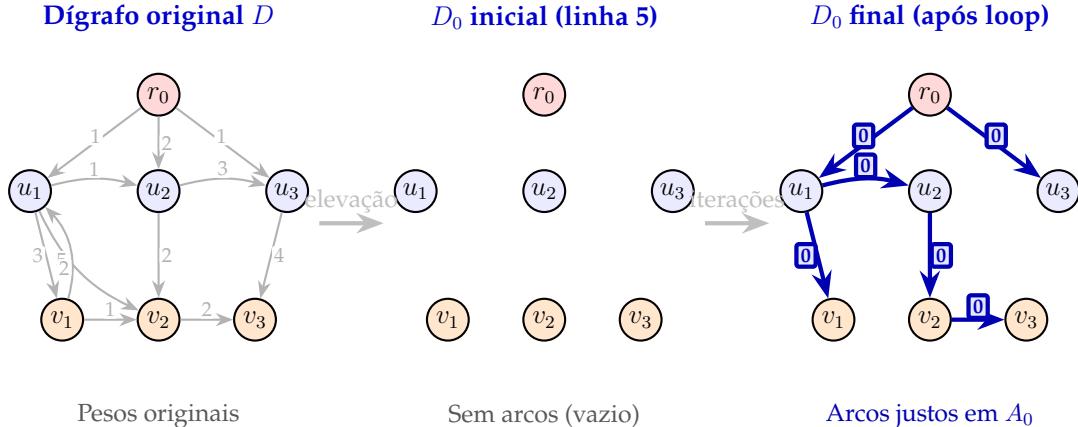


Figura 37 – Execução da função `phase1` em D_{32} . À esquerda, o dígrafo original D com pesos dos arcos. Ao centro, D_0 inicial apóis linha 5 (`build_D_zero`), contendo apenas vértices sem arcos. À direita, D_0 final apóis o loop de elevação de potenciais, contendo apenas os arcos **justos** (custo reduzido zero) que formam uma r-arborescência. Durante as iterações (linhas 7-22), os potenciais são elevados para cada componente sem arcos justos entrando, até que todos os vértices sejam alcançáveis a partir de r_0 através de arcos em D_0 .

A seguir ilustramos a execução da função `phase2` em D_{32} , mostrando como a r-arborescência é construída incrementalmente a partir do conjunto A_0 de arcos justos obtidos da Fase 1. A figura apresenta quatro estados: o conjunto inicial A_0 com 6 arcos justos, e três estados intermediários da arborescência Arb sendo construída, destacando em verde os arcos adicionados em cada iteração.

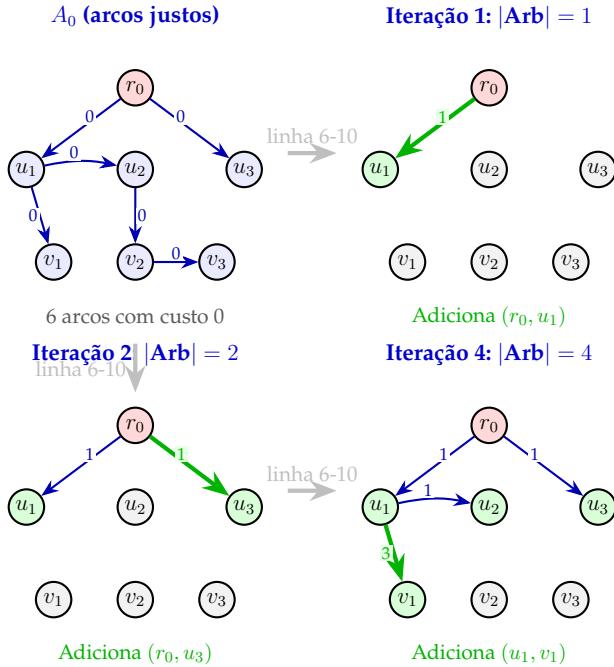


Figura 38 – Execução da função `phase2` em D_{32} . Superior esquerdo: conjunto A_0 com 6 arcos justos (custo reduzido zero) obtidos da Fase 1. Superior direito: Iteração 1 — adiciona arco (r_0, u_1) pois $r_0 \in \text{Arb}$ e $u_1 \notin \text{Arb}$. Inferior esquerdo: Iteração 2 — adiciona (r_0, u_3) . Inferior direito: Iteração 4 — após adicionar (u_1, u_2) na iteração 3, adiciona (u_1, v_1) . Vértices verdes foram recém-adicionados, vértices cinzas ainda não pertencem a Arb. O processo continua até que todos os 6 arcos de A_0 sejam incluídos, formando uma r-arborescência com pesos originais de D .

As funções auxiliares implementadas nesta seção correspondem diretamente aos passos da Fase 1 do algoritmo de András Frank da seguinte forma:

Correspondência entre Teoria e Implementação — Fase 1

Passo 1 — Inicialização:

- **Descrição teórica:** Defina $y(v) := 0$ para todo $v \in V$.
- **Implementação:** A função `build_D_zero(D)` inicializa o dígrafo D_0 vazio, que será populado apenas com arcos justos. Implicitamente, os potenciais iniciam em zero, pois os pesos no dígrafo D representam os custos reduzidos $c_y(u, v) = c(u, v) - y(v)$. Com $y(v) = 0$, temos $c_y = c$ no início.

Passo 2 — Iteração de elevação de potenciais:

- **Descrição teórica:** Enquanto existir conjunto $X \subseteq V \setminus \{r\}$ sem arco justo entrando:
 - Calcule $\Delta(X) := \min\{c(u, v) - y(v) : u \notin X, v \in X\}$.

- Para cada $v \in X$, atualize $y(v) := y(v) + \Delta(X)$.
- **Implementação:** Esse processo de iteração é realizado pela composição sequencial de três funções auxiliares:
 1. `get_arcs_entering_X(D, X)`: Identifica o conjunto $\{(u, v) : u \notin X, v \in X\}$, isto é, todos os arcos que cruzam a fronteira de X . Essa função corresponde diretamente à definição do conjunto sobre o qual o mínimo é calculado na fórmula $\Delta(X)$.
 2. `get_minimum_weight_cut(arcs)`: Calcula $\min\{data["w"] : (u, v, data) \in arcs\}$, que é exatamente $\Delta(X) = \min\{c_y(u, v) : u \notin X, v \in X\}$. Como os pesos no dígrafo já representam custos reduzidos (são atualizados a cada iteração), essa função devolve precisamente o valor teórico de $\Delta(X)$.
 3. `update_weights_in_X(D, arcs, min_weight, A_zero, D_zero)`: Implementa a atualização dos potenciais. Para cada arco (u, v) entrando em X , subtrai `min_weight` de $D[u][v]["w"]$, efetivamente calculando o novo custo reduzido:

$$c'_y(u, v) = c_y(u, v) - \Delta(X) =$$

$$c(u, v) - y(v) - \Delta(X) =$$

$$c(u, v) - (y(v) + \Delta(X)) = c(u, v) - y'(v),$$

onde $y'(v) = y(v) + \Delta(X)$ é o novo potencial. Arcos cujo custo reduzido atinge zero ($D[u][v]["w"] == 0$) são adicionados a A_0 e D_0 , tornando-se **justos**.

Passo 3 — Construção de A_0 :

- **Descrição teórica:** Defina $A_0 := \{a \in A : c_y(a) = 0\}$, o conjunto de arcos justos.
- **Implementação:** A construção de A_0 ocorre de forma incremental durante as iterações do Passo 2. Cada chamada a `update_weights_in_X` verifica quais arcos atingiram custo reduzido zero e os adiciona tanto à lista `A_zero` quanto ao dígrafo `D_zero`. Esse processo continua até que a função principal da Fase 1 (que será apresentada adiante) determine que cada componente fortemente conexa em D_0 (exceto a raiz) possui ao menos um arco justo entrando, garantindo que A_0 é suficiente para formar a base de uma r -arborescência.

- O resultado final é o conjunto completo $A_0 = \{a \in A : c_y(a) = 0\}$, usado na Fase 2 para construir a arborescência ótima através de uma seleção gulosa de arcos justos.

2.3.7 Fase 2: Construção da arborescência

Esta é a função principal da Fase 2, responsável por construir a r-arborescência de custo mínimo a partir do conjunto A_0 de arcos justos. A construção é incremental: inicia-se com a raiz e adiciona-se iterativamente arcos de A_0 que conectam vértices já incluídos a novos vértices, garantindo que cada vértice não-raiz receba exatamente um arco de entrada.

Recebe como entrada um dígrafo `D_original`, a raiz `r0`, e a lista `A_zero` de arcos justos. A implementação cria um novo dígrafo vazio `Arb` (linha 2) e adiciona a raiz (linha 3).

O loop principal (linhas 5-12) itera $n - 1$ vezes, onde $n = |V|$, pois uma r-arborescência tem exatamente $|V| - 1$ arcos. Em cada iteração:

1. Percorre os arcos (u, v) de `A_zero` (linha 6).
2. Verifica se u já está em `Arb` e v ainda não (linha 7).
3. Se sim, obtém os dados do arco do dígrafo original (linha 8) e adiciona (u, v) a `Arb` (linha 9).
4. Interrompe o loop interno para reiniciar a busca, garantindo descoberta em largura (linha 10).

A função devolve o dígrafo `Arb` representando a r-arborescência de custo mínimo. A complexidade é $O(nm)$ no pior caso, pois cada uma das $O(n)$ iterações pode percorrer todos os $O(m)$ arcos de `A_zero`.

Fase 2: Construção da arborescência

Constrói incrementalmente a r-arborescência a partir de A_0 , adicionando iterativamente arcos que conectam vértices já incluídos a novos vértices.

```

1 def phase2(D_original, r0, A_zero):
2     Arb = nx.DiGraph()
3     Arb.add_node(r0)
4     n = len(D_original.nodes())
5     for _ in range(n - 1):
6         for u, v in A_zero:

```

```

7         if u in Arb.nodes() and v not in Arb.nodes():
8             edge_data = D_original.get_edge_data(u, v)
9             Arb.add_edge(u, v, **edge_data)
10            break
11    return Arb

```

Apresentamos também uma versão alternativa da Fase 2 que utiliza busca em largura (BFS) para construir a r-arborescência de forma mais eficiente. Diferentemente da versão anterior que itera $n - 1$ vezes sobre todos os arcos, esta implementação usa uma fila de prioridade para explorar os arcos em ordem, evitando buscas lineares repetidas.

Recebe como entrada um dígrafo D_{original} , a raiz r_0 , e a lista A_{zero} de arcos justos. A implementação começa criando um dígrafo auxiliar Arb (linha 2) onde cada arco de A_{zero} recebe um peso igual ao seu índice na lista (linhas 3-4), estabelecendo uma ordem de exploração. Inicializa-se o conjunto V de vértices visitados contendo apenas a raiz (linha 5) e uma fila de prioridade vazia q (linha 6).

Todos os arcos que saem da raiz em Arb são adicionados à fila de prioridade (linhas 7-8), usando o peso (índice) como critério de ordenação. Cria-se então o dígrafo A que conterá a arborescência resultante (linha 9).

O loop principal (linhas 10-17) extraí arcos da fila de prioridade em ordem crescente de índice. Para cada arco (u, v) extraído (linha 11), verifica-se se o vértice destino v já foi visitado (linha 12); em caso positivo, o arco é ignorado via `continue` (linha 13). Caso contrário, adiciona-se o arco (u, v) à arborescência A com o peso original de D_{original} (linha 14), marca-se v como visitado (linha 15), e todos os arcos que saem de v em Arb são adicionados à fila de prioridade para exploração futura (linhas 16-17).

A função devolve o dígrafo A representando a r-arborescência de custo mínimo. A complexidade é $O(m \log m)$, onde $m = |A_0|$, devido às operações de inserção e remoção na fila de prioridade. Esta versão é mais eficiente que a anterior quando $|A_0|$ é grande, pois evita percorrer todos os arcos em cada iteração.

Fase 2 (versão BFS): Construção da arborescência com fila de prioridade

Constrói a r-arborescência usando busca em largura guiada por fila de prioridade, explorando arcos de A_0 em ordem e evitando buscas lineares repetidas. Complexidade $O(m \log m)$.

```

1 def phase2_v2(D_original, r0, A_zero):
2     Arb = nx.DiGraph()
3     for i, (u, v) in enumerate(A_zero):

```

```

4      Arb.add_edge(u, v, w=i)
5  V = {r0}
6  q = []
7  for u, v, data in Arb.out_edges(r0, data=True):
8      heapq.heappush(q, (data["w"], u, v))
9  A = nx.DiGraph()
10 while q:
11     _, u, v = heapq.heappop(q)
12     if v in V:
13         continue
14     A.add_edge(u, v, w=D_original[u][v]["w"])
15     V.add(v)
16     for x, y, data in Arb.out_edges(v, data=True):
17         heapq.heappush(q, (data["w"], x, y))
18 return A

```

As duas versões da Fase 2 implementadas acima correspondem diretamente ao Passo 4 da descrição teórica do algoritmo de András Frank:

Correspondência entre Teoria e Implementação — Fase 2

Passo 4 — Construção da arborescência (caso acíclico):

- **Descrição teórica:** Se (V, A_0) forma uma r-arborescência, devolva A_0 . Por otimalidade dos potenciais duais, trata-se de uma r-arborescência de custo mínimo.
- **Implementação:** Ambas as versões de phase2 constroem uma r-arborescência a partir do conjunto A_0 de arcos justos obtido na Fase 1. A corretude baseia-se no fato de que todos os arcos em A_0 têm custo reduzido zero, e a Fase 1 garante que existe uma r-arborescência formada exclusivamente por arcos justos.
 - **Versão 1 (phase2):** Construção incremental por exploração exaustiva. Em cada uma das $n - 1$ iterações, percorre todos os arcos de A_0 procurando um arco (u, v) tal que u já pertence à arborescência parcial e v ainda não. Essa abordagem simples corresponde diretamente à ideia teórica de construir a arborescência adicionando um vértice por vez, conectando-o à estrutura existente através de um arco justo. Complexidade: $O(nm)$.
 - **Versão 2 (phase2_v2):** Construção por busca em largura guiada por fila de prioridade. Cria um dígrafo auxiliar onde arcos são indexados, usa

a fila de prioridade para explorar arcos sistematicamente a partir da raiz, evitando buscas lineares repetidas. Essa versão otimizada mantém a mesma correção teórica — construir uma r-arborescência usando apenas arcos de A_0 — mas melhora a eficiência prática. Complexidade: $O(m \log m)$.

Nota sobre Passos 5-7 (contração/recursão/expansão):

- A descrição teórica do algoritmo de András Frank inclui os Passos 5-7 para tratar o caso onde A_0 contém ciclos dirigidos, exigindo contração, resolução recursiva e reexpansão, de forma análoga ao algoritmo de Chu-Liu–Edmonds.
- Na implementação apresentada, optamos por uma abordagem não-recursiva baseada em componentes fortemente conexas. A Fase 1 já garante que A_0 formará uma r-arborescência ao término das iterações de elevação de potenciais, eliminando a necessidade de tratar ciclos explicitamente na Fase 2. Essa simplificação é possível porque a elevação de potenciais progressivamente "quebra" todos os ciclos ao criar novos arcos justos que conectam diferentes componentes, até que reste apenas uma única componente fortemente conexa contendo todos os vértices.
- Portanto, quando a Fase 2 é executada, o conjunto A_0 já está livre de ciclos e forma uma r-arborescência, correspondendo diretamente ao caso tratado pelo Passo 4 da descrição teórica. A verificação prévia `has_arborescence(D, r0)` (realizada pela função principal) confirma essa propriedade antes de invocar a Fase 2.

2.3.8 Verificação de otimalidade dual

Esta função verifica se a condição de otimalidade dual é satisfeita para a r-arborescência construída. Segundo a teoria de programação linear dual aplicada ao problema de arborescência de custo mínimo, uma solução é ótima se e somente se cada conjunto $X \subseteq V \setminus \{r\}$ que teve seu potencial elevado durante a Fase 1 possui exatamente um arco entrando na arborescência final.

Recebe como entrada a arborescência `Arb` e a lista `Dual_list` contendo pares $(X, \Delta(X))$ onde X é um conjunto de vértices cujos potenciais foram elevados e $\Delta(X) > 0$ é o valor da elevação. A implementação itera sobre cada par (X, z) na lista dual (linha 2), e para cada conjunto X , percorre todos os arcos (u, v) da arborescência (linha 3). Inicializa um contador `count` em zero (linha 4) e verifica se o arco cruza a fronteira de

X , isto é, se $u \notin X$ e $v \in X$ (linha 5). Quando essa condição é satisfeita, incrementa o contador (linha 6) e imediatamente verifica se já há mais de um arco entrando em X (linha 7). Caso positivo, a condição de optimalidade dual é violada e a função devolve `False` (linha 8).

A função devolve `True` se todos os conjuntos em `Dual_list` possuem exatamente um arco entrando na arborescência, confirmando que a solução satisfaz as condições de folga complementar da programação linear dual. A complexidade é $O(km)$, onde $k = |\text{Dual_list}|$ e $m = |A|$, pois para cada conjunto dual verifica-se todos os arcos da arborescência.

Esta verificação é fundamental para garantir a correção do algoritmo: a Fase 1 constrói uma solução dual viável (potenciais $y(v)$), a Fase 2 constrói uma solução primal viável (arborescência), e esta função confirma que ambas satisfazem as condições de folga complementar, implicando optimalidade pelo teorema da dualidade forte.

Verificação de optimalidade dual

Verifica se a condição de optimalidade dual é satisfeita, confirmando que cada conjunto dual possui exatamente um arco entrando na arborescência.

```

1 def check_dual_optimality_condition(Arb, Dual_list):
2     for X, z in Dual_list:
3         count = 0
4         for u, v in Arb.edges():
5             if u not in X and v in X:
6                 count += 1
7             if count > 1:
8                 return False
9     return True

```

2.3.9 O algoritmo completo de András Frank

Finalmente, apresentamos a função principal que implementa o algoritmo de András Frank para encontrar uma r-arborescência de custo mínimo em um dígrafo com pesos. A função integra as fases de construção dos potenciais duais, obtenção dos arcos justos, construção da arborescência e verificação de optimalidade dual.

Verificação de optimalidade dual

Implementa o algoritmo completo de András Frank, integrando as fases de construção dos potenciais duais, obtenção dos arcos justos, construção da arborescência e verificação de

optimalidade dual.

```

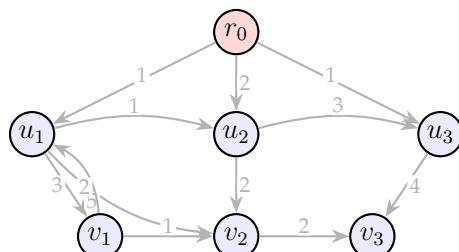
1 def andras_frank_algorithm(D):
2     A_zero, Dual_list = phase1(D, "r0")
3     arborescence_frank = phase2(D, "r0", A_zero)
4     arborescence_frank_v2 = phase2_v2(D, "r0", A_zero)
5     dual_frank = check_dual_optimality_condition(
6         arborescence_frank, Dual_list)
7     dual_frank_v2 = check_dual_optimality_condition(
8         arborescence_frank_v2, Dual_list)
9     return arborescence_frank, arborescence_frank_v2, dual_frank,
       dual_frank_v2

```

Exemplo de execução do algoritmo

Aqui ilustraremos cada fase do algoritmo de András Frank: inicialização dos potenciais, iterações de elevação, construção de A_0 , e formação da arborescência final. Utilizaremos o dígrafo D_{32} estendido com a raiz r_0 , mostrado na Figura 26.

Dígrafo inicial D com raiz r_0



Pesos originais $c(u, v)$

Figura 39 – Dígrafo inicial D com raiz r_0 e 6 vértices adicionais. O dígrafo contém múltiplos arcos com pesos variados. A Fase 1 do algoritmo iniciará com potenciais $y(v) = 0$ para todos os vértices, correspondendo aos custos reduzidos iniciais $c_y(u, v) = c(u, v)$.

Fase 1 — Iteração 1:

O algoritmo inicia com todos os potenciais $y(v) = 0$ e D_0 vazio (sem arcos justos). Calcula-se as componentes fortemente conexas de D_0 : cada vértice forma sua própria componente. As fontes são todos os vértices exceto r_0 . Para o conjunto $X = \{u_1, u_2, u_3\}$, identifica-se os arcos entrando: (r_0, u_1) com peso 1, (r_0, u_2) com peso 2, (r_0, u_3) com peso

1. O mínimo é $\Delta(X) = 1$. Subtraindo esse valor, os arcos (r_0, u_1) e (r_0, u_3) tornam-se justos (peso 0) e são adicionados a A_0 .

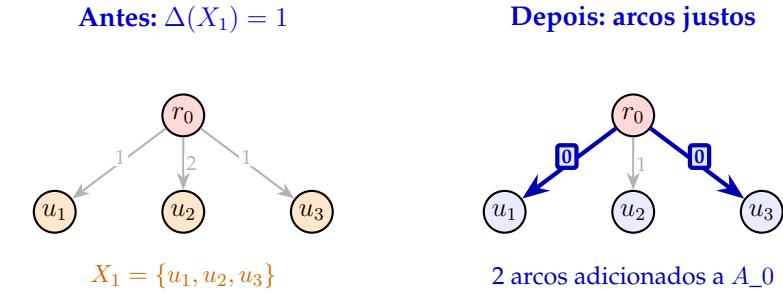


Figura 40 – Fase 1, Iteração 1: Elevação de potenciais para $X_1 = \{u_1, u_2, u_3\}$. À esquerda, o conjunto X_1 (em laranja) sem arcos justos entrando. Calcula-se $\Delta(X_1) = \min\{1, 2, 1\} = 1$. À direita, após subtrair $\Delta(X_1)$, os arcos (r_0, u_1) e (r_0, u_3) tornam-se **justos** (custo reduzido 0) e são adicionados a A_0 .

Fase 1 — Iteração 2:

Com $A_0 = \{(r_0, u_1), (r_0, u_3)\}$, as componentes fortemente conexas são: $\{r_0, u_1, u_3\}$ e $\{u_2\}$, $\{v_1\}$, $\{v_2\}$, $\{v_3\}$. A fonte (excluindo a raiz) é $\{u_2\}$. Para $X = \{u_2\}$, o único arco entrando é (r_0, u_2) com peso atual 1 (após iteração anterior). Portanto $\Delta(X) = 1$, e após subtração, (r_0, u_2) torna-se justo. Também na mesma iteração, para $X = \{v_1, v_2, v_3\}$, calcula-se $\Delta(X) = \min\{3, 2, 4, 5\} = 2$, e os arcos são atualizados conforme ilustrado.

Iteração 2: $\Delta(\{v_1, v_2, v_3\}) = 2$

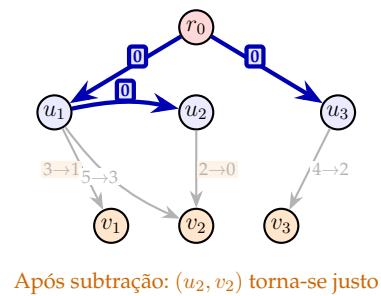
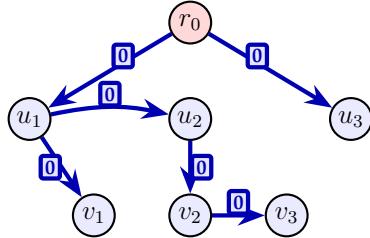


Figura 41 – Fase 1, Iteração 2: Para o conjunto $X = \{v_1, v_2, v_3\}$ (em laranja), calcula-se $\Delta(X) = \min\{3, 2, 4, 5\} = 2$. Após subtrair esse valor dos arcos entrando em X , o arco (u_2, v_2) atinge custo reduzido 0 e é adicionado a A_0 . Note que (u_1, v_1) também se tornou justo na mesma iteração ao processar $X = \{u_2\}$.

Fase 1 — Estado final:

Após mais iterações elevando potenciais para os conjuntos restantes, o dígrafo D_0 de arcos justos fica completo, contendo uma r-arborescência. A Figura 42 mostra o conjunto final A_0 com 6 arcos de custo reduzido zero.

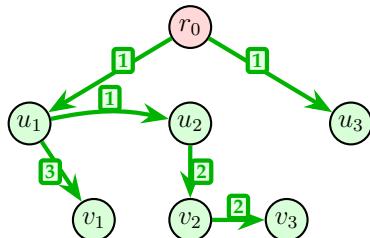
A_0 final: arcos justos apóis Fase 1

$$|A_0| = 6 \text{ arcos com custo reduzido } c_y = 0$$

Figura 42 – Conjunto final A_0 de arcos justos apóis conclusão da Fase 1. Todos os 6 arcos **destacados** possuem custo reduzido zero: $(r_0, u_1), (r_0, u_3), (u_1, u_2), (u_1, v_1), (u_2, v_2), (v_2, v_3)$. O dígrafo formado por esses arcos é acíclico e alcança todos os vértices a partir de r_0 , satisfazendo as condições para prosseguir à Fase 2.

Fase 2 — Construção da arborescência:

Com A_0 completo e acíclico, a Fase 2 constrói incrementalmente a arborescência final. Inicia-se com $\text{Arb} = \{r_0\}$ e em cada iteração adiciona-se um arco $(u, v) \in A_0$ tal que $u \in \text{Arb}$ e $v \notin \text{Arb}$. A Figura 43 mostra a arborescência resultante.

Arborescência final (Fase 2)

$$\text{Custo total: } 1 + 1 + 1 + 3 + 2 + 2 = 10$$

Figura 43 – Arborescência de custo mínimo final obtida pela Fase 2. Os pesos mostrados são os *originais* de D , restaurados apóis a construção. Cada vértice não-raiz possui exatamente um arco de entrada, não há ciclos, e todos os vértices são alcançáveis a partir de r_0 . A verificação de otimalidade dual confirma que cada conjunto X que teve potenciais elevados possui exatamente um arco da arborescência cruzando sua fronteira, garantindo que a solução é ótima.

Verificação de otimalidade dual:

A função `check_dual_optimality_condition` confirma que para cada par $(X, \Delta(X))$ em `Dual_list` (conjuntos cujos potenciais foram elevados com $\Delta(X) > 0$), existe exatamente um arco da arborescência final cruzando a fronteira de X . Essa condição, juntamente com os arcos justos, garante que as condições de folga complementar da

programação linear dual são satisfeitas, implicando que a arborescência encontrada é de custo mínimo global.

2.3.10 Correspondência entre teoria e implementação

A implementação em Python do algoritmo de András Frank segue fielmente a descrição teórica primal-dual apresentada anteriormente. A tabela abaixo estabelece o paralelo direto entre os passos teóricos e sua realização no código:

Descrição Teórica	Implementação Python
Passo 1: Inicialização Defina $y(v) := 0$ para todo $v \in V$. Inicialize $A_0 := \emptyset$. Construa dígrafo vazio D_0 (arcos justos).	Função phase1 — Linhas 2–5: <pre>D_copy = D_original.copy() A_zero = [] D_zero = build_D_zero(D_copy) Potenciais $y(v) = 0$ implícitos, custos $c_y = c$.</pre>
Passo 2: Elevação de potenciais Enquanto $\exists X \subseteq V \setminus \{r\}$ sem arco justo: Calcule $\Delta(X) := \min\{c_y(u, v) : u \notin X, v \in X\}$ Atualize $y(v) := y(v) + \Delta(X), \forall v \in X$ Adicione arcos com $c_y = 0$ a A_0	Loop principal — Linhas 7–22: <pre>C = nx.condensation(D_zero) sources = [x for x in C.nodes()] if C.in_degree(x) == 0] Para fonte u (exceto raiz): X = C.nodes[u]["members"] arcs = get_arcs_entering_X(D, X) min_w = get_minimum_weight_cut(arcs) update_weights_in_X(D, arcs, min_w, A_zero, D_zero)</pre>
Passo 2(a): Identificar arcos entrando Determine $\{(u, v) \in A : u \notin X, v \in X\}$	Função get_arcs_entering_X: <pre>return [(u, v, data) for u, v, data in D.edges(data=True) if u not in X and v in X]</pre>
Passo 2(b): Calcular $\Delta(X)$ $\Delta(X) := \min\{c_y(u, v) : u \notin X, v \in X\}$	Função get_minimum_weight_cut: <pre>return min(data["w"]) for _, _, data in arcs)</pre>
Passo 2(c): Atualizar pesos Para (u, v) entrando em X : $c_y(u, v) := c_y(u, v) - \Delta(X)$ Se $c_y(u, v) = 0$, adicione a A_0	Função update_weights_in_X: <pre>for u, v, _ in arcs: D[u][v]["w"] -= min_weight if D[u][v]["w"] == 0: A_zero.append((u, v)) D_zero.add_edge(u, v)</pre>
Passo 3: Verificar término Se D_0 contém r-arborescência, encerre.	Condição — Linhas 11–12: <pre>if len(sources) == 1: break Uma fonte \Rightarrow r-arborescência acíclica.</pre>
Passo 4: Construir arborescência Construa F a partir de A_0 , conectando vértices incrementalmente.	Função phase2 (incremental): <pre>Arb = nx.DiGraph(); Arb.add_node(r0) for _ in range(n - 1): for u, v in A_zero: if u in Arb and v not in Arb: Arb.add_edge(u, v, **data) break Complexidade: $O(nm)$.</pre>
Passo 4: Versão otimizada Mesma ideia, com fila de prioridade.	Função phase2_v2 (BFS): <pre>Arb = nx.DiGraph() for i, (u, v) in enumerate(A_zero): Arb.add_edge(u, v, w=i) q = [] # fila de prioridade while q: _, u, v = heapq.heappop(q) if v in V: continue A.add_edge(u, v, w=D[u][v]["w"]) Complexidade: $O(m \log m)$.</pre>
Otimalidade dual Para cada X elevado ($\Delta(X) > 0$), exatamente um arco de F cruza $\delta^-(X)$.	Função check_dual_optimality: <pre>for X, z in Dual_list: count = 0 for u, v in Arb.edges(): if u not in X and v in X: count += 1 if count > 1: return False return True</pre>

Tabela 2 – Correspondência entre a descrição teórica do algoritmo de András Frank e sua implementação em Python. Cores: inicialização (azul), elevação de potenciais (verde/laranja/roxo/amarelo), verificação (ciano), construção (vermelho) e validação dual (rosa).

Esta correspondência demonstra que a implementação traduz fielmente a abordagem primal-dual em código executável. As funções auxiliares (`get_arcs_entering_X`, `get_minimum_weight_cut`, `update_weights_in_X`, `phase1`, `phase2`, `phase2_v2`, `check_dual_optimality_condition`) encapsulam exatamente as operações descritas na teoria, preservando as propriedades de correção e as garantias de otimalidade do algoritmo original. A utilização de componentes fortemente conexas (`nx.condensation`) para identificar conjuntos sem arcos justos entrando é uma implementação eficiente da verificação teórica, evitando enumeração explícita de todos os subconjuntos de vértices.

Referências

KLEINBERG, J.; TARDOS, É. *Algorithm Design*. [S.l.]: Addison-Wesley, 2006. Citado 4 vezes nas páginas [25](#), [26](#), [37](#) e [46](#).

SCHRIJVER, A. *Combinatorial Optimization: Polyhedra and Efficiency*. [S.l.]: Springer, 2003. Citado 2 vezes nas páginas [25](#) e [26](#).

Anexos

ANEXO A – Anexo A

Conteúdo do anexo A.