

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

Sumário

1 Algoritmo de Chu–Liu–Edmonds

Neste capítulo, apresentaremos o algoritmo de Chu–Liu–Edmonds, que determina uma arborescência de custo mínimo em um dígrafo 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 dígrafo, $c: A \rightarrow \mathbb{R}$ 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 -dígrafo 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.

1.1.1 Contração de ciclos

Suponha doravante que (D, c, r) é um r -dígrafo 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,

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

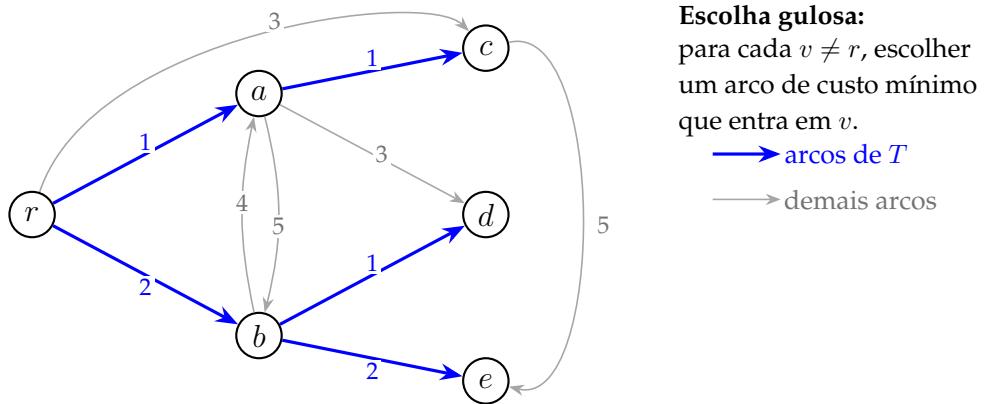


Figura 1 – A figura ilustra a escolha gulosa quando esta produz uma r -arborescência. Os arcos em azul são os escolhidos; os cinza são os demais arcos de digrafo.

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

A seguinte figura ilustra que podemos não ter tanta sorte.

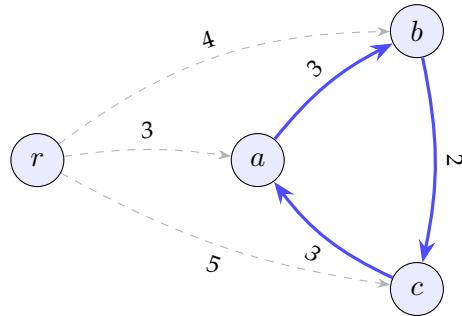


Figura 2 – Os arcos azuis são os da escolha gulosa.

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

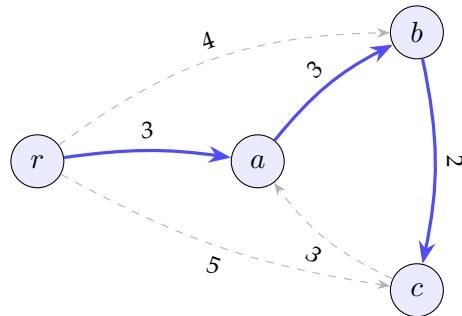


Figura 3 – Os arcos azuis são os da escolha gulosa.

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.

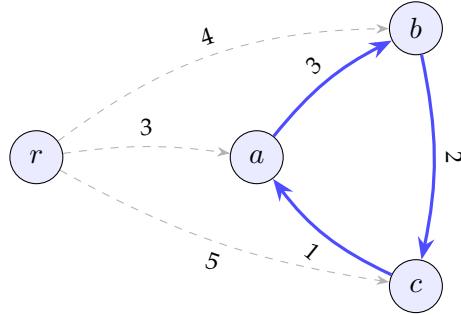


Figura 4 – Os arcos azuis são os da escolha gulosa.

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 ??.

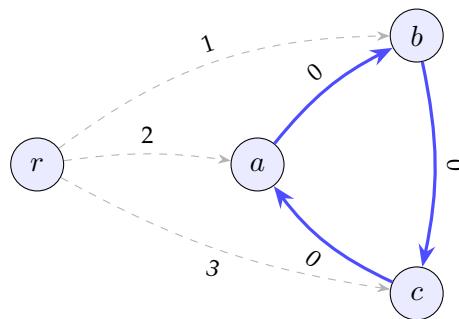


Figura 5 – Os arcos azuis são os da escolha gulosa.

Podemos agora retomar o caso no qual o subdigrafo gerador H de D , cujo arcos são aqueles cujo 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.

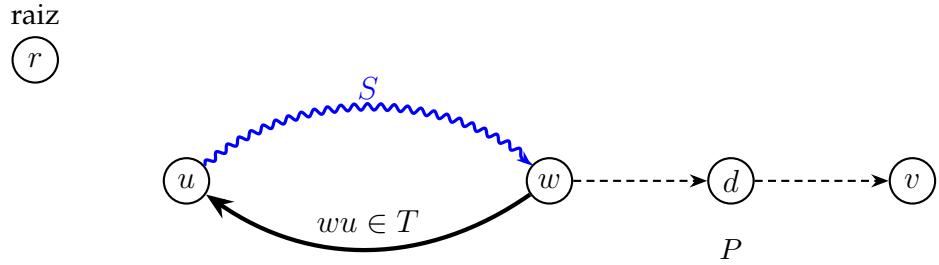


Figura 6 – O caminho simples maximal P inicia em u e termina em v . A porção S de P entre u e w é indicada pelo arco ondulado azul; o caminho $S \cdot u$ é um ciclo.

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 grafo menor. Ao final, *expandimos* as contrações removendo exatamente um arco interno de cada ciclo para manter grau de entrada 1 e aciclicidade global.

1.1.2 Contração de ciclos

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

Observamos que a contração pode gerar arcos paralelos entre um mesmo par de vértices, mas descartamos laços em x_C . Os custos dos arcos que entram no supervértice x_C devem ser ajustados de modo a preservar a otimalidade, conforme estabelecido na proposição a seguir.

Proposição 1.2. *Seja C um ciclo dirigido em D com $r \notin C$, e seja D' o dígrafo 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*

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

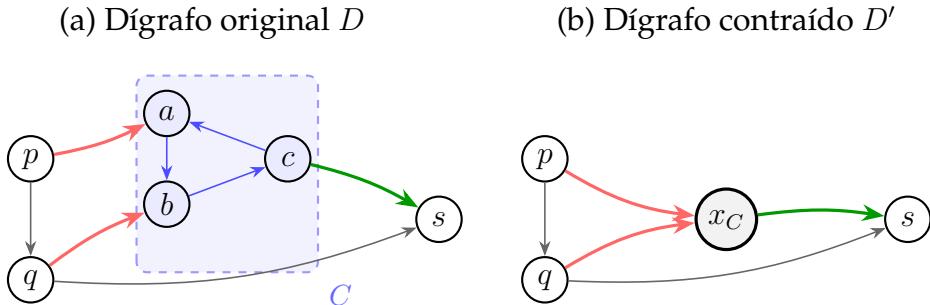


Figura 7 – Ilustração da contração de um ciclo $C = \{a, b, c\}$. No dígrafo original (a), o ciclo C está destacado em azul claro; arcos internos ao ciclo (em azul) são descartados na contração; arcos que entram em C (em vermelho) e que saem de C (em verde) são redirecionados. No dígrafo contraído (b), o ciclo é substituído pelo supervértice x_C , e os arcos externos são ajustados apropriadamente.

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 .

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 dígrafo 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 grafo contraído D' com custos ajustados c' e, em seguida, expandir a solução para o grafo original D .

1.2 Descrição do algoritmo

A seguir apresentamos uma descrição do algoritmo de Chu–Liu/Edmonds. Detalhes de implementação serão discutidos na próxima seção. Denotamos por A' o conjunto de arcos escolhidos na construção da r-arborescência.

Construa A' escolhendo, para cada $v \neq r$, um arco de menor custo que entra em v . Se (V, A') é acíclico, então A' já é uma r-arborescência ótima, pois realizamos o menor custo de entrada em cada vértice e nenhuma troca pode reduzir o custo mantendo as restrições (??, Sec. 4.9).

Se A' contiver um ciclo dirigido C (que não inclui r), normalizamos os custos de entrada, contraímos C em um supervértice x_C ajustando arcos que entram em C por $c'(u, x_C) = c(u, w) - c(a_w)$, e resolvemos recursivamente no grafo contraído.

As arborescências obtidas no grafo contraído estão em correspondência bijetiva com as arborescências do grafo original que possuem exatamente um arco entrando no ciclo C . Como, após a normalização, todos os arcos internos de C passam a ter custo reduzido igual a zero, a expansão do supervértice recupera uma arborescência válida no grafo original sem alterar o custo total da solução.

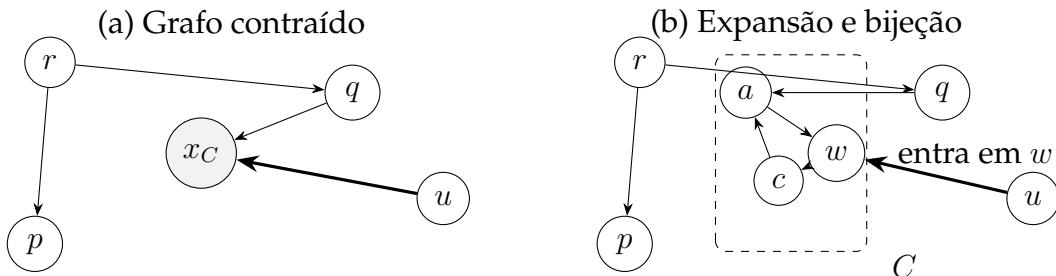


Figura 8 – Bijeção entre arborescências no grafo contraído e no original: toda arborescência em D' escolhe exatamente um arco que entra em x_C ; ao expandir C , esse arco corresponde a um (u, w) que entra em algum $w \in C$ e os arcos internos (de custo reduzido zero) são mantidos, preservando o custo total.

Na expansão, reintroduzimos C e removemos exatamente um arco interno para manter grau de entrada 1 e aciclicidade global (????).

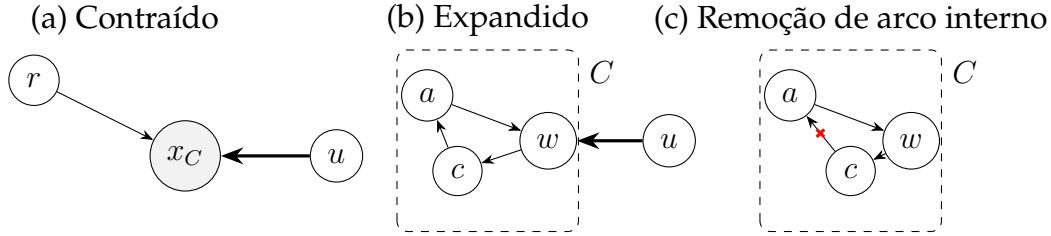


Figura 9 – Reexpansão de C : no grafo contraído seleciona-se um arco que entra em x_C ; ao expandir, x_C é substituído por C e o arco selecionado entra em algum $w \in C$; remove-se exatamente um arco interno de C para eliminar o ciclo, preservando conectividade e custo total (arcos internos têm custo reduzido zero).

Abaixo, temos a descrição formal do algoritmo.

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

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 $F^* := \{a_v : v \neq r\}$.
2. Se (V, F^*) é acíclico, devolva F^* . Por (??, Obs. 4.36), trata-se de uma r-arborescência de custo mínimo.
3. Caso contrário, seja C um ciclo dirigido de F^* (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 grafo original, ele corresponde a (u, w) com $w \in C$. Forme

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

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 (????, 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 $\sum c'$ (??, Obs. 4.37). Em particular, os arcos a_v de menor custo que entram em v têm custo reduzido zero e formam F^* .

2. *Caso acíclico*: se (V, F^*) é acíclico, então já é uma r-arborescência e, por realizar o mínimo custo de entrada em cada $v \neq r$, é ótima (??, Obs. 4.36).
3. *Caso com ciclo (contração/expansão)*: se F^* 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 grafo 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 .

Como 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 (????, 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 optimalidade.

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 apresenta uma implementação em Python do algoritmo. A arquitetura segue os passos teóricos: recebe como entrada um digrafo ponderado, os custos das arestas e o vértice raiz. O procedimento seleciona, para cada vértice, o arco de menor custo de entrada, verifica se o grafo é acíclico e, se necessário, contrai ciclos e ajusta custos. Ao final, retorna como saída a r-arborescência ótima: um conjunto de arestas que conecta todos os vértices à raiz com custo mínimo.

Formalizamos a implementação do algoritmo de Chu–Liu/Edmonds:

- **Entrada:** digrafo $D = (V, A)$, custos $c : A \rightarrow \mathbb{R}$ no atributo "w", raiz $r \in V$.
- **Hipóteses:** (i) D é conexo a partir de r — todo $v \neq r$ é alcançável; (ii) para todo $X \subseteq V \setminus \{r\}$, existe arco entrando em X (condições de Edmonds (??)); (iii) custos não negativos.
- **Saída:** subgrafo T com $|A_T| = |V| - 1$ arestas, grau de entrada 1 para todo $v \neq r$, alcançabilidade a partir de r e custo $\sum_{a \in A_T} c(a)$ mínimo.
- **Convenções:** arcos paralelos permitidos após contrações; laços descartados.

A implementação consiste em funções auxiliares que traduzem cada passo teórico (normalização, construção de F^* , contração, reexpansão) em operações sobre `nx.DiGraph`, orquestradas por uma função principal que gerencia o fluxo recursivo. As subseções seguintes detalham cada função auxiliar: lógica, parâmetros, retornos 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, arestas e atributos, garantindo operações eficientes: adicionar/remover aresta $O(1)$ amortizado, iterar vizinhos $O(\deg(u))$, percorrer todas as arestas $O(m)$.

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

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()`: retorna visão iterável sobre V , permitindo percorrer todos os vértices.
- `D.in_edges(v, data="w")`: retorna arestas entrantes em v com pesos, produzindo tuplas (u, v, w) . Usado para encontrar o arco de menor custo que entra em cada vértice.
- `D.out_edges(u, data="w")`: retorna arestas saíntes de u com pesos, análogo a `in_edges`. Necessário na reexpansão para reintegrar arestas do ciclo.
- `D[u][v]["w"]`: acessa diretamente o peso da aresta (u, v) para leitura ou modificação. Usado na normalização para ajustar custos reduzidos.

Modificação de estrutura

- `D.add_edge(u, v, w=peso)`: adiciona aresta (u, v) com peso especificado, criando vértices automaticamente se não existirem. Usado para construir F^* e reintegrar arestas na reexpansão.
- `D.remove_edges_from(edges)`: remove múltiplas arestas em lote. Aplicado para eliminar arestas internas do ciclo contraído.
- `D.remove_nodes_from(nodes)`: remove vértices e todas as suas arestas incidentes. Usado para eliminar os vértices do ciclo original após contração.

Detecção de ciclos

- `nx.find_cycle(G, orientation="original")`: detecta ciclos via DFS (detalhado na próxima subseção).

Essa função retorna um iterador sobre as arestas do ciclo (tuplas (u, v, key)). Dois aspectos importantes, o retorno é iterador (não lista), economizando memória e em grafos acíclicos, lança a exceção `NetworkXNoCycle` em vez de retornar valor sentinela. Isso segue o princípio EAFP (*Easier to Ask for Forgiveness than Permission*) do Python e casa naturalmente com o fluxo do algoritmo: tratamos o caso acíclico com `try-except`, distinguindo caso base (sem ciclo, retornar solução) de caso recursivo (com ciclo, contrair e recursão).

Abaixo, detalhamos as funções auxiliares que implementam os passos do algoritmo e ao final apresentamos a função principal que orquestra o fluxo recursivo.

1.3.2 Normalização por vértice

Esta função implementa a normalização de custos reduzidos: calcula $y(v) = \min\{w(u, v)\}$ e substitui cada peso $w(u, v)$ por $w(u, v) - y(v)$, garantindo que ao menos uma aresta de entrada tenha custo zero. Como cada r-arborescência possui exatamente uma aresta 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 rótulo node do vértice a ser normalizado. A implementação coleta todas as arestas de entrada de node com seus pesos usando o método `D.in_edges(node, data="w")`, que retorna uma lista de tuplas $(u, node, w)$ (linha 2). Em seguida, verifica se a lista está vazia e se estiver retorna imediatamente sem fazer alterações (linhas 3–4). Caso contrário, calcula o peso mínimo yv através de uma compreensão de gerador que extrai o terceiro elemento de cada tupla (linha 5) e, para cada predecessor u, subtrai yv do peso armazenado em `D[u][node]["w"]` (linha 6).

Não retorna nenhum valor (retorno implícito `None`), pois a operação é realizada in-place: o grafo D passado como parâmetro é modificado diretamente, e ao menos uma aresta de entrada de node terá custo reduzido zero após a execução. A complexidade é $O(\deg^-(v))$, pois cada operação percorre as arestas de entrada uma única vez.

Normalização por vértice: custos reduzidos

Normaliza os pesos das arestas que entram em node, subtraindo de cada uma o menor peso de entrada. Modifica o grafo D in-place.

```

1 def normalize_incoming_edge_weights(D: nx.DiGraph, node: str):
2     predecessors = list(D.in_edges(node, data="w"))
3     if not predecessors:
4         return
5     yv = min((w for _, _, w in predecessors))
6     D[u][node]["w"] -= yv

```

A Figura ?? ilustra o funcionamento da normalização:

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

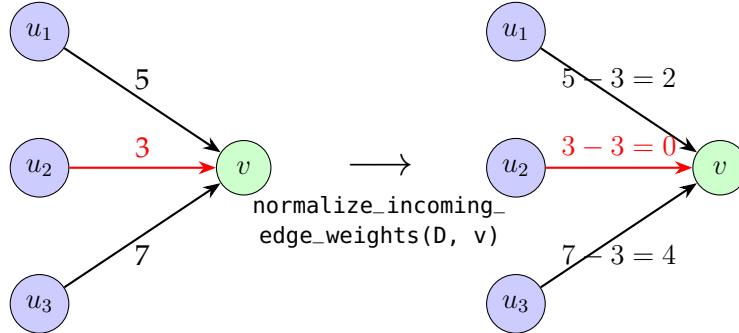


Figura 10 – Exemplo de normalização de custos reduzidos. À esquerda, vértice v com três arestas de entrada (pesos 5, 3 e 7). À direita, após aplicar `normalize_incoming_edge_weights(D, v)`: o menor peso $y(v) = 3$ é subtraído de todas as entradas, resultando em custos reduzidos 2, 0 e 4. A aresta (u_2, v) (em vermelho) tem custo zero e será selecionada para F^* .

Observe que as diferenças relativas são preservadas: a aresta mais cara permanece 4 unidades acima da mais barata, e a intermediária mantém sua posição relativa. Como cada r-arborescência contém exatamente uma aresta 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.

1.3.3 Construção de F^* :

Esta função constrói o subdigrafo F^* selecionando, para cada vértice $v \neq r_0$, uma única aresta de custo reduzido zero que entra em v .

Recebe como entrada um digrafo D (objeto `nx.DiGraph`) e o rótulo $r0$ da raiz. A implementação cria um novo digrafo vazio `F_star` (linha 2) em vez de modificar D diretamente; essa escolha de criar uma estrutura separada é fundamental porque F^* é um subgrafo conceitual usado para detecção de ciclos, e preservar D inalterado permite que as operações subsequentes (como contração) trabalhem com o grafo original completo, evitando perda de informação sobre arestas não selecionadas que podem ser necessárias após reexpansões.

Em seguida, para cada vértice v diferente de $r0$ (linhas 3–4), utilizando o método `D.nodes()`, coleta todas as arestas de entrada de v com seus pesos em uma lista e armazena na variável `in_edges` (linha 5); a materialização em lista é necessária porque a subsequente iteração sobre as arestas para encontrar aquela de peso zero poderia causar problemas se trabalhássemos diretamente com a visão retornada por `in_edges`, especialmente em cenários de modificação concorrente. Se não houver arestas de entrada, prossegue para o próximo vértice (linhas 6–7) usando `continue`, pois um vértice isolado

ou inacessível não contribui para F^* e sua ausência será detectada posteriormente como violação das hipóteses de conectividade.

Caso contrário, utiliza uma compreensão de gerador combinada com `next` para encontrar o primeiro predecessor u cuja aresta (u, v) tem peso zero (linha 8); a escolha de `next` com gerador em vez de uma busca exaustiva é eficiente porque interrompe a iteração assim que encontra a primeira aresta de custo zero, evitando processamento desnecessário das arestas restantes (embora teoricamente todas as arestas de custo zero sejam equivalentes, na prática apenas uma é necessária para F^*). A função `next` retorna `None` se nenhuma aresta de peso zero existir, o que teoricamente não deveria ocorrer após a normalização correta (que garante ao menos uma aresta de custo zero por vértice), mas o tratamento defensivo evita erros em casos degenerados. Se tal aresta existir, adiciona-a a `F_star` com peso zero usando o método `add_edge` (linhas 9–10); a especificação explícita de `w=0` garante que F^* contenha apenas arestas de custo reduzido zero, propriedade fundamental para a corretude do algoritmo.

Retorna o digrafo `F_star` contendo exatamente uma aresta entrando em cada $v \neq r_0$, todas com custo reduzido zero. O grafo original `D` não é modificado, preservando o estado para operações futuras. A complexidade é $O(m)$, onde m é o número de arestas, pois cada aresta é considerada no máximo uma vez durante a iteração sobre todos os vértices: para cada um dos $n - 1$ vértices não-raiz, examina-se suas arestas de entrada (totalizando no máximo m arestas ao longo de todas as iterações), e para cada vértice a busca por aresta de peso zero é interrompida no primeiro match, resultando em tempo linear no tamanho do grafo.

Construção de F star

Constrói o subdigrafo F^ a partir do digrafo D , selecionando para cada vértice (exceto a raiz r_0) uma aresta de custo reduzido zero que entra nele.*

```

1 def get_Fstar(D: nx.DiGraph, r0: str):
2     F_star = nx.DiGraph()
3     for v in D.nodes():
4         if v != r0:
5             in_edges = list(D.in_edges(v, data="w"))
6             if not in_edges:
7                 continue
8             u = next((u for u, _, w in in_edges if w == 0), None)
9             if u:
10                F_star.add_edge(u, v, w=0)
11
return F_star

```

A Figura ?? ilustra a construção de F^* :

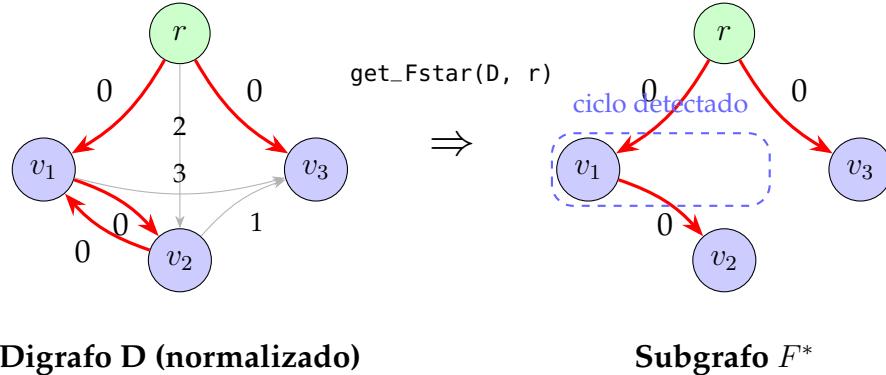


Figura 11 – Exemplo de construção de F^* a partir de um digrafo normalizado. À esquerda, o digrafo D após normalização, onde cada vértice não-raiz possui ao menos uma aresta de entrada com custo zero (em vermelho). À direita, o subgrafo F^* resultante contém apenas as arestas de custo zero selecionadas, uma por vértice. Note que F^* pode conter ciclos (como $\{v_1, v_2\}$) que serão tratados nas etapas subsequentes.

A detecção de ciclos é crucial, pois a presença de um ciclo em F^* indica que a escolha de arestas de custo reduzido zero não formou uma arborescência válida. Esses ciclos precisam ser tratados nas etapas subsequentes do algoritmo.

As funções de normalização por vértice e construção de F^* 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 $F^* := \{a_v : v \neq r\}$.

1.3.4 Detecção de ciclo:

Esta função detecta a presença de um ciclo dirigido em F^* e retorna um subgrafo que o contém; se F^* for acíclico, retorna None.

Recebe como entrada um digrafo `F_star` (objeto `nx.DiGraph`). A implementação utiliza um bloco `try` (linha 2) para capturar exceções caso não haja ciclo; esta escolha de tratamento por exceção é necessária porque a API do NetworkX adota o padrão EAFP (*Easier to Ask for Forgiveness than Permission*), onde `nx.find_cycle` não retorna um valor especial (como `None`) quando o grafo é acíclico, mas sim lança a exceção `NetworkXNoCycle` para sinalizar a ausência de ciclos.

Em seguida a função inicializa um conjunto vazio `nodes_in_cycle` (linha 3) e emprega a função `nx.find_cycle` do NetworkX (linha 4), que realiza uma busca

em profundidade (DFS) para detectar ciclos. A função `nx.find_cycle` percorre o grafo visitando vértices e arestas: ao encontrar uma aresta (u, v) onde v já está na pilha de recursão da DFS, identifica um ciclo e retorna um iterador sobre todas as arestas que compõem esse ciclo. O laço na linha 4 itera sobre essas arestas retornadas, desempacotando cada uma na forma $(u, v, _)$ (ignorando o terceiro elemento com $_$, que contém metadados de orientação), e para cada aresta (u, v) adiciona ambos os vértices ao conjunto `nodes_in_cycle` (linha 5); a escolha de usar conjunto em vez de lista garante que cada vértice seja adicionado apenas uma vez mesmo que o ciclo tenha múltiplas arestas incidentes, e a operação de adição tem complexidade $O(1)$ amortizada.

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

Se nenhum ciclo existir, a exceção `nx.NetworkXNoCycle` é capturada no bloco `except` (linha 8) e a função retorna `None` (linha 9);

No final, um subgrafo contendo os vértices e arestas do ciclo detectado é retornado, ou `None` se não houver ciclo. O grafo original `F_star` não é modificado. A complexidade é $O(m)$, onde m é o número de arestas, pois a DFS visita cada aresta no máximo uma vez.

Detecção de ciclo dirigido em F^*

Detecta um ciclo dirigido em F^ e retorna um subgrafo contendo seus vértices e arestas, ou `None` se for acíclico.*

```

1 def find_cycle(F_star: nx.DiGraph):
2     try:
3         nodes_in_cycle = set()
4         for u, v, _ in nx.find_cycle(F_star, orientation="original"):
5             nodes_in_cycle.update([u, v])
6         return F_star.subgraph(nodes_in_cycle).copy()
7     except nx.NetworkXNoCycle:
8         return None

```

A Figura ?? ilustra o processo de detecção de ciclo:

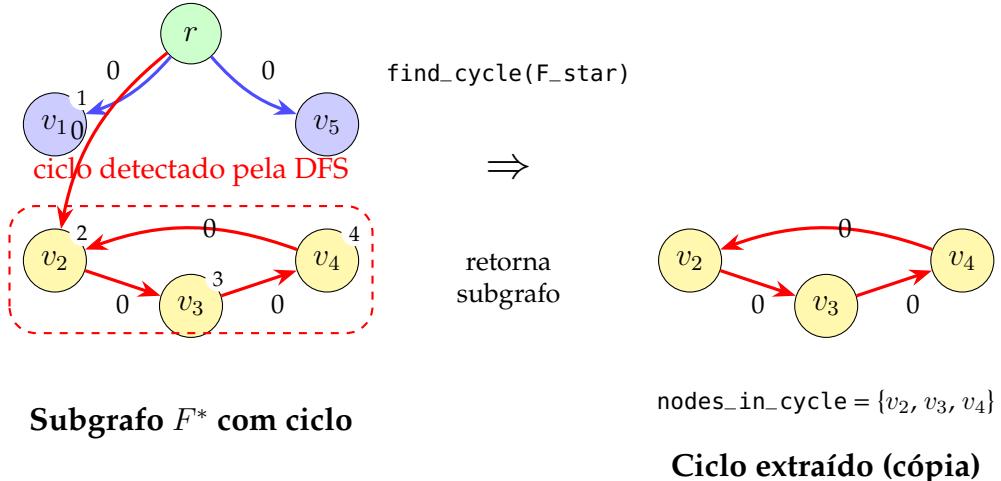


Figura 12 – Exemplo de detecção de ciclo em F^* . À esquerda, o subgrafo F^* contém um ciclo formado pelos vértices $\{v_2, v_3, v_4\}$ (destacados em amarelo). A DFS percorre o grafo e detecta o ciclo ao encontrar a aresta (v_4, v_2) , onde v_2 já está na pilha de recursão. À direita, a função retorna uma cópia do subgrafo induzido pelos vértices do ciclo, contendo apenas os três vértices e as três arestas que formam o ciclo.

Ao detectar um ciclo, a função permite que o algoritmo de Chu–Liu/Edmonds prossiga para a etapa de contração, onde o ciclo será reduzido a um supervértice, facilitando a resolução do problema no grafo modificado.

1.3.5 Contração de ciclo:

Esta função contrai um ciclo dirigido C em um supervértice x_C , redirecionando arcos incidentes e ajustando custos segundo a regra de custos reduzidos. Retorna dicionários auxiliares para reexpansão.

Recebe como entrada um digrafo D (objeto `nx.DiGraph`), o ciclo C a ser contraído e o rótulo `label` do novo supervértice. A implementação coleta 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 `in_to_cycle` (linha 3), um dicionário que tem como chave vértices externos ao ciclo e como valor tuplas (v, w) , onde v é o vértice do ciclo conectado a u e w é o peso da aresta (u, v) ; essa estrutura preserva não apenas o peso mínimo, mas também o ponto exato de entrada no ciclo, informação crucial para a reexpansão posterior.

Para cada vértice u no digrafo D (linha 4), se u não pertence ao ciclo (linha 5), identifica a aresta de menor peso que sai de u e entra em C (linhas 6–9) usando uma compreensão de gerador: a expressão `((v, w) for _, v, w in D.out_edges(u, data="w") if v in cycle_nodes)` itera sobre todas as arestas que saem de u , desempacota cada aresta na forma `(_, v, w)` (ignorando a origem com `_`, capturando o destino

v e o peso w), 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 7) para comparar pelo segundo elemento (o peso), e retorna `None` se não houver arestas (linha 8). A escolha de selecionar apenas a aresta de *menor peso* reflete a propriedade fundamental do algoritmo: qualquer solução ótima que conecta um vértice externo ao ciclo contraído usará necessariamente a aresta de custo mínimo, pois todas as outras seriam subótimas. Se tal aresta existir, armazena em `in_to_cycle` (linhas 9–10).

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 (linhas 11–12). Para cada par, cria uma aresta de u para `label` com peso w , efetivamente redirecionando as arestas de entrada para o supervértice. A separação entre coleta (linhas 4–10) e criação (linhas 11–12) é necessária porque modificar o grafo durante a iteração sobre seus vértices causaria comportamento indefinido; ao coletar primeiros todos os dados em estruturas auxiliares, garantimos que as modificações posteriores sejam seguras.

De forma análoga, constrói o dicionário `out_from_cycle` (linha 13) para mapear arestas que saem do ciclo. Para cada vértice v em D (linha 14), se v não pertence ao ciclo (linha 15), identifica a aresta de menor peso que sai de C e entra em v (linhas 16–17) usando uma compreensão de gerador análoga: a expressão `((u, w) for u, _, w in D.in_edges(v, data="w") if u in cycle_nodes)` itera sobre todas as arestas que entram em v , desempacota cada aresta na forma $(u, _, w)$ (capturando a origem u , ignorando o destino com $_$, e capturando o peso w), filtra apenas aquelas cuja origem u pertence ao ciclo, e produz tuplas (u, w) ; a função `min` seleciona a de menor peso pela mesma razão de otimalidade. Se existir, armazena em `out_from_cycle` (linhas 18–19). Depois, itera sobre `out_from_cycle` e cria arestas de `label` para cada vértice v com os respectivos pesos (linhas 20–21). Por fim, remove todos os vértices de C do grafo (linha 22); essa remoção é realizada por último para garantir que todas as operações de consulta (linhas 4–21) tenham acesso aos dados originais antes da modificação estrutural.

Retorna dois dicionários: `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 aresta interna do ciclo deve ser removida para restaurar a propriedade de arborescência. O digrafo D é modificado *in-place*: os vértices de C são removidos e substituídos por `label`. A escolha de modificação *in-place* (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 ao gerenciamento de referências. A complexidade é $O(m)$, onde m é o número de arestas, pois cada aresta incidente ao ciclo é processada uma vez: os laços

nas linhas 4–10 e 14–19 examinam cada aresta no máximo uma vez, e as operações de adição (linhas 11–12, 20–21) e remoção (linha 22) têm custo proporcional ao número de arestas afetadas.

Contração de ciclo

Contraí o ciclo C em um supervértice label, redirecionando arcos incidentes e ajustando custos. Modifica D in-place e retorna 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, w) for _, v, w in D.out_edges(u, data="w") if v in
8                     cycle_nodes),
9                     key=lambda x: x[1],
10                     default=None,)
11         if min_weight_edge_to_cycle:
12             in_to_cycle[u] = min_weight_edge_to_cycle
13         for u, (v, w) in in_to_cycle.items():
14             D.add_edge(u, label, w=w)
15         out_from_cycle: dict[str, tuple[str, float]] = {}
16         for v in D.nodes:
17             if v not in cycle_nodes:
18                 min_weight_edge_from_cycle = min(
19                     ((u, w) for u, _, w in D.in_edges(v, data="w") if u in
20                     cycle_nodes), key=lambda x: x[1], default=None)
21             if min_weight_edge_from_cycle:
22                 out_from_cycle[v] = min_weight_edge_from_cycle
23             for v, (u, w) in out_from_cycle.items():
24                 D.add_edge(label, v, w=w)
25             D.remove_nodes_from(cycle_nodes)
26     return in_to_cycle, out_from_cycle

```

A Figura ?? ilustra o processo de contração de ciclo:

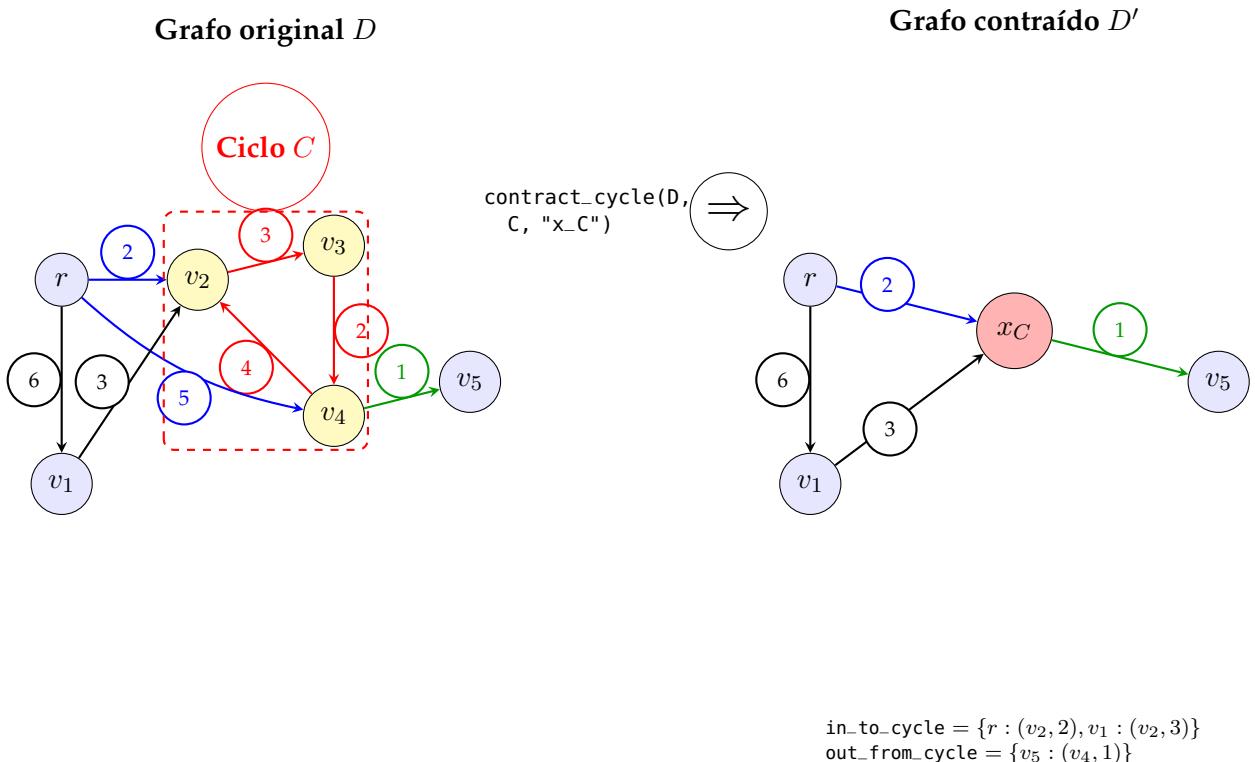


Figura 13 – Exemplo de contração de ciclo. À esquerda, grafo original D com ciclo $C = \{v_2, v_3, v_4\}$ (em amarelo). Vértices externos r , v_1 e v_5 têm arestas conectando ao ciclo: r envia aresta para v_2 (peso 2) e v_4 (peso 5); v_4 envia aresta para v_5 (peso 1). À direita, após a contração: o ciclo é substituído pelo supervértice x_C (vermelho). As arestas de entrada são redirecionadas: (r, x_C) recebe peso 2 (menor entre 2 e 5). A aresta 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, F^*) é acíclico, devolva F^* . Por (??, Obs. 4.36), trata-se de uma r -arborescência de custo mínimo. Passo 3: Caso contrário, seja C um ciclo dirigido de F^* (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 $D' = (V', A')$.

1.3.6 Remoção de arestas que entram na raiz:

Esta função remove todas as arestas que entram no vértice raiz r_0 , garantindo que a raiz não tenha predecessores. 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 grafo original contiver arestas entrando em r_0 , essas arestas 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 arestas 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 arestas. Portanto, esta função atua como um passo de pré-processamento essencial que prepara o grafo para os passos subsequentes do algoritmo.

A escolha de implementar esta operação como uma função auxiliar separada (em vez de incluí-la apenas inline na função 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 descritivo (`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 todas as arestas 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 rótulo r_0 da raiz. A implementação armazena em uma lista todas as arestas que entram em r_0 usando o método `in.edges` (linha 2). Se a lista não estiver vazia (linha 3), remove todas essas arestas usando o método `remove.edges.from` (linha 4). Este método da biblioteca NetworkX recebe como parâmetro uma lista de tuplas representando arestas na forma

(u, v) e remove cada uma delas do grafo. A operação é realizada em lote: NetworkX itera sobre a lista fornecida e , para cada tupla (u, v) , remove a aresta correspondente da estrutura interna de adjacência. Se alguma aresta especificada não existir no grafo, ela é silenciosamente ignorada sem gerar erro. A complexidade de `remove_edges_from` é $O(k)$, onde k é o número de arestas na lista de entrada, pois cada remoção individual tem custo $O(1)$ em média devido ao uso de dicionários aninhados para armazenar arestas.

Por fim, a função retorna o grafo D atualizado in-place com todas as arestas de entrada em $r\theta$ são removidas (linha 5). A complexidade total da função é $O(\deg^-(r_0))$, pois a operação coleta e remove cada aresta de entrada uma única vez.

Remoção de arestas que entram na raiz

Remove todas as arestas que entram na raiz $r\theta$, modificando D in-place e retornando o grafo atualizado.

```

1 def remove_edges_to_rθ(D: nx.DiGraph, rθ: str):
2     in_edges = list(D.in_edges(rθ))
3     if in_edges:
4         D.remove_edges_from(in_edges)
5     return D

```

1.3.7 Remoção de arco interno:

Esta função é invocada durante a fase de reexpansão do ciclo contraído, após a chamada recursiva retornar com a arborescência ótima T' do grafo contraído. Quando o supervértice x_C é expandido de volta para o ciclo original C , uma aresta externa (u, v) é adicionada conectando um vértice externo u a um vértice v dentro do ciclo. Como o ciclo C originalmente continha exatamente uma aresta entrando em cada um de seus vértices (formando um ciclo fechado), e agora v recebe uma aresta adicional vinda 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 a aresta interna que anteriormente entrava em v , mantendo apenas a nova aresta externa. 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 subgrafo do ciclo C (objeto `nx.DiGraph`) 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 as arestas de entrada de v , extraíndo apenas o

vértice origem u (ignorando metadados com $_$), e `next` retorna o primeiro (e teoricamente único) predecessor. Em seguida, remove a aresta (`predecessor, v`) do ciclo usando o método `remove_edge` (linha 3).

A função modifica o subgrafo C in-place e não retorna valor. A complexidade é $O(\deg^-(v))$, dominada pela operação de busca das arestas de entrada, embora em ciclos simples isso seja tipicamente $O(1)$ pois cada vértice tem exatamente um predecessor.

Remover arco interno na reexpansão

Remove a aresta interna que entra no vértice de entrada v do ciclo C durante a reexpansão, pois v passa a receber uma aresta externa, e manter ambas violaria a propriedade de arborescência.

```
1 def remove_internal_edge_to_cycle_entry(C: nx.DiGraph, v):
2     predecessor = next((u for u, _ in C.in_edges(v)), None)
3     C.remove_edge(predecessor, v)
```

A Figura ?? ilustra o objetivo da função:

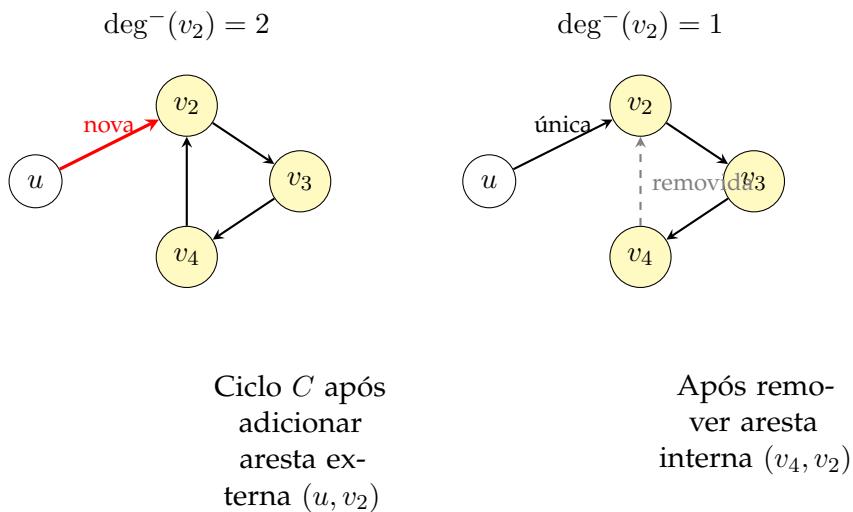


Figura 14 – Remoção de aresta interna durante reexpansão. À esquerda, ciclo $C = \{v_2, v_3, v_4\}$ após adicionar aresta externa (u, v_2) vindoura da arborescência T' : o vértice v_2 tem grau de entrada 2 (aresta externa vermelha de u e aresta interna do ciclo vindra de v_4), violando a propriedade de arborescência. À direita, após remover a aresta interna (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. A aresta removida é mostrada tracejada em cinza.

1.3.8 Procedimento principal (recursivo):

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

A implementação segue a estrutura do algoritmo:

Preservação do grafo original (linha 2):

Cria uma cópia $D_copy = D.copy()$ para preservar os pesos originais. Como as operações de normalização e contração modificam os pesos das arestas in-place, a cópia é necessária para restaurar os custos corretos na arborescência final. Complexidade: $O(m + n)$.

Normalização e construção de F^* (linhas 3–6):

Itera sobre todos os vértices não-raiz (linhas 3–5), chamando `normalize_incoming_edge_weight(v)` para cada um. Após normalizar todos os vértices, constrói F^* (linha 6) chamando `get_Fstar(D_copy, r0)`, que seleciona uma aresta de custo reduzido zero entrando em cada vértice não-raiz.

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

Verifica se F^* é uma arborescência válida usando `nx.is_arborescence(F_star)` (linha 7). Se sim, restaura os pesos originais de D para cada aresta de F^* (linhas 8–9) e retorna F_{star} como solução (linha 10). A função `nx.is_arborescence` testa conectividade, aciclicidade e grau de entrada correto simultaneamente.

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

Caso F^* contenha um ciclo, detecta C chamando `find_cycle(F_star)` (linha 12). Cria um rótulo único `contracted_label = f"contracted_{level}"` para o supervértice (linha 13). Contraí o ciclo chamando `contract_cycle(D_copy, C, contracted_label)` (linhas 14–15), que modifica D_copy in-place criando o digrafo contraído D' e retorna os dicionários `in_to_cycle` e `out_from_cycle`. Chama-se recursivamente (linha 16) com `find_optimum_arborescence_chuliu(D_copy, r0, level+1)`, obtendo F' .

Reexpansão do ciclo contraído (linhas 17–30):

Identifica a aresta externa que entra no supervértice em F' (linha 17) e extrai o vértice externo u (linha 18). Consulta $v = \text{in_to_cycle}[u]$ para determinar o vértice do ciclo que recebe a conexão (linha 19). Remove a aresta interna que entrava em v chamando `remove_internal_edge_to_cycle_entry(C, v)` (linha 20), quebrando o ciclo no ponto de entrada. Adiciona a aresta externa (u, v) a F' (linha 21) e reintegra as demais arestas do ciclo (linhas 22–23). Processa as arestas de saída (linhas 24–26): para cada $(\text{contracted_label}, w)$ em F' , adiciona (v_{out}, w) usando `out_from_cycle[w]`. Remove o supervértice (linha 27), restaura os pesos originais (linhas 28–29) e retorna `F_prime` (linha 30).

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

O código completo da função principal é apresentado a seguir:

Procedimento principal (recursivo)

Implementa o algoritmo de Chu–Liu/Edmonds de forma recursiva para encontrar a arborescência de custo mínimo em um digrafo ponderado D com raiz r_0 . Normaliza custos, constrói F^ , detecta ciclos e, se houver, contrai em supervértice, resolve recursivamente no grafo reduzido e reexpande, restaurando a arborescência ótima no grafo original. Retorna um `nx.DiGraph` contendo exatamente $|V| - 1$ arestas com grau de entrada 1 para cada vértice exceto a raiz.*

```

1 def find_optimum_arborescence_chuli(D: nx.DiGraph, r0: str, level=0,):
2     D_copy = D.copy()
3     for v in D_copy.nodes:
4         if v != r0:
5             normalize_incoming_edge_weights(D_copy, v)
6     F_star = get_Fstar(D_copy, r0)
7     if nx.is_arborescence(F_star):
8         for u, v in F_star.edges:
9             F_star[u][v]["w"] = D[u][v]["w"]
10    return F_star
11 else:
12     C: nx.DiGraph = find_cycle(F_star)
13     contracted_label = f"\n {level}"
```

```

14     in_to_cycle, out_from_cycle = contract_cycle(
15         D_copy, C, contracted_label)
16     F_prime = find_optimum_arborescence_chuliu(D_copy, r0, level + 1)
17     in_edge = next(iterator(F_prime.in_edges(contracted_label, data="w")),
18                     None)
18     u, _, _ = in_edge
19     v, _ = in_to_cycle[u]
20     remove_internal_edge_to_cycle_entry(C, v)
21     F_prime.add_edge(u, v)
22     for u_c, v_c in C.edges:
23         F_prime.add_edge(u_c, v_c)
24     for _, z, _ in F_prime.out_edges(contracted_label, data=True):
25         u_cycle, _ = out_from_cycle[z]
26         F_prime.add_edge(u_cycle, z)
27     F_prime.remove_node(contracted_label)
28     for u, v in F_prime.edges:
29         F_prime[u][v]["w"] = D[u][v]["w"]
30     return F_prime

```

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

1.3.8.1 Exemplo de execução do algoritmo

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

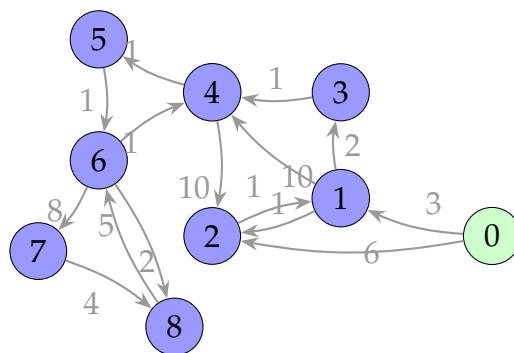


Figura 15 – Grafo direcionado ponderado inicial com raiz no vértice 0. O grafo contém 9 vértices e múltiplas arestas com pesos variados. O primeiro passo do algoritmo seria remover arestas que entram na raiz, porém não há nenhuma neste caso, logo não existe necessidade de alterar o grafo.

O primeiro passo do nosso algoritmo seria remover as arestas que entram na raiz (vértice 0), porém não há nenhuma nesse caso, logo não existe a necessidade de alterar o grafo.

O próximo passo é normalizar os pesos das arestas de entrada para cada vértice. Nessa etapa, para cada vértice v (exceto a raiz), o algoritmo encontra a aresta de menor peso que entra em v e subtrai esse menor peso de todas as arestas que entram em v (isso serve para zerar o peso da aresta mínima de entrada em cada vértice).

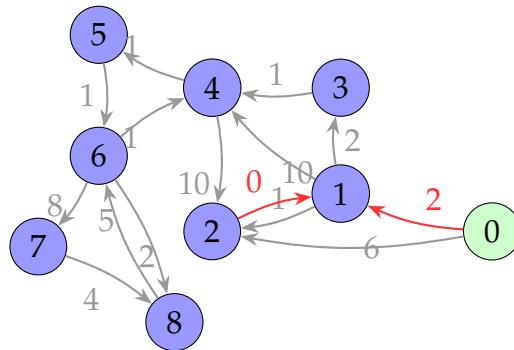


Figura 16 – Normalização parcial das arestas de entrada para o vértice 1. As arestas de entrada são $(0 \rightarrow 1)$ com peso original 3 e $(2 \rightarrow 1)$ com peso original 1. Elegendo a aresta $(2 \rightarrow 1)$ como a de menor peso (peso mínimo = 1), subtraímos este valor de todas as arestas 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 F^* : para isso, selecionamos para cada vértice a aresta de custo reduzido zero de entrada. Detectamos um ciclo em F^* , formado pelos vértices $\{1, 2\}$. Portanto, precisamos contrair esse ciclo em um supervértice $n * 0$.

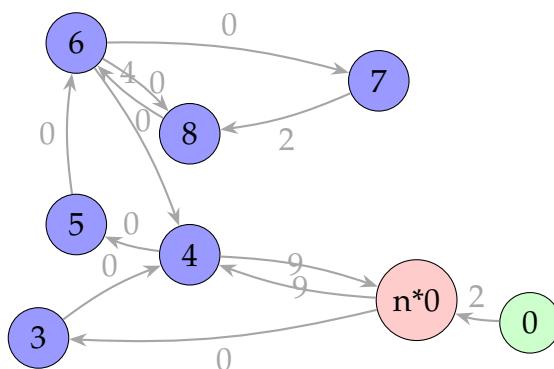


Figura 17 – Grafo contraído após detecção do ciclo $C = \{1, 2\}$ em F^* . O ciclo foi contraído no supervértice $n * 0$ (destacado em vermelho). As arestas que entravam ou saíam do ciclo foram redirecionadas para o supervértice, com custos ajustados segundo as fórmulas $c'(u, x_C) := c(u, w) - y(w)$ para arestas de entrada e $c'(x_C, v) := c(w, v)$ para arestas de saída.

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

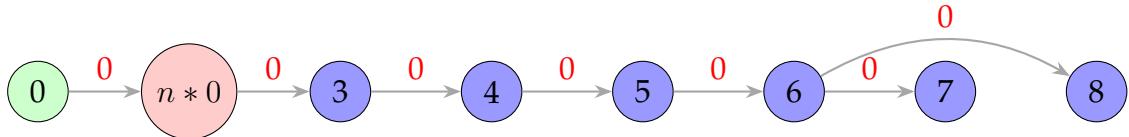


Figura 18 – Arborescência ótima F' obtida no grafo contraído. Todas as arestas selecionadas têm custo reduzido 0 (destacados em vermelho), e o grafo forma uma arborescência válida enraizada em 0: cada vértice (exceto a raiz) tem exatamente uma aresta de entrada, não há ciclos, e todos os vértices são alcançáveis a partir da raiz. Como F' é acíclico, alcançamos o caso base da recursão.

Após validarmos que F^* não possui mais ciclos e forma uma arborescência, iniciamos o processo de reexpansão do ciclo contraído para obter a arborescência final no grafo original. Adicionamos a aresta de entrada ao ciclo $(0, 1)$, as arestas internas do ciclo modificado $(1, 2)$, e as arestas de saída $(1, 3)$, chegando a uma arborescência válida.

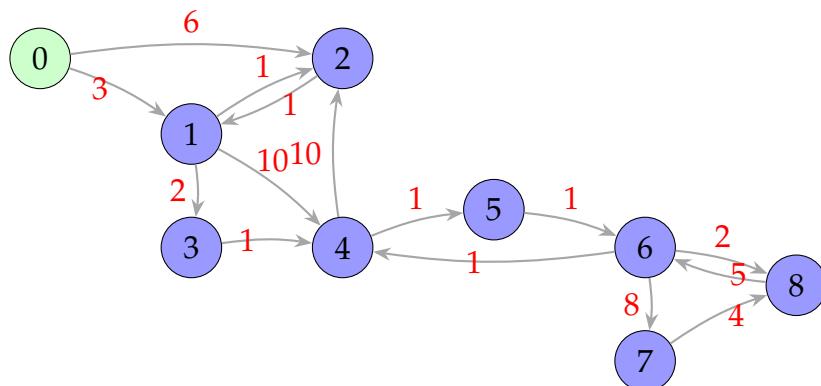


Figura 19 – Arborescência ótima final no grafo original com pesos restaurados. O supervértice $n * 0$ foi expandido de volta para os vértices 1 e 2, com a aresta externa $(0, 1)$ escolhida pela solução recursiva conectando ao ciclo. A aresta interna $(2, 1)$ do ciclo original foi removida para manter a propriedade de arborescência ($\deg^-(v) = 1$). O resultado é uma 0-arborescência de custo mínimo com exatamente 8 arestas, 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 F^* 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 $F^* := \{a_v : v \neq r\}$.</p>	<p>Linhos 3–6: <code>for v in D_copy.nodes: normalize_incoming_edge_weights(D_copy, v) F_star = get_Fstar(D_copy, r0)</code> Calcula $y(v)$ e cria custos reduzidos, depois constrói F^* selecionando arestas de custo zero.</p>
<p>Passo 2: Verificação de aciclicidade (caso base) Se (V, F^*) é acíclico, devolva F^*. Por Obs. 4.36 de (??), trata-se de uma r-arborescência de custo mínimo.</p>	<p>Linhos 7–10: <code>if nx.is_arborescence(F_star): [restaura pesos originais] return F_star</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 F^* (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>Linhos 11–15: <code>C = find_cycle(F_star) 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 16: <code>F_prime = find_optimum_arborescence_chuliu(D_copy, r0, level+1)</code> Chamada recursiva resolve o problema no grafo 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 a aresta interna entrando em v, e reintegre demais arestas de C. Restaure custos originais.</p>	<p>Linhos 17–30: <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 (`normalize_incoming_edge_weights`, `get_Fstar`, `find_cycle`, `contract_cycle`, `remove_internal_edge_to_cycle_entry`) encapsulam exatamente as operações descritas na formulação teórica, preservando as propriedades de correção e complexidade do algoritmo original.

1.3.10 Transição para a abordagem primal-dual

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

No capítulo seguinte, revisitaremos o mesmo problema sob uma ótica primal–dual em duas fases, proposta por András Frank. Essa perspectiva organiza a normalização via 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 das arestas que entram em v , sem mudar a otimalidade global: preserva a ordem relativa entre entradas e torna “apertadas” (custo reduzido zero) as candidatas corretas, habilitando contrações e uma prova de corretude via cortes apertados.

Referências

KLEINBERG, J.; TARDOS, É. *Algorithm Design*. [S.l.]: Addison-Wesley, 2006. Citado 5 vezes nas páginas [17](#), [18](#), [19](#), [31](#) e [40](#).

SCHRIJVER, A. *Combinatorial Optimization: Polyhedra and Efficiency*. [S.l.]: Springer, 2003. Citado 3 vezes nas páginas [18](#), [19](#) e [20](#).

Anexos

ANEXO A – Anexo A

Conteúdo do anexo A.