

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 ao Programa de
Pós-Graduação como requisito parcial para
obtenção do título de Mestre.

Universidade

Faculdade

Programa de Pós-Graduação

Orientador: Prof. Dr. Mário Leston

Brasil

2025

Dedicatória (opcional).

Agradecimientos

Agradecimientos (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

- Figura 1 – Ciclo gerado pelas escolhas locais "mais baratas por vértice". Os arcos grossos (custo 1) entram em a, b, c e formam $a \rightarrow b \rightarrow c \rightarrow a$. Os arcos tracejados partindo de r existem, mas são mais caros e por isso não são escolhidos pelo critério local. 11
- Figura 2 – Bijecção 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. 12
- Figura 3 – 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). 13
- Figura 4 – 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^* 19
- Figura 5 – 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. 21
- Figura 6 – 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. 23

- Figura 7 – 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. 26
- Figura 8 – 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 vinda 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. 29
- Figura 9 – 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. 35
- Figura 10 – 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. 35
- Figura 11 – 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. . . 36

- Figura 12 – 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. 36
- Figura 13 – 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. 37

Sumário

1	ALGORITMO DE CHU-LIU/EDMONDS	11
1.1	Descrição do algoritmo	12
1.1.1	Corretude	14
1.1.2	Complexidade	14
1.2	Implementação em Python	15
1.2.1	Representação de Digrafos: NetworkX	15
1.2.1.1	Estrutura Interna	15
1.2.1.2	Operações Fundamentais	16
1.2.1.3	Detecção de Ciclos via Busca em Profundidade	16
1.2.2	Especificação do Algoritmo	17
1.2.3	Normalização por vértice	18
1.2.4	Construção de F^* :	19
1.2.5	Detecção de ciclo:	21
1.2.6	Contração de ciclo:	23
1.2.7	Remoção de arestas que entram na raiz:	27
1.2.8	Remoção de arco interno:	28
1.2.9	Procedimento principal (recursivo):	30
1.2.9.1	Exemplo de execução do algoritmo	35
1.2.10	Transição para a abordagem primal-dual	37
	REFERÊNCIAS	38
	ANEXOS	39
	ANEXO A – ANEXO A	40

1 Algoritmo de Chu–Liu/Edmonds

O algoritmo de Chu–Liu/Edmonds encontra uma r -arborescência de custo mínimo em um digrafo ponderado. A estratégia funciona de forma gulosa ao escolher, para cada vértice $v \neq r$, o arco de entrada mais barato. No entanto, essa abordagem pode gerar ciclos dirigidos, incompatíveis com a estrutura de arborescência. O algoritmo resolve esse problema combinando normalização de custos, contração de ciclos em supervértices e expansão controlada para garantir otimalidade.

Para entender por que a estratégia gulosa ingênua falha, consideremos as restrições estruturais de uma r -arborescência: cada $v \neq r$ deve ter exatamente um arco de entrada e r tem grau de entrada zero. Se escolhermos para cada vértice o arco mais barato que nele entra, podemos formar um ciclo dirigido C onde todos os vértices recebem seu único arco de dentro do próprio C . Nesse caso, nenhum arco entraria em C a partir de $V \setminus C$ (o corte $\delta^-(C)$ ficaria vazio) e, como $r \notin C$, não existiria caminho de r para os vértices de C , contrariando a alcançabilidade exigida.

A Figura 1 ilustra com um microexemplo: três vértices a, b, c (todos fora de r) onde o arco mais barato que entra em b vem de a , o de c vem de b e o de a vem de c , formando o ciclo $a \rightarrow b \rightarrow c \rightarrow a$. Embora existam arcos de r para cada vértice, eles são mais caros e não são escolhidos pelo critério local, deixando os vértices "presos" no ciclo sem conexão com a raiz.

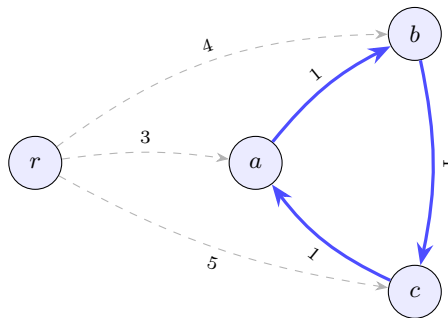


Figura 1 – Ciclo gerado pelas escolhas locais "mais baratas por vértice". Os arcos grossos (custo 1) entram em a, b, c e formam $a \rightarrow b \rightarrow c \rightarrow a$. Os arcos tracejados partindo de r existem, mas são mais caros e por isso não são escolhidos pelo critério local.

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 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 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 (KLEINBERG; TARDOS, 2006, 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 do grafo contraído correspondem, em bijeção, às arborescências do grafo original com exatamente um arco entrando em C . Como os arcos internos de C têm custo reduzido zero, os custos são preservados na ida e na volta.

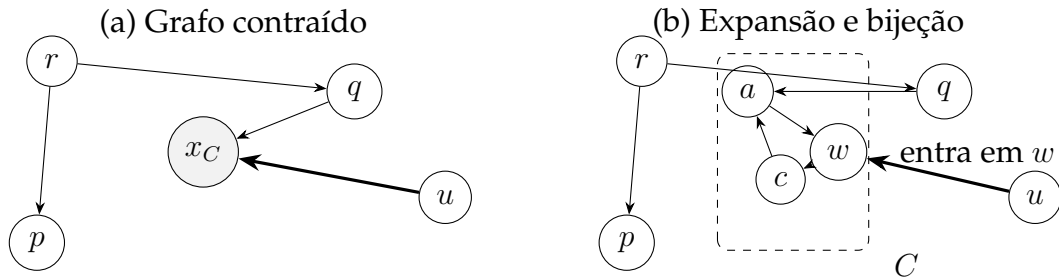


Figura 2 – Bijeção entre arborescências no grafo contraído e no original: toda arborescência em D' escolhe exatamente um arco que entra em x_C ; ao expandir C , esse arco corresponde a um (u, w) que entra em algum $w \in C$ e os arcos internos (de custo reduzido zero) são mantidos, preservando o custo total.

Na expansão, reintroduzimos C e removemos exatamente um arco interno para manter grau de entrada 1 e aciclicidade global (SCHRIJVER, 2003; KLEINBERG; TARDOS, 2006).

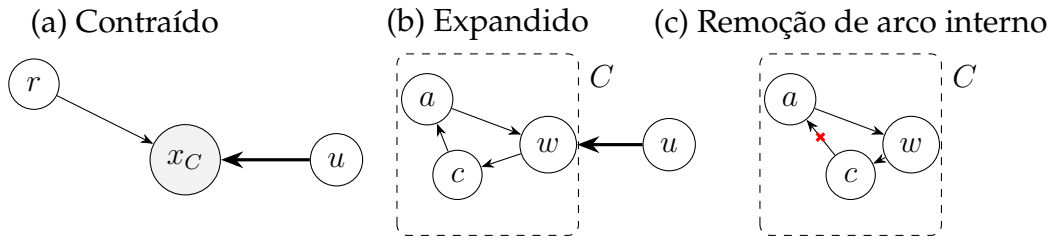


Figura 3 – Reexpansão de C : no grafo contraído seleciona-se um arco que entra em x_C ; ao expandir, x_C é substituído por C e o arco selecionado entra em algum $w \in C$; remove-se exatamente um arco interno de C para eliminar o ciclo, preservando conectividade e custo total (arcos internos têm custo 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 (KLEINBERG; TARDOS, 2006, 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 (KLEINBERG; TARDOS, 2006; SCHRIJVER, 2003, Sec. 4.9).

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

1.1.1 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 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 (KLEINBERG; TARDOS, 2006, 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 (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 otimalidade.

1.1.2 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.

1.2 Implementação em Python

Esta seção apresenta uma implementação em Python do algoritmo de Chu–Liu/Edmonds. Antes de detalhar o algoritmo propriamente dito, estabelecemos a infraestrutura necessária: a biblioteca NetworkX para representação de digrafos, as operações fundamentais sobre essas estruturas e a busca em profundidade para detecção de ciclos. A implementação a seguir adota a versão $O(mn)$ por simplicidade e está disponível no repositório do projeto (<https://github.com/lorenypsum/GraphVisualizer>).

1.2.1 Representação de Digrafos: NetworkX

A implementação utiliza a biblioteca NetworkX¹, que fornece a classe `nx.DiGraph` para representar grafos direcionados. Esta classe constitui a base para todas as operações do algoritmo, oferecendo estruturas de dados eficientes e uma API rica para manipulação de digrafos.

1.2.1.1 Estrutura Interna

A classe `nx.DiGraph` utiliza dicionários aninhados do Python para armazenar o digrafo $D = (V, A)$, combinando simplicidade conceitual com eficiência computacional:

- **Vértices:** um dicionário mapeia cada vértice para seus atributos. O método `D.nodes()` retorna uma visão (`NodeView`) sobre V , permitindo iteração e verificação de pertinência em tempo $O(n)$.
- **Arestas:** estruturas de adjacência bidirecionais mantêm, para cada $u \in V$:
 - *Sucessores:* dicionário com vértices v tais que $(u, v) \in A$, acessível via `D[u]`.
 - *Predecessores:* dicionário com vértices w tais que $(w, u) \in A$, consultado por `D.in_edges(u)`.
- **Atributos de arestas:** cada aresta armazena atributos arbitrários como dicionários. A notação `D[u][v]["w"]` acessa o atributo "w" (peso) da aresta (u, v) .

Esta representação garante operações eficientes: adicionar ou remover uma aresta custa $O(1)$ em média, consultar os vizinhos de u requer $O(\deg(u))$, e iterar sobre todas as arestas leva tempo $O(m)$.

¹ NetworkX é uma biblioteca Python para criação, manipulação e estudo de estruturas, dinâmicas e funções de redes complexas. Disponível em <https://networkx.org/>.

1.2.1.2 Operações Fundamentais

A implementação do algoritmo de Chu–Liu/Edmonds utiliza as seguintes operações da API NetworkX:

- `D.nodes()`: retorna visão sobre V , permitindo iteração e testes de pertinência.
- `D.in_edges(v, data="w")`: retorna as arestas entrantes em v , opcionalmente com atributos. O parâmetro `data="w"` produz tuplas (u, v, w) onde w é o peso da aresta.
- `D.out_edges(u, data="w")`: retorna as arestas saíntes de u , com semântica análoga a `in_edges`.
- `D.add_edge(u, v, **attr)`: adiciona (u, v) com atributos opcionais, criando u e v automaticamente se não existirem.
- `D.remove_edges_from(edges)`: remove múltiplas arestas em lote, recebendo lista de tuplas (u, v) .
- `D.remove_nodes_from(nodes)`: remove múltiplos vértices e todas as suas arestas incidentes.

Essas operações são utilizadas intensivamente nas fases de normalização, contração de ciclos e reexpansão do algoritmo.

1.2.1.3 Detecção de Ciclos via Busca em Profundidade

A detecção de ciclos dirigidos é crucial no algoritmo de Chu–Liu/Edmonds: após selecionar os arcos de custo reduzido zero, precisamos verificar se formam um grafo acíclico (caso base) ou contêm um ciclo que exige contração (caso recursivo). A biblioteca NetworkX implementa essa verificação através da busca em profundidade (Depth-First Search, DFS).

A partir de um vértice s , a DFS marca s como visitado e recursivamente explora cada vizinho não visitado até atingir a máxima profundidade de cada ramo antes de retroceder (backtrack). Durante a travessia, cada vértice passa por três estados:

- **Não visitado:** ainda não explorado pela DFS.
- **Em processamento:** na pilha de recursão atual (ancestral no caminho DFS).
- **Concluído:** totalmente processado (todos os descendentes explorados).

Um ciclo é detectado quando a DFS encontra uma aresta (u, v) onde v está *em processamento*, indicando que existe um caminho de v até u na árvore DFS e (u, v) fecha

um ciclo. A complexidade desta operação é $O(n + m)$, pois cada vértice é visitado exatamente uma vez e cada aresta é examinada no máximo uma vez.

A função `nx.find_cycle(G, orientation="original")` implementa detecção de ciclos via DFS. Aceita como parâmetros o digrafo G e opcionalmente `orientation`, que controla o formato de saída das arestas. A função percorre G executando DFS a partir de cada vértice não visitado e, ao detectar uma aresta de retorno (u, v) onde v está na pilha de recursão, identifica um ciclo.

O retorno é um *iterador* (não uma lista materializada) sobre as arestas do ciclo, onde cada elemento é uma tupla (u, v, key) contendo os vértices da aresta e metadados de orientação (tipicamente ignorados via desempacotamento com `_`). Esta escolha de retornar um iterador reduz o uso de memória.

Um aspecto crucial do design é o tratamento de grafos acíclicos: em vez de retornar `None` ou lista vazia, a função lança a exceção `NetworkXNoCycle`. Esta decisão segue o princípio EAFP (*Easier to Ask for Forgiveness than Permission*) do Python, forçando o código cliente a tratar explicitamente o caso acíclico com `try-except`, o que resulta em código mais robusto e com intenção clara — exatamente o padrão usado no algoritmo de Chu–Liu/Edmonds para distinguir caso base (acíclico) de caso recursivo (com ciclo).

1.2.2 Especificação do Algoritmo

Com a representação estabelecida, especificamos formalmente o algoritmo implementado:

- **Entrada:** digrafo ponderado $D = (V, A)$ (objeto `nx.DiGraph`), custos $c : A \rightarrow \mathbb{R}$ armazenados no atributo `"w"` das arestas, raiz $r \in V$.
- **Hipóteses:**
 - D é conexo a partir de r : (i) todo $v \neq r$ é alcançável a partir de r (caso contrário, não há r -arborescência); (ii) para todo subconjunto não vazio $X \subseteq V \setminus \{r\}$, existe ao menos um arco que entra em X ($\delta^-(X) \neq \emptyset$; condições clássicas de existência à la Edmonds ([SCHRIJVER, 2003](#))).
 - Os custos são não negativos: $c(a) \geq 0$ para todo $a \in A$.
- **Saída:** subgrafo T (objeto `nx.DiGraph`) com $|A_T| = |V| - 1$ arestas, tal que cada $v \neq r$ tem grau de entrada 1, todos os vértices são alcançáveis a partir de r e $\sum_{a \in A_T} c(a)$ é mínimo.
- **Convenções:** arcos paralelos (múltiplos arcos entre o mesmo par de vértices) são permitidos após contrações; laços (self-loops) são descartados.

Criamos funções auxiliares para traduzir cada passo do algoritmo teórico em operações concretas sobre o objeto `nx.DiGraph` e uma função principal chama essas auxiliares na ordem correta, gerenciando contrações e expansões e todo o fluxo descrito formalmente na seção anterior.

A seguir, detalhamos as implementações das funções auxiliares, apresentamos como elas correspondem aos passos do algoritmo teórico, apresentamos exemplos de uso e por fim discutiremos a função principal que orquestra a execução do algoritmo. Cada função é explicada em termos de sua lógica, parâmetros, retornos e complexidade, começando pela normalização dos custos por vértice.

1.2.3 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 y_v através de uma compreensão de gerador que extrai o terceiro elemento de cada tupla (linha 5) e, para cada predecessor u , subtrai y_v 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 4 ilustra o funcionamento da normalização:

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

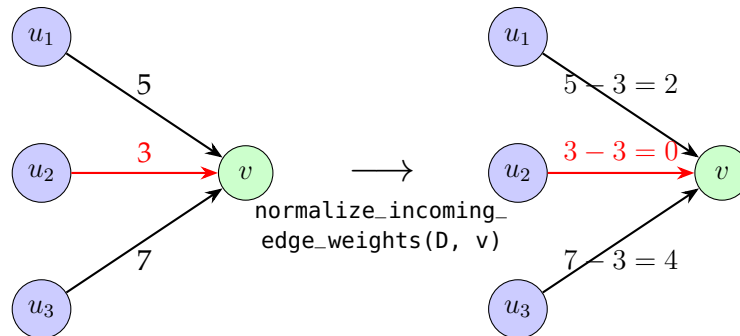


Figura 4 – Exemplo de normalização de custos reduzidos. À esquerda, vértice v com três arestas de entrada (pesos 5, 3 e 7). À direita, após aplicar `normalize_incoming_edge_weights(D, v)`: o menor peso $y(v) = 3$ é subtraído de todas as entradas, resultando em custos reduzidos 2, 0 e 4. A aresta (u_2,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

1.2.4 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.*

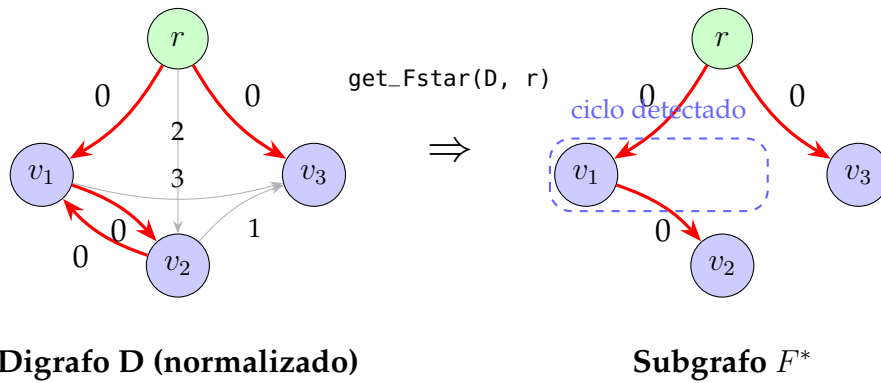
```
1 def get_Fstar(D: nx.DiGraph, r0: str):
2     F_star = nx.DiGraph()
3     for v in D.nodes():
4         if v != r0:
5             in_edges = list(D.in_edges(v, data="w"))
6             if not in_edges:
```

```

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 5 ilustra a construção de F^* :



Digrafo D (normalizado)

Subgrafo F^*

Figura 5 – 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 $F^* := \{a_v : v \neq r\}$.

1.2.5 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 (ver Seção 1.2.1.3). 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 6 ilustra o processo de detecção de ciclo:

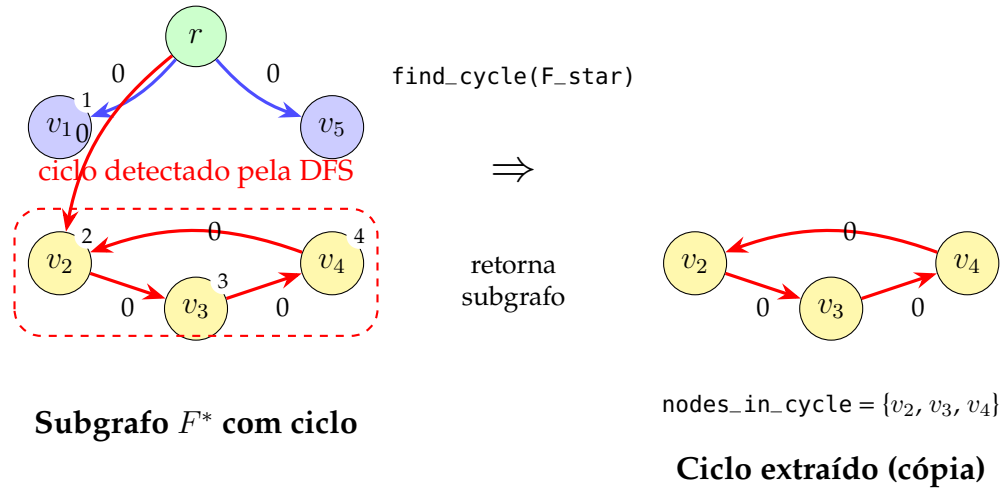


Figura 6 – Exemplo de detecção de ciclo em F^* . À esquerda, o subgrafo F^* contém um ciclo formado pelos vértices $\{v_2, v_3, v_4\}$ (destacados em amarelo). A DFS percorre o grafo e detecta o ciclo ao encontrar a aresta (v_4, v_2) , onde v_2 já está na pilha de recursão. À direita, a função retorna 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.

1.2.6 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 primeiro 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 7 ilustra o processo de contração de ciclo:

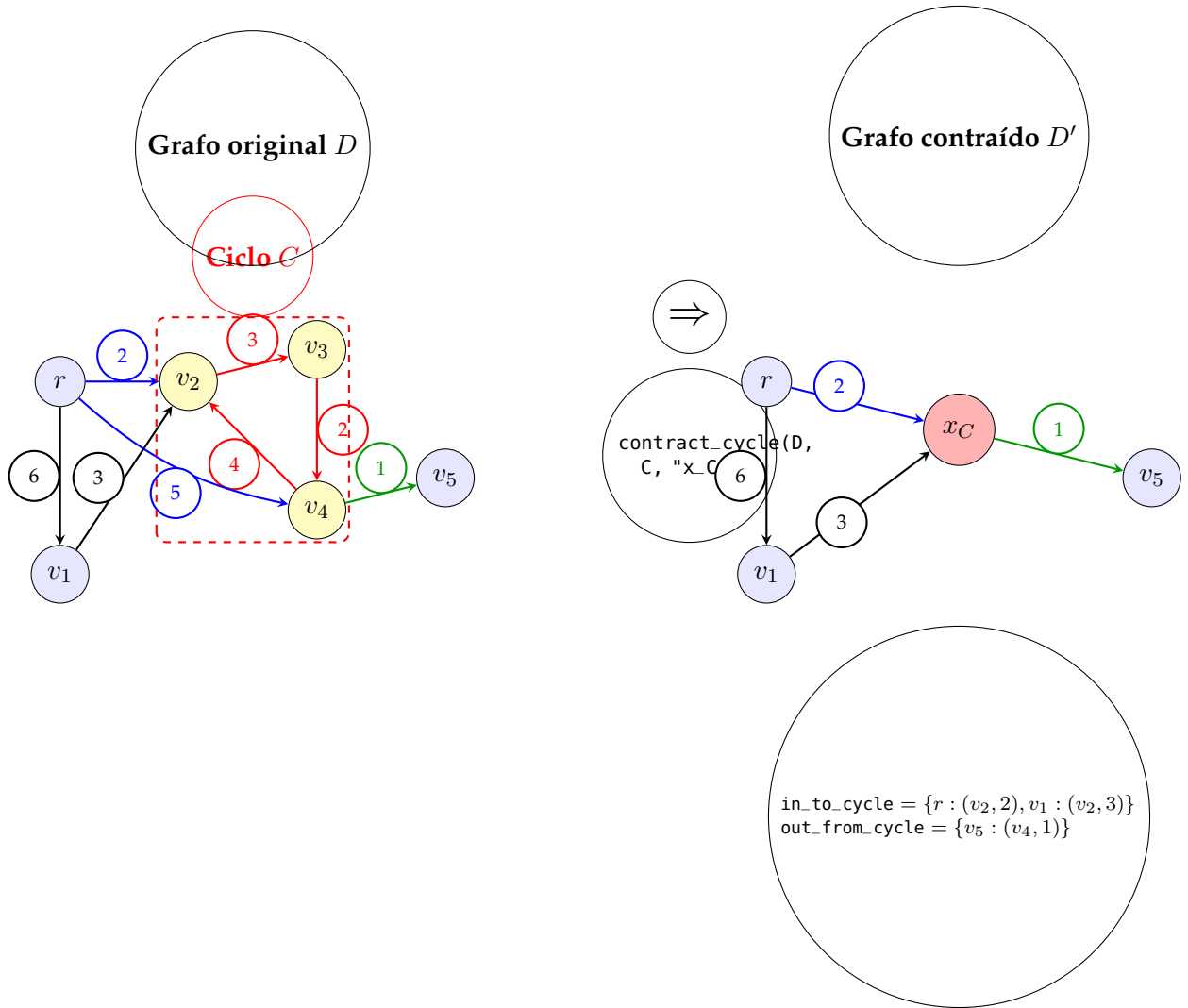


Figura 7 – Exemplo de contração de ciclo. À esquerda, grafo original D com ciclo $C = \{v_2, v_3, v_4\}$ (em amarelo). Vértices externos r , v_1 e v_5 têm arestas conectando ao ciclo: r envia aresta para v_2 (peso 2) e v_4 (peso 5); v_4 envia aresta para v_5 (peso 1). À direita, após a contração: o ciclo 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 detecção 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 (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 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 $D' = (V', A')$.

1.2.7 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 , criaríamos 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 `r0` 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_0 , modificando D in-place e retornando o grafo atualizado.

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

1.2.8 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 8 ilustra o objetivo da função:

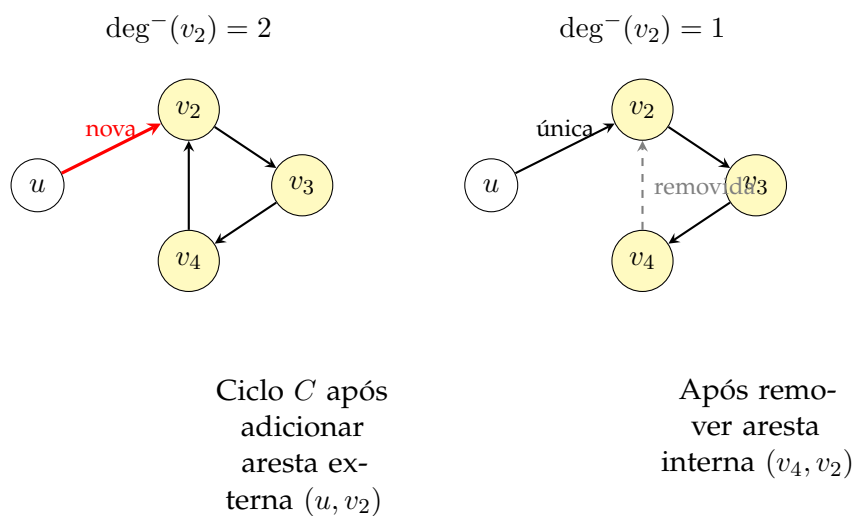


Figura 8 – Remoção de aresta interna durante reexpansão. À esquerda, ciclo $C = \{v_2, v_3, v_4\}$ após adicionar aresta externa (u, v_2) vindoura da arborescência T' : o vértice v_2 tem grau de entrada 2 (aresta externa vermelha de u e aresta interna do ciclo vinda de v_4), violando a propriedade de 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.

1.2.9 Procedimento principal (recursivo):

Esta função implementa o algoritmo de Chu–Liu/Edmonds de forma recursiva, orquestrando todas as fases do método 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 diretamente os cinco passos da descrição teórica, cada um apresentado a seguir com sua correspondência código-teoria:

Passo 1: Cópia do grafo original

Descrição teórica: Antes de iniciar o algoritmo, preserve o grafo original $D = (V, A)$ com sua função de custo $c : A \rightarrow \mathbb{R}$, pois as transformações subsequentes modificarão custos in-place.

Implementação (linha 2): Cria uma cópia do grafo original `D_copy = D.copy()` para preservar os pesos originais.

Justificativa: A cópia é essencial porque os passos de normalização e contração modificam os pesos das arestas in-place. Sem preservar os valores originais, seria impossível restaurar os custos corretos na arborescência final (passo 5), violando a correção do algoritmo. A biblioteca NetworkX cria uma cópia profunda da estrutura do grafo, incluindo todos os atributos de arestas e vértices, com complexidade $O(m + n)$.

Passo 2: Normalização e construção de F^*

Descrição teórica: Para cada $v \neq r$, escolha $a_v \in \arg \min_{(u,v) \in A} c(u, v)$. Defina o potencial dual $y(v) := c(a_v)$ e construa $F^* := \{a_v : v \neq r\}$ contendo exatamente uma aresta de entrada por vértice não-raiz.

Implementação (linhas 3–6): Normaliza os custos de todas as arestas que entram em cada vértice $v \neq r_0$ (linhas 3–5): para cada vértice, itera sobre todos os vértices do grafo, excluindo a raiz, e chama `normalize_incoming_edge_weights(D_copy, v)`, que implementa a definição de potencial $y(v) := c(a_v)$ onde $a_v \in \arg \min_{(u,v) \in A} c(u, v)$. A função subtrai $y(v)$ de cada aresta de entrada (u, v) , criando custos reduzidos $c'(u, v) = c(u, v) - y(v)$. Após a normalização completa, constrói o grafo funcional F^* (linha 6) chamando `get_Fstar(D_copy, r0)`, que implementa exatamente $F^* := \{a_v : v \neq r\}$, selecionando uma aresta de custo reduzido zero entrando em cada vértice não-raiz.

Justificativa: A normalização garante que cada vértice tenha pelo menos uma entrada de custo zero, propriedade fundamental que permite construir F^* selecionando arestas de custo reduzido zero sem perder otimalidade (pela teoria de dualidade e

complementaridade primal-dual). Selecionar apenas arestas de custo reduzido zero (ao invés de simplesmente o mínimo) é crucial porque garante que a solução seja ótima em relação aos custos normalizados, e a normalização preserva a estrutura de otimalidade do problema original.

Passo 3: Verificação de aciclicidade (caso base)

Descrição teórica: Se (V, F^*) é acíclico, então F^* é uma r -arborescência. Por Observação 4.36 de (KLEINBERG; TARDOS, 2006), trata-se de uma r -arborescência de custo mínimo. Devolva F^* com custos originais restaurados.

Implementação (linhas 7–10): Verifica se F^* é uma arborescência válida usando `nx.is_arborescence(F_star)` (linha 7), que testa se o grafo é conexo, acíclico e possui exatamente uma entrada por vértice (exceto a raiz). Se a verificação retorna `True`, o algoritmo alcançou o caso base da recursão. A implementação restaura os pesos originais de D para cada aresta de F^* (linhas 8–9): itera sobre todas as arestas (u, v) em `F_star.edges` e copia o atributo de peso `'weight'` do grafo original D para a aresta correspondente em `F_star`. Retorna `F_star` como a r -arborescência de custo mínimo (linha 10).

Justificativa: O uso de `nx.is_arborescence` (ao invés de apenas verificar aciclicidade com `nx.is_directed_acyclic_graph`) garante não apenas ausência de ciclos, mas também conectividade e grau de entrada correto ($\deg^-(v) = 1$ para todo $v \neq r$), todas propriedades necessárias para certificar que F^* é uma r -arborescência válida. A restauração dos pesos originais é obrigatória porque o algoritmo trabalha com custos reduzidos internamente, mas deve retornar uma solução expressa nos custos do problema original; a Observação 4.36 de Kleinberg garante que a estrutura ótima encontrada com custos reduzidos corresponde à solução ótima nos custos originais.

Passo 4: Contração e resolução recursiva (caso recursivo)

Descrição teórica: Caso contrário, existe um ciclo dirigido C em F^* (com $r \notin C$). Contração: contraia C em um supervértice x_C e defina custos reduzidos 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 arestas paralelas. Denote o digrafo contraído por $D' = (V', A')$. Resolva recursivamente em D' obtendo arborescência ótima F' .

Implementação (linhas 11–16): Caso F^* contenha um ciclo, entra no caso recursivo (linha 11: `else`). Detecta um ciclo C chamando `C = find_cycle(F_star)` (linha 12), que utiliza `nx.find_cycle` para encontrar um ciclo dirigido em F^* . Cria um rótulo único `contracted_label = f"contracted_{level}"` para o supervértice x_C usando o nível de recursão (linha 13), e contrai o ciclo chamando `in_to_cycle, out_from_cycle = contract_cycle(D_copy, C, contracted_label)` (linhas 14–15). Esta função implementa a operação "contraia C em x_C " e os ajustes de custos especificados nas fórmulas acima: para cada aresta externa entrando no ciclo, calcula $c'(u, x_C) := c(u, w) - y(w)$; para arestas saindo, define $c'(x_C, v) := c(w, v)$. A função modifica `D_copy` in-place criando o digrafo contraído $D' = (V', A')$ e retorna os dicionários `in_to_cycle` e `out_from_cycle` que armazenam os mapeamentos necessários para a reexpansão. Chama-se recursivamente (linha 16) com `F_prime = find_optimum_arborescence_chuliu(D_copy, r0, level+1)`, obtendo a arborescência ótima F' da instância reduzida D' .

Justificativa: O rótulo baseado em `level` garante que supervértices em diferentes níveis de recursão tenham identificadores únicos, evitando colisões de nomes. A contração reduz o problema a uma instância menor preservando a estrutura de otimalidade (provado pela teoria de dualidade e potenciais), permitindo resolver recursivamente. A modificação in-place (ao invés de criar nova cópia) reduz significativamente o uso de memória em grafos grandes ou com múltiplos níveis recursivos. Os dicionários são essenciais para a reexpansão correta, pois armazenam qual vértice do ciclo original deve receber cada conexão externa identificada pela solução recursiva.

Passo 5: Reexpansão do ciclo contraído

Descrição teórica: Dada a arborescência ótima F' em D' , expanda x_C de volta para o ciclo original C . Se $(u, x_C) \in F'$, adicione (u, v) onde v é o vértice do ciclo original mapeado por u , remova a aresta interna que entrava em v para manter $\deg^-(v) = 1$, e reintegre as demais arestas de C e suas saídas. Restaure custos originais e devolva a arborescência final.

Implementação (linhas 17–30): Na fase de reexpansão, a implementação desfaz a substituição do supervértice pelo ciclo original. Identifica a aresta externa que entra no supervértice em F' : `external_edge = list(F_prime.in_edges(contract_label))[0]` (linha 17). Extrai o vértice externo u , `_ = external_edge` dessa aresta (linha 18). Consulta `v = in_to_cycle[u]` para determinar o vértice v dentro do ciclo original que deve receber essa conexão (linha 19), implementando o mapeamento in-

verso da contração. Remove a aresta interna que entrava em v chamando `remove_internal_edge_to_cycle_entry(C, v)` (linha 20), "quebrando" o ciclo C no ponto de entrada v e garantindo que cada vértice mantenha exatamente uma entrada. Adiciona a aresta externa `F_prime.add_edge(u, v)` a F' (linha 21). Reintegra todas as arestas restantes do ciclo C usando `F_prime.add_edges_from(C.edges)` (linhas 22–23). Processa as arestas de saída do supervértice (linhas 24–26): para cada aresta `(contracted_label, w)` em F' , consulta o vértice origem real `v_out = out_from_cycle[w]` e adiciona `F_prime.add_edge(v_out, w)`. Remove o supervértice `F_prime.remove_node(contract_label)` de F' (linha 27), completando a substituição de x_C pelo ciclo modificado. Por fim, restaura os pesos originais de D para todas as arestas de F' (linhas 28–29): itera sobre `F_prime.edges` e copia os atributos `'weight'` do grafo original, desfazendo os custos reduzidos. Retorna `F_prime` como a r -arborescência de custo mínimo (linha 30).

Justificativa: A remoção da aresta interna é obrigatória porque o vértice v recebe uma nova entrada vinda do exterior; manter ambas violaria a propriedade de arborescência ($\deg^-(v) = 1$ para todo $v \neq r$). A escolha de qual aresta remover é determinada univocamente pela solução recursiva em D' : a aresta externa escolhida em F' determina o ponto de entrada no ciclo, e a aresta interna correspondente deve ser removida. A reintegração do ciclo (menos uma aresta) combina a solução recursiva no grafo contraído com a estrutura cíclica original, preservando otimalidade: o ciclo reduzido contribui com $|C| - 1$ arestas, e a aresta externa estabelece a conexão correta com o resto da arborescência. Como no caso base (Passo 3), a restauração é necessária para que a solução retornada reflita os custos originais do problema, não os custos reduzidos usados internamente pelo algoritmo.

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 no Passo 1 (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.

Essa decomposição em cinco passos explícitos demonstra como o algoritmo de Chu–Liu/Edmonds se traduz em operações concretas: o Passo 1 preserva o grafo original (linha 2), o Passo 2 normaliza e constrói F^* (linhas 3–6), o Passo 3 verifica aciclicidade e retorna no caso base (linhas 7–10), o Passo 4 contrai e resolve recursivamente (linhas 11–16), e o Passo 5 reexpande e restaura custos (linhas 17–30). A estratégia primal do método — induzir custos reduzidos zero, extrair estrutura funcional, e resolver ciclos por contração/expansão — emerge naturalmente dessa correspondência entre formulação teórica e código, com cada escolha de implementação fundamentada em necessidades de correção, eficiência ou clareza algorítmica.

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_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_chuliu(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 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(iter(F_prime.in_edges(contracted_label, data="w")),
18                          None)
19        u, _, _ = in_edge
20        v, _ = in_to_cycle[u]
21        remove_internal_edge_to_cycle_entry(C, v)
22        F_prime.add_edge(u, v)
23        for u_c, v_c in C.edges:
24            F_prime.add_edge(u_c, v_c)
25        for _, z, _ in F_prime.out_edges(contracted_label, data=True):
26            u_cycle, _ = out_from_cycle[z]
27            F_prime.add_edge(u_cycle, z)
28        F_prime.remove_node(contracted_label)
29        for u, v in F_prime.edges:
30            F_prime[u][v]["w"] = D[u][v]["w"]
31    return F_prime

```

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.

1.2.9.1 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.

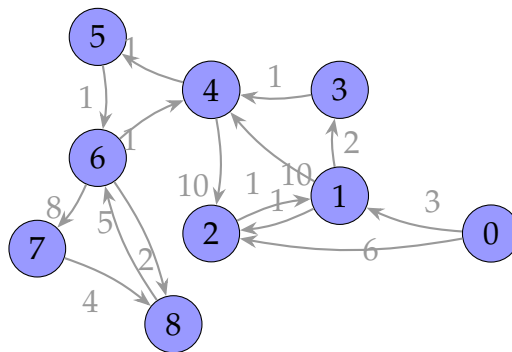


Figura 9 – Grafo direcionado ponderado inicial com raiz no vértice 0. O grafo contém 9 vértices e múltiplas arestas com pesos variados. O primeiro passo do algoritmo seria remover arestas que entram na raiz, porém não há nenhuma neste caso, logo não existe necessidade de alterar o grafo.

O 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).

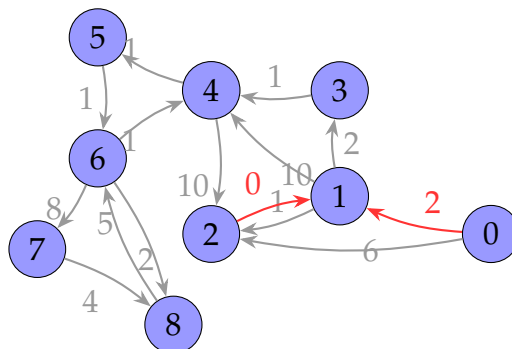


Figura 10 – Normalização parcial das arestas de entrada para o vértice 1. As arestas de entrada são $(0 \rightarrow 1)$ com peso original 3 e $(2 \rightarrow 1)$ com peso original 1. Elegendo a aresta $(2 \rightarrow 1)$ como a de menor peso (peso mínimo = 1), subtraímos este valor de todas as arestas 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 $n * 0$.

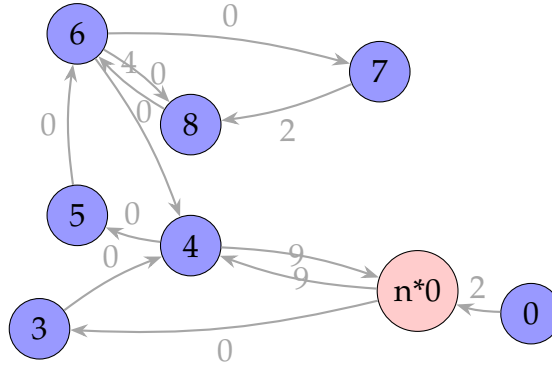


Figura 11 – Grafo contraído após detecção do ciclo $C = \{1, 2\}$ em F^* . O ciclo foi contraído no supervértice $n * 0$ (destacado em vermelho). As arestas que entravam ou saíam do ciclo foram redirecionadas para o supervértice, com custos ajustados segundo as fórmulas $c'(u, x_C) := c(u, w) - y(w)$ 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.

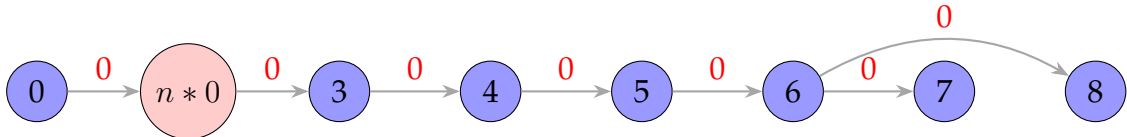


Figura 12 – Arborescência ótima F' obtida no grafo contraído. Todas as arestas selecionadas têm custo reduzido 0 (destacados em vermelho), e o grafo forma uma arborescência válida enraizada em 0: cada vértice (exceto a raiz) tem exatamente uma aresta de entrada, não há ciclos, e todos os vértices são 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.

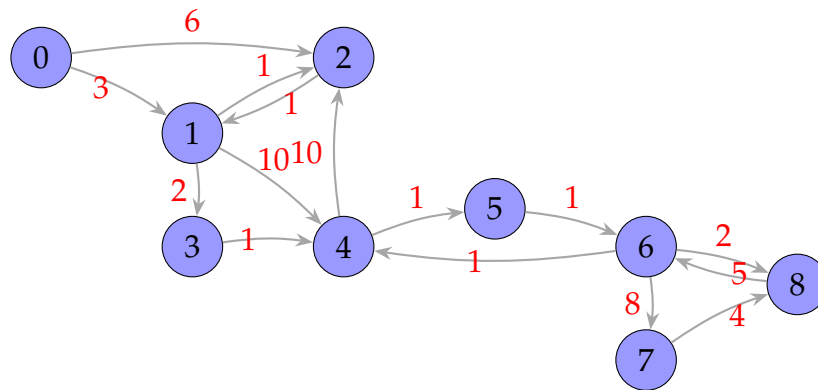


Figura 13 – Arborescência ótima final no grafo original com pesos restaurados. O supervértice $n * 0$ foi expandido de volta para os vértices 1 e 2, com a aresta externa $(0, 1)$ escolhida pela solução recursiva conectando ao ciclo. A aresta interna $(2, 1)$ do ciclo original foi removida para manter 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.

1.2.10 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 potenciais² $y(\cdot)$, explica os custos reduzidos e introduz a noção de cortes apertados (família laminar) como guias das contrações. Veremos como a mesma mecânica operacional (normalizar \rightarrow contrair \rightarrow expandir) emerge de condições duais que também sugerem otimizações e generalizações.

² 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 [12](#), [13](#), [14](#), [26](#) e [31](#).

SCHRIJVER, A. *Combinatorial Optimization: Polyhedra and Efficiency*. [S.l.]: Springer, 2003. Citado 4 vezes nas páginas [12](#), [13](#), [14](#) e [17](#).

Anexos

ANEXO A – Anexo A

Conteúdo do anexo A.