

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. 10
- Figura 2 – Ajuste de custo reduzido para um arco entrando em um ciclo contraído: o arco (u, w) com $w \in C$ torna-se (u, x_C) com custo reduzido $c'(u, x_C) = c(u, w) - c(a_w)$, onde a_w é o arco de menor custo que entra em w 11
- Figura 3 – 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. 12
- Figura 4 – 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). 12
- Figura 5 – 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^* 21
- Figura 6 – 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. 23
- Figura 7 – 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. 25

Figura 8 – 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.

Sumário

1	ALGORITMO DE CHU-LIU/EDMONDS	10
1.1	O problema dos ciclos e a solução por contração	10
1.1.1	Supervértices e contração de ciclos	11
1.2	Descrição do algoritmo	11
1.2.1	Exemplo prático: Chu-Liu/Edmonds	13
1.2.2	Corretude	15
1.2.3	Complexidade	16
1.3	Implementação em Python	16
1.3.1	Representação de Digrafos: NetworkX	17
1.3.1.1	Estrutura Interna	17
1.3.1.2	Operações Fundamentais	17
1.3.1.3	Busca em Profundidade (DFS)	18
1.3.1.3.1	Funcionamento básico:	18
1.3.1.3.2	Deteção de ciclos em digrafos:	18
1.3.1.3.3	Complexidade:	18
1.3.1.3.4	Implementação em NetworkX:	19
1.3.2	Especificação do Algoritmo	19
1.3.3	Normalização por vértice	20
1.3.4	Construção de F^* :	21
1.3.5	Deteção de ciclo:	24
1.3.6	Contração de ciclo:	25
1.3.7	Remoção de arestas que entram na raiz:	30
1.3.7.1	Remoção de arco interno:	31
1.3.7.2	Procedimento principal (recursivo):	31
1.3.7.3	Notas finais sobre a implementação	34
1.3.7.4	Decisões de projeto e implicações práticas	35
1.3.7.5	Transição para a abordagem primal-dual	35
	REFERÊNCIAS	37
	ANEXOS	38
	ANEXO A – ANEXO A	39

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.

1.1 O problema dos ciclos e a solução por contração

Em 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.

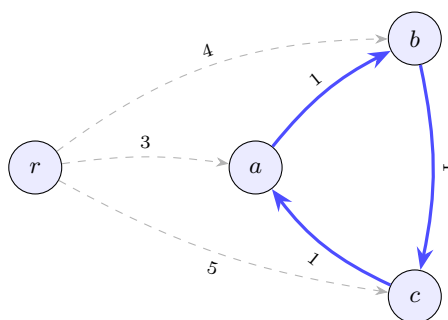


Figura 1 – Ciclo gerado pelas escolhas locais "mais baratas por vértice". Os arcos grossos (custo 1) entram em a, b, c e formam $a \rightarrow b \rightarrow c \rightarrow a$. Os arcos tracejados partindo de r existem, mas são mais caros e por isso não são escolhidos pelo critério local.

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.

1.1.1 Supervértices e contração de ciclos

Dado um subconjunto $C \subseteq V$ que forma um ciclo dirigido, a *contração de* C substitui todos os vértices de C por um único vértice x_C — o **supervértice**. Todo arco com exatamente uma ponta em C passa a ser incidente a x_C : arcos (u, w) com $u \notin C$, $w \in C$ tornam-se (u, x_C) ; arcos (w, v) com $w \in C$, $v \notin C$ tornam-se (x_C, v) ; e arcos com ambas as pontas em C são descartados.

Para preservar a comparação relativa dos custos, ajustamos os arcos que *entram* em C : para um arco (u, w) com $w \in C$, definimos $c'(u, x_C) = c(u, w) - c(a_w)$, onde a_w é o arco mais barato que entra em w . Essa normalização garante que decisões ótimas no grafo contraído podem ser traduzidas de volta na expansão.

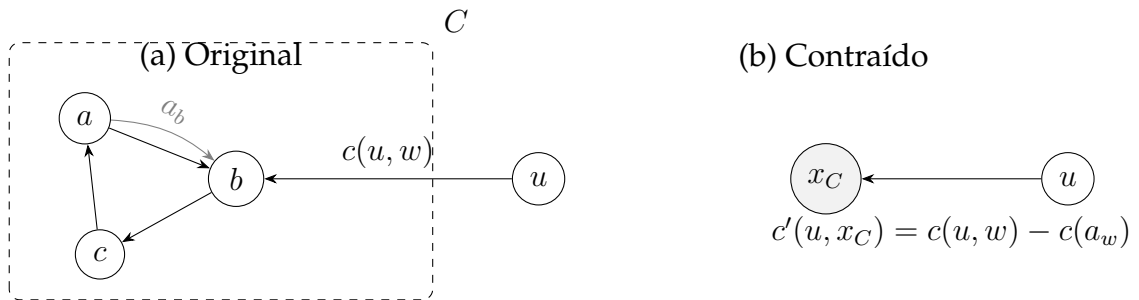


Figura 2 – Ajuste de custo reduzido para um arco entrando em um ciclo contraído: o arco (u, w) com $w \in C$ torna-se (u, x_C) com custo reduzido $c'(u, x_C) = c(u, w) - c(a_w)$, onde a_w é o arco de menor custo que entra em w .

A Figura 2 mostra o ajuste: o arco (u, b) com custo 7 torna-se (u, x_C) com custo reduzido $7 - 5 = 2$, já que $a_b = (a \rightarrow b)$ tem custo 5.

1.2 Descrição do algoritmo

Apresentamos o algoritmo em visão operacional de alto nível, focando na lógica e nos passos principais. 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.

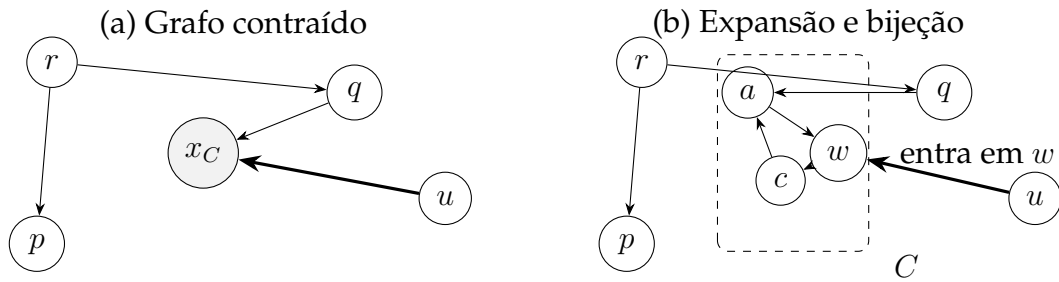


Figura 3 – Bijeção entre arborescências no grafo contraído e no original: toda arborescência em D' escolhe exatamente um arco que entra em x_C ; ao expandir C , esse arco corresponde a um (u, w) que entra em algum $w \in C$ e os arcos internos (de custo reduzido zero) são mantidos, preservando o custo 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).

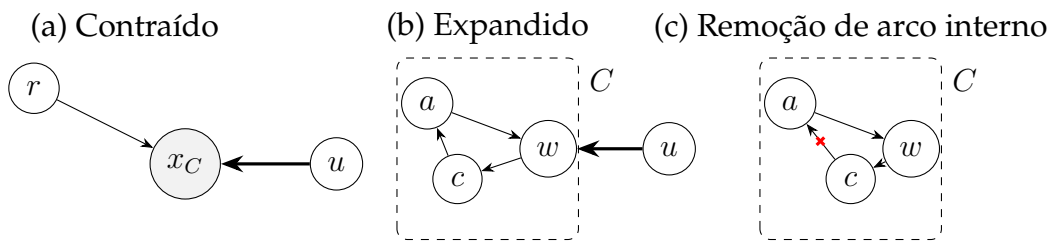


Figura 4 – Reexpansão de C : no grafo contraído seleciona-se um arco que entra em x_C ; ao expandir, x_C é substituído por C e o arco selecionado entra em algum $w \in C$; remove-se exatamente um arco interno de C para eliminar o ciclo, preservando conectividade e custo total (arcos internos têm custo reduzido zero).

Abaixo, a descrição formal do algoritmo.

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

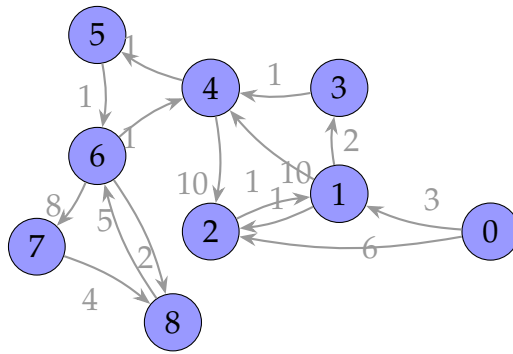
$$T := (T' \setminus \{\text{arcos incidentes a } x_C\}) \cup \{(u, w)\} \cup ((F^* \cap A(C)) \setminus \{a_w\}).$$

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.

1.2.1 Exemplo prático: Chu–Liu/Edmonds

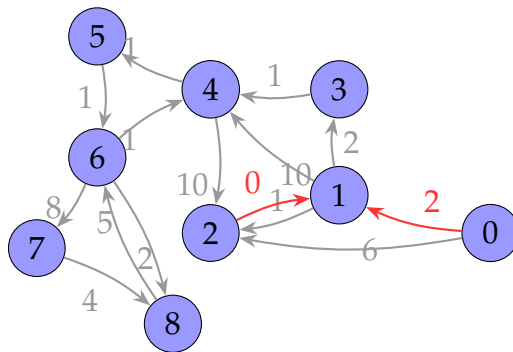
A seguir, ilustramos o funcionamento do algoritmo de Chu–Liu/Edmonds em um grafo de teste. Mostramos o grafo original, os principais passos do algoritmo e a arborescência final encontrada. A Figura abaixo apresenta o grafo original com os pesos das arestas



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.

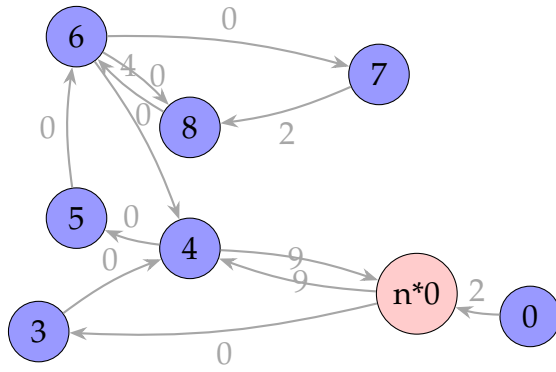
Dessa forma, o próximo passo é normalizar os pesos das arestas de entrada para cada vértice, nessa etapa, Para cada vértice X (exceto a raiz), o algoritmo encontra a aresta de menor peso que entra em X e subtrai esse menor peso de todas as arestas que entram em X (relembrando que isso serve para zerar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

Normalizando pesos de arestas de entrada para '1': Nesse processo notamos que as únicas arestas de entrada são 0 e 2 onde $(0 \rightarrow 1)$ tem peso 3.0 e $(2 \rightarrow 1)$ tem peso 1.0, elegendo a aresta 2 como a de menor peso podemos subtrair o peso das arestas restantes (no caso, o peso da aresta 0) pelo valor do peso da aresta 2, resultando em um novo peso de '2' para a aresta 0

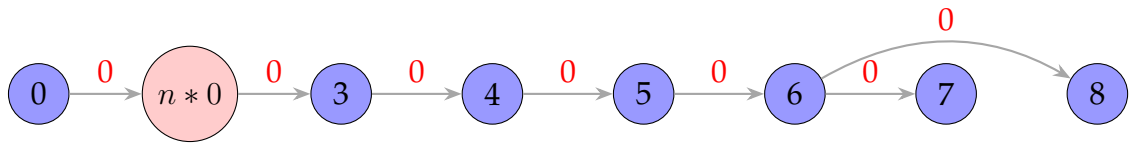


Repetiremos o passo anterior para todas as outras arestas

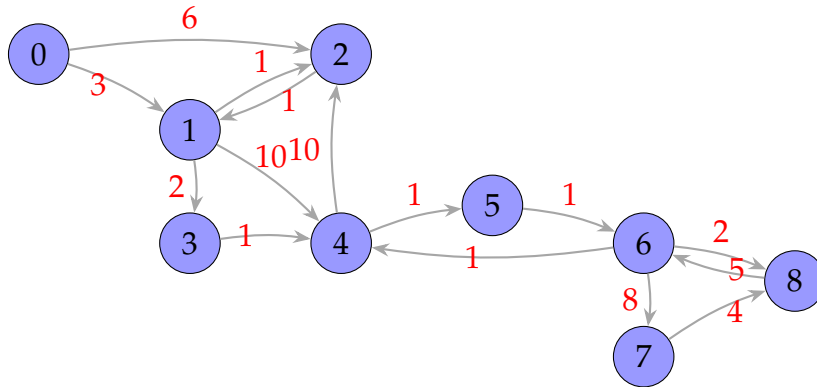
Com os pesos normalizados, o próximo passo é construir F^* , para isso, selecionamos para cada vértice, a aresta de menor custo de entrada. Além disso, detectamos um ciclo em F^* , formado pelos vértices $\{1$ e $2\}$. Portanto, precisamos contrair esse ciclo em um supervértice $n * 0$. O resultado é o seguinte:



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



Após validarmos que a F^* não possui mais ciclos e notarmos que F^* forma uma arborescência iremos começar o processo de expansão do ciclo contraído para obter a arborescência final no grafo original. Dessa forma, Adicionamos a aresta de entrada ao ciclo: $(0, 1)$, $(1, 2)$ e a aresta externa de saída: $(1, 3)$, chegando em uma arborescência válida.



1.2.2 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.

1.2.3 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>).

1.3 Implementação em Python

Esta seção apresenta uma implementação em Python do algoritmo de Chu–Liu/Edmonds. 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.

1.3.1 Representação de Digrafos: NetworkX

A implementação utiliza a biblioteca NetworkX¹, que fornece estruturas de dados eficientes para grafos e digrafos. A classe `nx.DiGraph` representa grafos direcionados (directed graphs) e constitui a base para todas as operações do algoritmo.

1.3.1.1 Estrutura Interna

Internamente, `nx.DiGraph` armazena o grafo usando dicionários aninhados do Python. Para um digrafo $D = (V, A)$:

- **Vértices:** mantidos em um dicionário que mapeia cada vértice para seus atributos. O método `D.nodes()` retorna uma visão (`NodeView`) sobre o conjunto de vértices, permitindo iteração em tempo $O(n)$.
- **Arestas:** armazenadas em estruturas de adjacência bidirecionais. Para cada vértice u , mantém-se:
 - Um dicionário de *sucessores*: vértices v tais que $(u, v) \in A$, acessível via `D[u]`.
 - Um dicionário de *predecessores*: vértices w tais que $(w, u) \in A$, usado por `D.in_edges(u)`.
- **Atributos de arestas:** cada aresta pode ter atributos arbitrários armazenados como dicionários. A notação `D[u][v][w]` acessa o atributo "w" (peso) da aresta (u, v) .

Esta representação garante acesso eficiente: adicionar ou remover uma aresta tem complexidade $O(1)$ em média, consultar os vizinhos de um vértice custa $O(\deg(v))$, e iterar sobre todas as arestas leva tempo $O(m)$.

1.3.1.2 Operações Fundamentais

As operações básicas usadas na implementação incluem:

- `D.nodes()`: retorna visão sobre os vértices, permitindo iteração e verificação de pertinência.
- `D.in_edges(v, data="w")`: retorna as arestas que entram em v , opcionalmente incluindo atributos. Com `data="w"`, produz tuplas (u, v, w) onde w é o peso.

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

- `D.out_edges(u, data="w")`: retorna as arestas que saem de u , análogo a `in_edges`.
- `D.add_edge(u, v, **attr)`: adiciona a aresta (u, v) com atributos opcionais. Se u ou v não existirem, são criados automaticamente.
- `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.

1.3.1.3 Busca em Profundidade (DFS)

A busca em profundidade (Depth-First Search, DFS) é um algoritmo fundamental de travessia de grafos que explora sistematicamente cada ramo do grafo até sua máxima profundidade antes de retroceder. Em digrafos, a DFS é particularmente útil para detectar ciclos dirigidos, uma operação essencial no algoritmo de Chu–Liu/Edmonds.

1.3.1.3.1 Funcionamento básico:

A partir de um vértice inicial s , a DFS marca s como visitado e recursivamente visita cada vizinho não visitado. Quando todos os vizinhos de um vértice foram explorados, o algoritmo retrocede (backtrack) para o vértice anterior na pilha de recursão.

1.3.1.3.2 Detecção de ciclos em digrafos:

Para detectar ciclos, a DFS mantém três estados para cada vértice:

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

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

1.3.1.3.3 Complexidade:

A DFS visita cada vértice exatamente uma vez e examina cada aresta no máximo uma vez (ao explorar os vizinhos). Portanto, a complexidade é $O(n + m)$, onde $n = |V|$ e $m = |A|$.

1.3.1.3.4 Implementação em NetworkX:

A biblioteca NetworkX fornece a função `nx.find_cycle(G, orientation="original")` que implementa detecção de ciclos baseada em DFS. Esta função aceita dois parâmetros principais: o grafo G a ser analisado e o parâmetro opcional `orientation` que controla o formato de saída das arestas encontradas.

Quando invocada, a função percorre o grafo executando DFS a partir de cada vértice não visitado. Durante a travessia, mantém o estado de cada vértice (não visitado, em processamento, ou concluído) e, ao detectar uma aresta de retorno — isto é, uma aresta (u, v) onde v está atualmente na pilha de recursão —, identifica a presença de um ciclo.

O retorno da função é um *iterador* (não uma lista materializada) sobre as arestas que compõem o ciclo detectado. Cada elemento do iterador é uma tupla (u, v, key) quando `orientation="original"`, onde u e v são os vértices da aresta e `key` contém metadados de orientação (que tipicamente ignoramos usando desempacotamento com `_`). A escolha de retornar um iterador em vez de uma lista reduz o uso de memória, permitindo processamento sob demanda das arestas do ciclo.

Um aspecto crucial do design desta função é seu comportamento em grafos acíclicos: em vez de retornar um valor especial como `None` ou uma lista vazia, a função lança a exceção `NetworkXNoCycle`. Esta decisão de design segue o princípio EAFP (*Easier to Ask for Forgiveness than Permission*) do Python, onde operações excepcionais são sinalizadas por exceções em vez de valores de sentinela. Isso força o código cliente a tratar explicitamente o caso acíclico com um bloco `try-except`, resultando em código mais robusto e com intenção clara.

A complexidade da função é $O(n + m)$, onde $n = |V|$ e $m = |A|$, pois no pior caso (grafo acíclico) a DFS visita todos os vértices e examina todas as arestas exatamente uma vez antes de concluir que não há ciclos.

1.3.2 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.

1.3.3 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 5 ilustra o funcionamento da normalização:

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

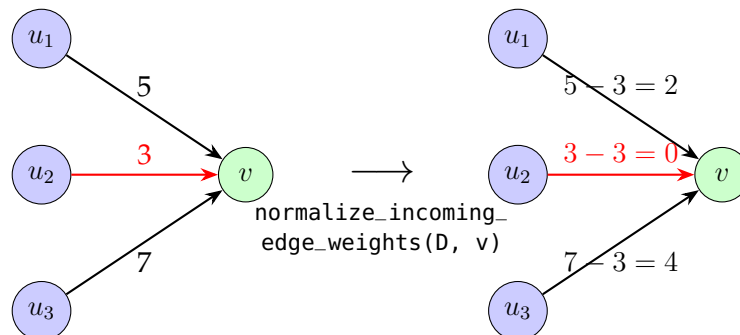


Figura 5 – Exemplo de normalização de custos reduzidos. À esquerda, vértice v com três arestas de entrada (pesos 5, 3 e 7). À direita, após aplicar `normalize_incoming_edge_weights(D, v)`: o menor peso $y(v) = 3$ é subtraído de todas as entradas, resultando em custos reduzidos 2, 0 e 4. A aresta (u_2,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

1.3.4 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 r_0 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 r_0 (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^*

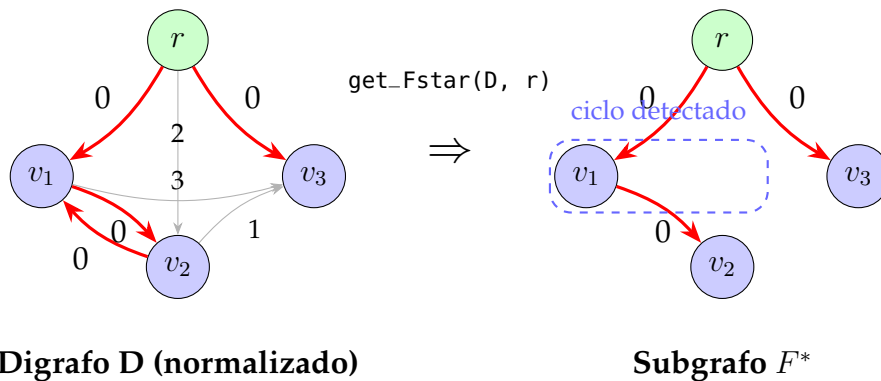
Constrói o subdigrafo F^* a partir do digrafo D , selecionando para cada vértice (exceto a raiz r_0) uma aresta de custo reduzido zero que entra nele.

```

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 6 ilustra a construção de F^* :



Digrafo D (normalizado)

Subgrafo F^*

Figura 6 – 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 $F^* := \{a_v : v \neq r\}$.

1.3.5 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 1.3.1.3). 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 7 ilustra o processo de detecção de ciclo:

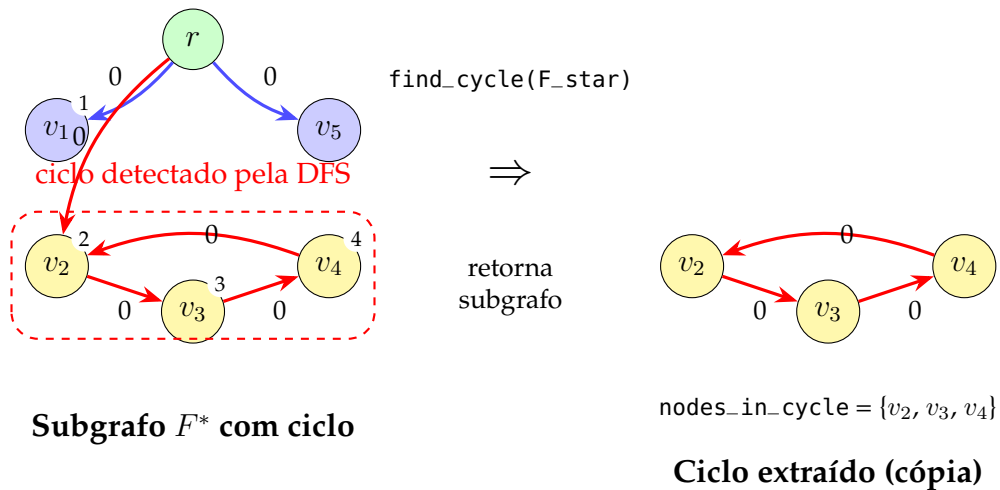


Figura 7 – Exemplo de detecção de ciclo em F^* . À esquerda, o subgrafo F^* contém um ciclo formado pelos vértices $\{v_2, v_3, v_4\}$ (destacados em amarelo). A DFS percorre o grafo e detecta o ciclo ao encontrar a aresta (v_4, v_2) , onde v_2 já está na pilha de recursão. À direita, a função retorna 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.

1.3.6 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

Contrai 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:

```

```
17         min_weight_edge_from_cycle = min(  
18             ((u, w) for u, _, w in D.in_edges(v, data="w") if u in  
                cycle_nodes), key=lambda x: x[1], default=None,  
19         if min_weight_edge_from_cycle:  
20             out_from_cycle[v] = min_weight_edge_from_cycle  
21     for v, (u, w) in out_from_cycle.items():  
22         D.add_edge(label, v, w=w)  
23     D.remove_nodes_from(cycle_nodes)  
24     return in_to_cycle, out_from_cycle
```

A Figura 8 ilustra o processo de contração de ciclo:

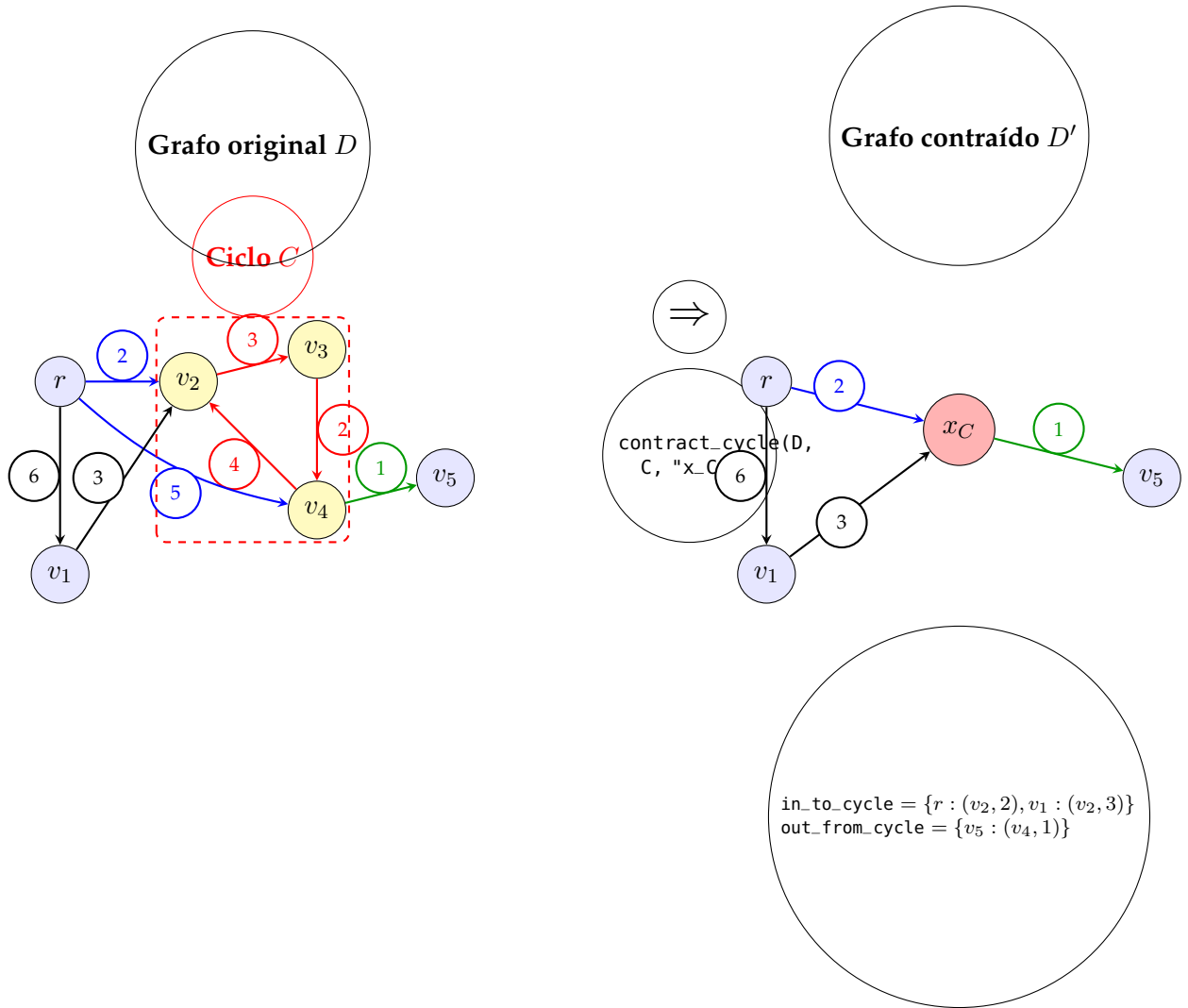


Figura 8 – Exemplo de contração de ciclo. À esquerda, grafo original D com ciclo $C = \{v_2, v_3, v_4\}$ (em amarelo). Vértices externos r , v_1 e v_5 têm arestas conectando ao ciclo: r envia aresta para v_2 (peso 2) e v_4 (peso 5); v_4 envia aresta para v_5 (peso 1). À direita, após a contração: o ciclo 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 $D' = (V', A')$.

1.3.7 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.

Recebe como entrada um digrafo D (objeto `nx.DiGraph`) e o rótulo `r0` da raiz. A implementação armazena em uma lista todas as arestas que entram em `r0` 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
```

1.3.7.1 Remoção de arco interno:

ao expandir o ciclo C , a função remove o arco interno que entra no vértice de entrada v do ciclo, já que v agora recebe um arco externo do grafo. A função modifica o subgrafo do ciclo *in-place* e executa em $O(\deg^-(v))$.

Remover arco interno na reexpansão

Remove a aresta interna que entra no vértice de entrada 'v' do ciclo C, pois 'v' passa a receber uma aresta externa do grafo.

```
1 % def remove_internal_edge_to_cycle_entry(C: nx.DiGraph, v):
2
3 %     predecessor = next((u for u, _ in C.in_edges(v)), None)
4
5 %     C.remove_edge(predecessor, v)
```

1.3.7.2 Procedimento principal (recursivo):

A função principal implementa o algoritmo de Chu–Liu/Edmonds de forma recursiva e atua como um orquestrador das fases do método. Em alto nível, ela mantém a seguinte lógica:

O procedimento principal do algoritmo segue estes passos: prepara a instância removendo entradas na raiz, normaliza os custos das arestas que entram em cada vértice (exceto a raiz) para garantir pelo menos uma entrada de custo reduzido zero, constrói o grafo funcional F^* escolhendo para cada vértice a entrada de menor custo reduzido, verifica se F^* é acíclico (se for, retorna como *r-arborescência ótima*), e, caso haja ciclo, contrai o ciclo em um supervértice, ajusta os custos das entradas e resolve recursivamente; ao retornar, expande o ciclo e remove uma aresta interna para garantir aciclicidade e grau de entrada igual a 1.

Mais especificamente, o procedimento garante as seguintes propriedades e passos:

- **Função (entradas/saídas):** Entrada: digrafo ponderado $D = (V, A)$, raiz r_0 , e, opcionalmente, funções `draw_fn` e `log` para visualização e registro. Saída: um subdigrafo dirigido T de D com $|V| - 1$ arcos em que todo $v \neq r_0$ tem grau de entrada 1, todos os vértices alcançam r_0 e o custo total $\sum_{a \in T} c(a)$ é mínimo.
- **Invariantes:** Após a normalização por vértice, cada $v \neq r_0$ tem pelo menos uma entrada de custo reduzido zero; o conjunto F^* contém exatamente uma entrada por vértice distinto de r_0 ; em toda contração, apenas arcos que *entram* no componente

têm seus custos reduzidos ajustados por $c'(u, x_C) = c(u, w) - c(a_w)$, preservando comparações relativas.

- **Deteção de ciclo e contração:** Se F^* contém um ciclo C , todos os seus arcos têm custo reduzido zero. O procedimento forma o supervértice x_C , reescreve arcos incidentes (descarta laços internos) e prossegue na instância menor. Essa etapa pode manter arcos paralelos e ignora laços.
- **Recursão e expansão:** Ao obter T' ótimo no grafo contraído, o método mapeia T' de volta para D : substitui o arco (u, x_C) por um (u, w) apropriado (com $w \in C$) e remove uma única aresta interna de C , restaurando a propriedade “uma entrada por vértice” e a aciclicidade.
- **Empates e robustez:** Empates de custo são resolvidos de modo determinístico/local, sem afetar a otimalidade. Arcos paralelos podem surgir após contrações e são tratados normalmente; laços são descartados por construção.
- **Logs e desenho (opcionais):** Na implementação disponibilizada no repositório do projeto integramos o solver com a interface do projeto de forma que se fornecidos, log recebe mensagens estruturadas por nível de recursão, e draw_fn e draw_step pode ser chamado para ilustrar passos relevantes (normalização, detecção/contração de ciclos, retorno da recursão e expansão).
- **Casos-limite:** Se algum $v \neq r_0$ não possui arco de entrada na instância corrente, detecta-se inviabilidade (não existe r-arborescência). Se F^* já é acíclico, retorna imediatamente (base da recursão).
- **Complexidade:** Em uma implementação direta, cada nível de recursão executa seleção/checagem/ajustes em tempo proporcional a $O(m)$, e há no máximo $O(n)$ níveis devido às contrações, totalizando $O(mn)$ e memória $O(m + n)$.

Essa rotina encapsula, portanto, a estratégia primal do método: induzir arestas de custo reduzido zero por normalização local, extrair uma estrutura funcional F^* de uma entrada por vértice, e resolver conflitos cíclicos por contração/expansão, preservando custos e correção em todas as etapas.

Procedimento principal (recursivo)

Função recursiva que encontra a arborescência ótima em um digrafo D com raiz r_0 usando o algoritmo de Chu–Liu/Edmonds.

```
1 % def find_optimum_arborescence_chuliu(
2 %     D: nx.DiGraph,
3 %     r0: str,
```

```

4 %     level=0,
5 % ):
6
7 %     D_copy = D.copy()
8
9 %     for v in D_copy.nodes:
10 %         if v != r0:
11 %             normalize_incoming_edge_weights(D_copy, v)
12
13 %     # Build F_star
14 %     F_star = get_Fstar(D_copy, r0)
15
16 %     if nx.is_arborescence(F_star):
17 %         for u, v in F_star.edges:
18 %             F_star[u][v]["w"] = D[u][v]["w"]
19 %         return F_star
20
21 %     else:
22 %         C: nx.DiGraph = find_cycle(F_star)
23
24 %         contracted_label = f"\n n*{level}"
25 %         in_to_cycle, out_from_cycle = contract_cycle(
26 %             D_copy, C, contracted_label
27 %         )
28
29 %         # Recursive call
30 %         F_prime = find_optimum_arborescence_chuliu(
31 %             D_copy,
32 %             r0,
33 %             level + 1
34 %         )
35
36 %         # Identify the vertex in the cycle that received the only incoming
37 %         edge from the arborescence
38 %         in_edge = next(iter(F_prime.in_edges(contracted_label, data="w")),
39 %             None)
40
41 %         u, _, _ = in_edge
42
43 %         v, _ = in_to_cycle[u]

```

```

42
43 %      # Remove the internal edge entering vertex 'v' from cycle C
44 %      remove_internal_edge_to_cycle_entry(
45 %          C, v
46 %      ) # Note: w is coming from F_prime, not from G
47
48 %      # Add the external edge entering the cycle (identified by in_edge)
49 %      , the weight will be corrected at the end using G
50 %      F_prime.add_edge(u, v)
51
52 %      # Add the remaining edges of the modified cycle C
53 %      for u_c, v_c in C.edges:
54 %          F_prime.add_edge(u_c, v_c)
55
56 %      # Add the external edges leaving the cycle
57 %      for _, z, _ in F_prime.out_edges(contracted_label, data=True):
58 %          u_cycle, _ = out_from_cycle[z]
59 %          F_prime.add_edge(u_cycle, z)
60
61 %      F_prime.remove_node(contract_label)
62
63 %      # Update the edge weights with the original weights from G
64 %      for u, v in F_prime.edges:
65 %          F_prime[u][v]["w"] = D[u][v]["w"]
66
67 %      return F_prime

```

1.3.7.3 Notas finais sobre a implementação

A implementação acima segue diretamente a descrição do algoritmo de Chu–Liu/Edmonds, enfatizando clareza e correção. Para aplicações práticas, otimizações podem ser introduzidas, como estruturas de dados eficientes para seleção de mínimos, detecção rápida de ciclos e manipulação de grafos dinâmicos. Além disso, a função pode ser adaptada para lidar com casos especiais, como grafos desconexos ou múltiplas raízes, conforme necessário.

A complexidade da implementação direta é $O(mn)$ no pior caso, onde m é o número de arestas e n o número de vértices, devido à potencial profundidade de recursão e ao processamento linear em cada nível. Implementações mais sofisticadas podem reduzir isso para $O(m \log n)$ usando estruturas avançadas, como heaps e union-find,

mas a versão apresentada prioriza a compreensão do algoritmo fundamental.

SAMIRA

1.3.7.4 Decisões de projeto e implicações práticas

Antes de prosseguir para uma visão alternativa do mesmo problema, vale destacar algumas decisões de projeto e implicações práticas da implementação de Chu–Liu/Edmonds:

- **Estruturas e efeitos colaterais:** Optamos por modificar grafos *in-place* (por exemplo, durante a normalização e a contração de ciclos) para reduzir alocações e facilitar a visualização incremental. Isso exige invariantes explícitos e cuidado com referências ativas ao grafo original.
- **Empates, paralelos e laços:** Empates são resolvidos de forma determinística/local sem afetar a otimalidade. A contração pode induzir *arcos paralelos*; preservamos apenas o de menor custo. Laços (self-loops) são descartados por construção.
- **Validação e testes:** O repositório inclui artefatos úteis para experimentação (por exemplo, `tests.py`, `test_results.csv`, `test_log.txt`). Onde um volume de grafos é gerado aleatoriamente, a função é executada e os resultados são validados são comparados com soluções de força bruta.
- **Integração com visualização e logs:** A função `draw_fn` permite registrar *snapshots* (normalização, formação de F^* , contração/expansão). O log facilita auditoria e depuração em execuções recursivas.
- **Extensões:** Variantes com múltiplas raízes, restrições adicionais (p.ex., proibições por partição) e empacotamento de arborescências exigem ajustes na fase de extração/expansão ou formulações via matroides.

1.3.7.5 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 4 vezes nas páginas [12](#), [13](#), [16](#) e [29](#).

SCHRIJVER, A. *Combinatorial Optimization: Polyhedra and Efficiency*. [S.l.]: Springer, 2003. Citado 4 vezes nas páginas [12](#), [13](#), [16](#) e [20](#).

Anexos

ANEXO A – Anexo A

Conteúdo do anexo A.