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).*

# Agradecimentos

Agradecimentos (opcional).

*“Epígrafe (opcional).”*

# Resumo

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

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

# Abstract

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

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

# Lista de ilustrações

# Sumário

# 1 Algoritmo de Chu–Liu/Edmonds

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 $a \to b \to c \to a$. 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.

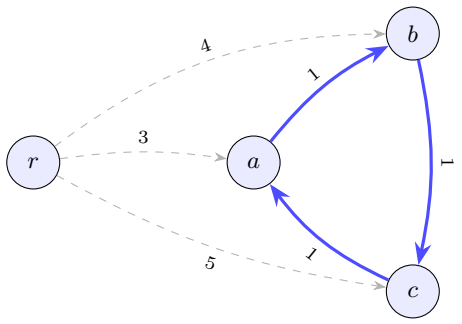



Figura 1 – Ciclo gerado pelas escolhas locais "mais baratas por vértice". Os arcos grossos (custo 1) entram em $a, b, c$ e formam $a \to b \to c \to a$. Os arcos tracejados partindo de $r$ existem, mas são mais caros e por isso não são escolhidos pelo critério local.

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.

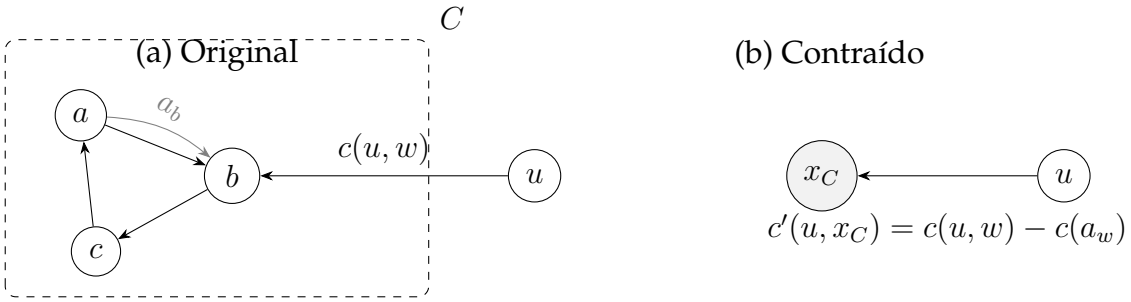



Figura 2 – Ajuste de custo reduzido para um arco entrando em um ciclo contraído: o arco $(u, w)$ com $w \in C$ torna-se $(u, x_C)$ com custo reduzido $c'(u, x_C) = c(u, w) - c(a_w)$, onde $a_w$ é o arco de menor custo que entra em $w$.

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 $a_b = (a \to b)$ 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.

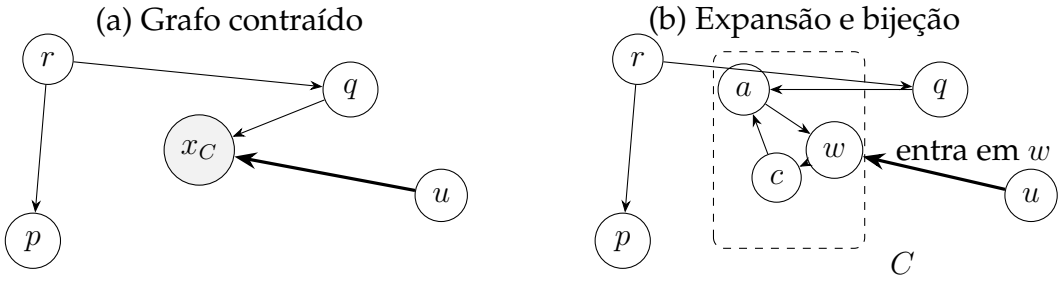



Figura 3 – Bijeção entre arborescências no grafo contraído e no original: toda arborescência em $D'$ escolhe exatamente um arco que entra em $x_C$; ao expandir $C$, esse arco corresponde a um $(u, w)$ que entra em algum $w \in C$ e os arcos internos (de custo reduzido zero) são mantidos, preservando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

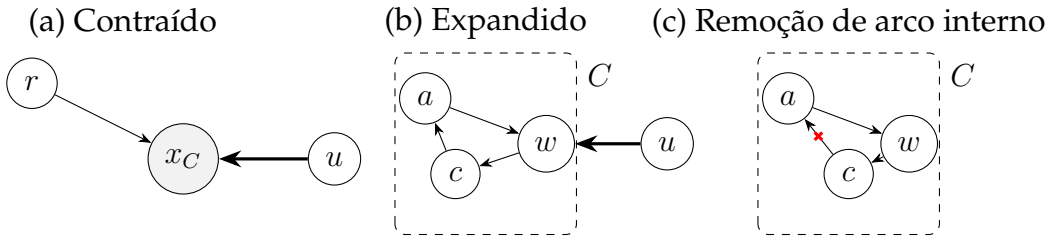



Figura 4 – Reexpansão de $C$: no grafo contraído seleciona-se um arco que entra em $x_C$; ao expandir, $x_C$ é substituído por $C$ e o arco selecionado entra em algum $w \in C$; remove-se exatamente um arco interno de $C$ para eliminar o ciclo, preservando conectividade e custo total (arcos internos 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 $c : A \to \mathbb{R}_{\geq 0}$, 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

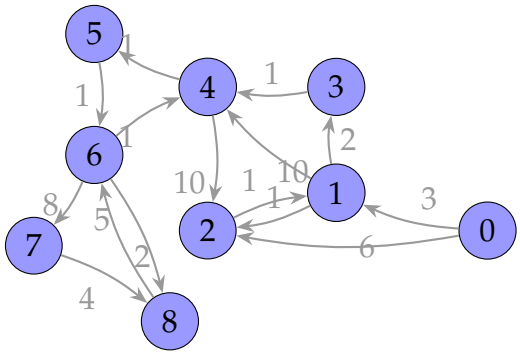
$$T := \big(T' \setminus \{\text{arcos incidentes a } x_C\}\big) \cup \{(u, w)\} \cup \big((F^* \cap A(C)) \setminus \{a_w\}\big).$$

   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

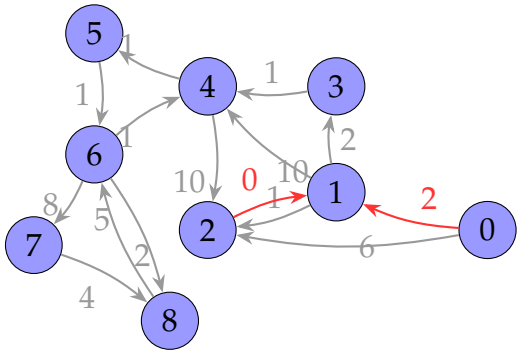
A seguir, ilustramos o funcionamento do algoritmo de Chu–Liu/Edmonds em um grafo de teste. Mostramos o grafo original, os principais passos do algoritmo e a arborescência final encontrada. A Figura abaixo apresenta o grafo original com os pesos das arestas




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.

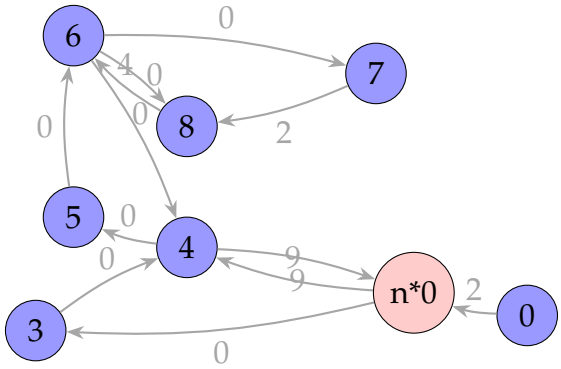
Dessa forma, o próximo passo é normalizar os pesos das arestas de entrada para cada vértice, nessa etapa, Para cada vértice X (exceto a raiz), o algoritmo encontra a aresta de menor peso que entra em X e subtrai esse menor peso de todas as arestas que entram em X (relembrando que isso serve para zerar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

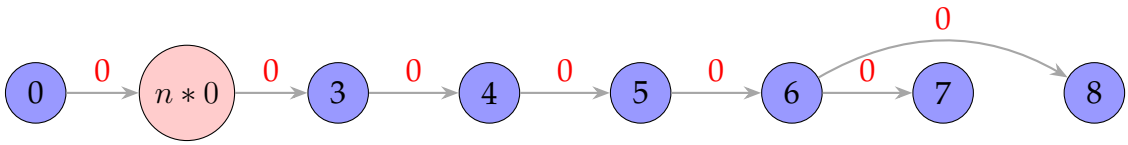



Repetiremos o passo anterior para todas as outras arestas

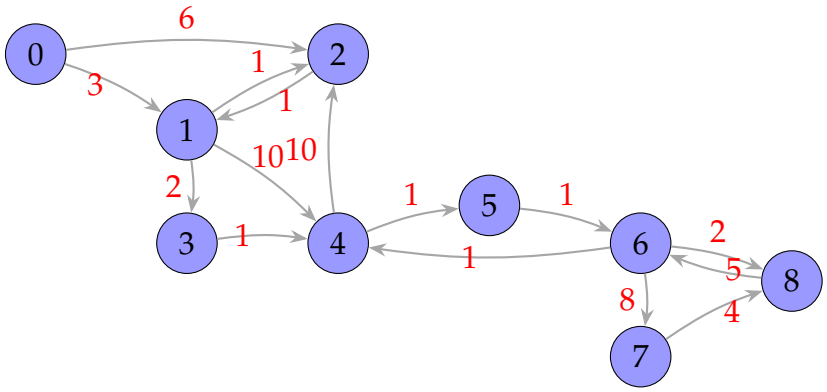
Com os pesos normalizados, o próximo passo é construir $F^*$, para isso, selecionamos para cada vértice, a aresta de menor custo de entrada. Além disso, detectamos um ciclo em $F^*$, formado pelos vértices $\{1$ e $2\}$. Portanto, precisamos contrair esse ciclo em um supervértice $n * 0$. O resultado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




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




Após validarmos que a F* não possuí mais ciclos e notarmos que F* forma uma arborescência iremos começar o processo de expanção 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.

- **Entrada:** digrafo ponderado $D = (V, A)$, custos $c : A \to \mathbb{R}$, raiz $r \in V$.
- **Hipóteses:**
  - $D$ é representado como um objeto `networkx.DiGraph`, com pesos armazenados no atributo de arestas `'w'`.
  - $D$ 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:** conjunto $A^* \subseteq A$ com $|A^*| = |V| - 1$, tal que cada $v \neq r$ tem grau de entrada 1, todos os vértices são alcançáveis a partir de $r$ e $\sum_{a \in A^*} c(a)$ é mínimo.
- **Convenções:** arcos paralelos (múltiplos arcos entre o mesmo par de vértices) são permitidos após contrações; laços (self-loops) são descartados.

A seguir, detalhamos as implementações das funções principais e auxiliares, começando pela normalização dos custos por vértice.

### 1.3.1 Normalização por vértice

Esta função normaliza[1] os custos das arestas que entram em um vértice $v$: calcula $y(v) = \min\{w(u, v)\}$ e substitui cada peso $w(u, v)$ por $w(u, v) - y(v)$.

A função recebe como entrada um digrafo `D` (objeto `nx.DiGraph`[2]) e o rótulo `node` do vértice cujas arestas de entrada devem ser normalizadas. A implementação coleta todas as arestas de entrada de `node` com seus pesos (linha 2) e, se a lista estiver vazia, retorna imediatamente sem fazer alterações (linhas 3–4). Caso contrário, calcula o peso mínimo `yv` usando uma compreensão de gerador[3] que extrai o terceiro elemento de

[1] Aqui, "normalizar"significa subtrair do peso de cada aresta que entra em $v$ o menor peso de entrada (custos reduzidos), preservando a ordem relativa; assim, ao menos uma entrada em $v$ passa a ter custo 0, sem afetar a comparação entre soluções.

[2] `nx.DiGraph` é a classe da biblioteca NetworkX que representa grafos dirigidos (directed graphs). Ela armazena vértices e arestas direcionadas, permitindo associar atributos arbitrários (como pesos) às arestas através de dicionários. A notação `D[u][v]["w"]` acessa o peso da aresta $(u, v)$.

[3] Em Python, uma *compreensão de gerador* (generator comprehension) é uma expressão da forma `(expr for item in iterable)` que produz valores sob demanda, sem criar uma lista completa na memória. Aqui, `(w for _, _, w in predecessors)` extrai apenas os pesos das tuplas, permitindo calcular o mínimo de forma eficiente.

cada tupla (linha 5) e, para cada predecessor `u`, subtrai `yv` do peso armazenado em `D[u][node]["w"]` (linha 6).

A função não retorna nenhum valor (retorno implícito `None`), pois a operação é realizada *in-place*[4]: o grafo `D` passado como parâmetro é modificado diretamente. Como efeito colateral, 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**

*Altera os pesos das arestas que entram em 'node' subtraindo de cada uma o menor peso de entrada no grafo D.*

```
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
```

### 1.3.2 Construção de $F^*$:

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

A função recebe como entrada um digrafo `D` (objeto `nx.DiGraph`) e o identificador `r0` da raiz. A implementação cria um novo digrafo vazio `F_star` (linha 2). Em seguida, para cada vértice `v` diferente de `r0` (linhas 3–4), coleta todas as arestas de entrada de `v` com seus pesos em uma lista e armazena na variável `in_edges` (linha 5). Se não houver arestas de entrada, prossegue para o próximo vértice (linhas 6–7). Caso contrário, utiliza uma compreensão de gerador para encontrar o primeiro predecessor `u` cuja aresta `(u, v)` tem peso zero (linha 8) e, se existir, adiciona essa aresta a `F_star` com peso zero (linhas 9–10).

A função 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. 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.

[4] No jargão de programação, "in-place"significa que a estrutura original é alterada diretamente, sem criar uma cópia. Isso economiza memória e tempo, mas introduz efeitos colaterais.

**Construção de F star**

*Constrói o subgrafo funcional F star a partir do grafo D e da raiz r0 entrando em r0.*

```
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
```

#### 1.3.2.1 Detecção de ciclo:

a função detecta um ciclo dirigido em $F^*$ (se existir) e retorna um subgrafo contendo o ciclo. Caso contrário, retorna None. A função utiliza a função `find_cycle` do NetworkX, que implementa um algoritmo eficiente de detecção de ciclos.

A função executa em $O(m)$. Isso ocorre porque a função `find_cycle` do NetworkX utiliza uma abordagem baseada em busca em profundidade (DFS) para detectar ciclos em grafos direcionados.

A complexidade dessa abordagem é linear em relação ao número de vértices e arestas do grafo, ou seja, $O(m)$, onde $m$ é o número de arestas. A função não modifica o grafo original, mas cria um subgrafo contendo apenas os vértices e arestas que fazem parte do ciclo detectado.

**Detecção de ciclo dirigido em $F^*$**

*Encontra um ciclo dirigido no grafo. Por fim, retorna um subgrafo contendo o ciclo, ou None caso não exista.*

```
1 % def find_cycle(F_star: nx.DiGraph):
2 %     try:
3 %         nodes_in_cycle = set()
4 %         # Extract nodes involved in the cycle
5 %         for u, v, _ in nx.find_cycle(F_star, orientation="original"):
6 %             nodes_in_cycle.update([u, v])
7 %         # Create a subgraph containing only the cycle
```

```
8 %         return F_star.subgraph(nodes_in_cycle).copy()
9
10 %     except nx.NetworkXNoCycle:
11 %         return None
```

#### 1.3.2.2 Contração de ciclo:

a função contrai um ciclo dirigido simples $C$ em um **supervértice** $x_C$, redirecionando arcos incidentes a $C$ e ajustando custos de acordo com a regra de *custos reduzidos*. O grafo é modificado *in-place* e a rotina devolve dicionários auxiliares para permitir a *reexpansão* correta do ciclo.

Em alto nível, o procedimento de contração de ciclo recebe como entrada um ciclo dirigido $C$ em $D$, a raiz $r_0$ (que não pertence a $C$), e um rótulo novo para o supervértice $x_C$. Para cada arco que entra em $C$, cria um arco para $x_C$ com custo ajustado; para cada arco que sai de $C$, redireciona a saída para partir de $x_C$; laços internos são descartados. O procedimento devolve dicionários que permitem reexpansão correta do ciclo ao final. Como efeito colateral, remove os vértices de $C$ e insere $x_C$ no grafo. Se não houver arco entrando em $C$, não existe r-arborescência; se não houver arco saindo, $x_C$ pode isolar componentes. O custo total é preservado e o procedimento é linear no número de arestas.

Essas escolhas garantem a *equivalência de custo* entre soluções ótimas no grafo contraído e no original após a reexpansão: os arcos internos de $C$ têm custo reduzido zero e apenas as entradas em $C$ recebem o desconto $y(w)$, mantendo a bijeção entre arborescências descrita anteriormente.

A expressão "no próprio lugar (inplace)" no docstring abaixo[5] indica que o grafo $D$ é modificado diretamente.

**Contração de ciclo**

*Contrai um ciclo C no grafo D, substituindo-o por um supervértice rotulado 'label'. Nesse processo, modifica o grafo D no próprio lugar (in-place) e por fim, devolve dicionários auxiliares para a reexpansão.*

```
1 % def contract_cycle(D: nx.DiGraph, C: nx.DiGraph, label: str):
2
```

[5] "Inplace" significa que a função altera diretamente a estrutura de dados existente, sem criar uma cópia. Assim, após a chamada, o grafo $D$ já refletirá as remoções, inserções e ajustes feitos. Isso reduz alocações e pode ser mais eficiente, mas exige cuidado com aliasing/referências ativas, pois o estado anterior não é preservado a menos que seja salvo explicitamente.

```
3
4 %     cycle_nodes: set[str] = set(C.nodes())
5
6 %     # Stores the vertex u outside the cycle and the vertex v inside the
    cycle that receives the minimum weight edge
7 %     in_to_cycle: dict[str, tuple[str, float]] = {}
8
9 %     for u in D.nodes:
10 %         if u not in cycle_nodes:
11 %             # Find the minimum weight edge that u has to any vertex in C
12 %             min_weight_edge_to_cycle = min(
13 %                 ((v, w) for _, v, w in D.out_edges(u, data="w") if v in
    cycle_nodes),
14 %                 key=lambda x: x[1],
15 %                 default=None,
16 %             )
17 %             if min_weight_edge_to_cycle:
18 %                 in_to_cycle[u] = min_weight_edge_to_cycle
19
20 %     for u, (v, w) in in_to_cycle.items():
21 %         D.add_edge(u, label, w=w)
22
23 %     # Stores the vertex v outside the cycle that receives the minimum
    weight edge from a vertex u inside the cycle
24 %     out_from_cycle: dict[str, tuple[str, float]] = {}
25
26 %     for v in D.nodes:
27 %         if v not in cycle_nodes:
28 %             # Find the minimum weight edge that v receives from any vertex
     in C
29 %             min_weight_edge_from_cycle = min(
30 %                 ((u, w) for u, _, w in D.in_edges(v, data="w") if u in
    cycle_nodes),
31 %                 key=lambda x: x[1],
32 %                 default=None,
33 %             )
34 %             if min_weight_edge_from_cycle:
35 %                 out_from_cycle[v] = min_weight_edge_from_cycle
36
37 %     for v, (u, w) in out_from_cycle.items():
```

```
38 %         D.add_edge(label, v, w=w)
39 %
40 %     # Remove all nodes in the cycle from G
41 %     D.remove_nodes_from(cycle_nodes)
42 %
43 %     return in_to_cycle, out_from_cycle
44 %
```

#### 1.3.2.3 Remoção de arestas que entram na raiz:

a função remove todas as arestas que entram no vértice raiz $r_0$ do grafo $G$. A função modifica o grafo *in-place* e executa em $O(\deg^-(r_0))$.

Isso ocorre porque a função obtém todas as arestas que entram em $r_0$ usando o método `in_edges` do NetworkX, que tem complexidade $O(\deg^-(r_0))$.

Em seguida, a função remove essas arestas usando o método `remove_edges_from`, que também opera em tempo linear em relação ao número de arestas sendo removidas. Portanto, o tempo total de execução da função é $O(\deg^-(r_0))$. A função não cria uma cópia do grafo original, mas altera diretamente a estrutura de dados do grafo fornecido.

**Remoção de arestas que entram na raiz**

*Remove todas as arestas que entram no vértice raiz r0 no grafo D. Por fim, retorna o grafo atualizado.*

```
1 % def remove_edges_to_r0(
2 %     D: nx.DiGraph, r0: str
3 % ):
4 %     # Remove all edges entering r0
5 %     in_edges = list(D.in_edges(r0))
6 %     if in_edges:
7 %         D.remove_edges_from(in_edges)
8 %     return D
```

#### 1.3.2.4 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.*

```
% def remove_internal_edge_to_cycle_entry(C: nx.DiGraph, v):

%     predecessor = next((u for u, _ in C.in_edges(v)), None)

%     C.remove_edge(predecessor, v)
```

#### 1.3.2.5 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.
- **Detecção 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 r0 usando o algoritmo de Chu–Liu/Edmonds.*

```
% def find_optimum_arborescence_chuliu(
%     D: nx.DiGraph,
%     r0: str,
%     level=0,
% ):

%     D_copy = D.copy()

```

```
 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
    edge from the arborescence
37 %         in_edge = next(iter(F_prime.in_edges(contracted_label, data="w")),
    None)
38 %
39 %         u, _, _ = in_edge
40 %
41 %         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)
       , the weight will be corrected at the end using G
49 %         F_prime.add_edge(u, v)
50
51 %         # Add the remaining edges of the modified cycle C
52 %         for u_c, v_c in C.edges:
53 %             F_prime.add_edge(u_c, v_c)
54
55 %         # Add the external edges leaving the cycle
56 %         for _, z, _ in F_prime.out_edges(contracted_label, data=True):
57
58 %             u_cycle, _ = out_from_cycle[z]
59 %             F_prime.add_edge(u_cycle, z)
60
61 %         F_prime.remove_node(contracted_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
```

#### 1.3.2.6 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

#### 1.3.2.7 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.

#### 1.3.2.8 Transição para a abordagem primal-dual

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

No capítulo seguinte, revisitaremos o mesmo problema sob uma ótica primal–dual em duas fases, proposta por András Frank. Essa perspectiva organiza a normalização via potenciais[6] $y(\cdot)$, explica os custos reduzidos e introduz a noção de cortes apertados (família laminar) como guias das contrações. Veremos como a mesma

[6] 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.

mecânica operacional (normalizar $\rightarrow$ contrair $\rightarrow$ expandir) emerge de condições duais que também sugerem otimizações e generalizações.

# Referências

KLEINBERG, J.; TARDOS, É. *Algorithm Design*. [S.l.]: Addison-Wesley, 2006. Citado 3 vezes nas páginas 11, 12 e 15.

SCHRIJVER, A. *Combinatorial Optimization: Polyhedra and Efficiency*. [S.l.]: Springer, 2003. Citado 4 vezes nas páginas 11, 12, 15 e 16.

# Anexos

# ANEXO A – Anexo A

Conteúdo do anexo A.