Lorena Silva Sampaio, Samira Haddad

Análise e Implementação de Algoritmos de Busca de uma r-Arborescência Inversa de Custo Mínimo em Grafos Dirigidos com Aplicação Didática Interativa

Brasil

2025

Lorena Silva Sampaio, Samira Haddad

Análise e Implementação de Algoritmos de Busca de uma r-Arborescência Inversa de Custo Mínimo em Grafos Dirigidos com Aplicação Didática Interativa

Dissertação apresentada à Universidade Federal do ABC como parte dos requisitos para a obtenção do título de Bacharel em Ciência da Computação.

Universidade Federal do ABC

Orientador: Prof. Dr. Mário Leston

Brasil
2025

Dedicatória (opcional).

Agradecimentos

Agradecimentos (opcional).

"Epígrafe (opcional)."

Resumo

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

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

Abstract

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

Keywords: Graphs. Arborescence. Algorithms. Visualization.

Lista de ilustrações

Sumário

1 Definições Preliminares

Neste capítulo, apresentamos os conceitos matemáticos fundamentais necessários para a compreensão dos algoritmos de arborescência de custo mínimo. Focamos exclusivamente nas definições que serão utilizadas nos capítulos subsequentes, estabelecendo a notação e a terminologia básica de grafos direcionados, arborescências e otimização.

1.1 Grafos Direcionados

Um grafo dirigido (ou *digrafo*) $D = (V, A)$ é composto por um conjunto V de *vértices* e um conjunto A de *arcos* (arestas direcionadas). Cada arco $(u, v) \in A$ conecta o vértice u (cauda) ao vértice v (cabeça).

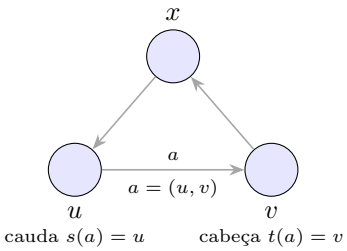


Figura 1 – Digrafo: arcos têm direção. No arco $a = (u, v)$, u é a cauda e v é a cabeça.

Graus, Trilhas e Circuitos

O grau de entrada de um vértice v, denotado $d^-(v)$, é o número de arcos que chegam a v. O grau de saída $d^+(v)$ é o número de arcos que saem de v.

Uma trilha (caminho direcionado) é uma sequência de vértices $P = (v_1, v_2, \ldots, v_{k+1})$ tal que cada par consecutivo (v_i, v_{i+1}) é um arco em A.

Um circuito (ciclo direcionado) é uma trilha que começa e termina no mesmo vértice: $v_1 = v_{k+1}$.

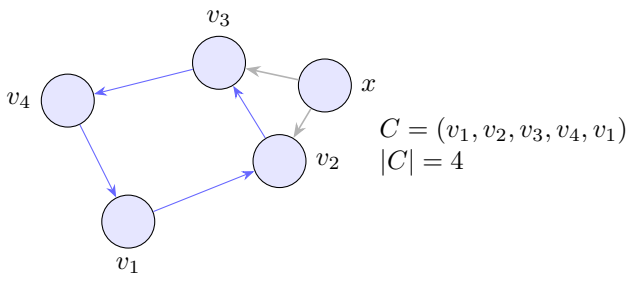
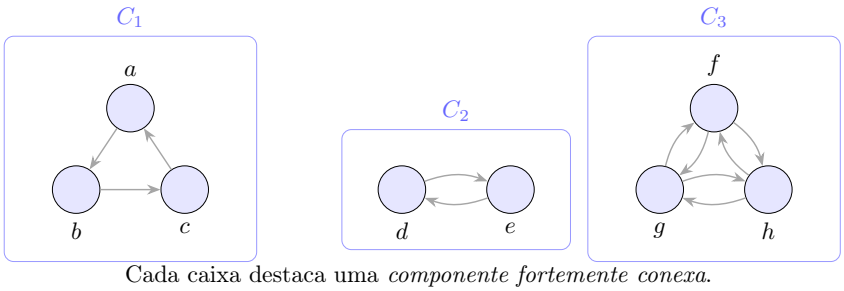


Figura 2 – Ciclo direcionado: o ciclo $C = (v_1, v_2, v_3, v_4, v_1)$ está destacado.

Componentes fortemente conexas

Uma componente fortemente conexa (CFC) é um subdigrafo maximal onde existe um caminho direcionado entre qualquer par de vértices. Dois vértices u e v pertencem à mesma CFC se, e somente se, existe caminho de u para v e de v para u.

Ao contrair cada CFC a um único vértice, obtemos o grafo condensado, que é sempre um DAG (grafo acíclico dirigido).



Cada caixa destaca uma *componente fortemente conexa*.
Não há arcos entre C_1, C_2 e C_3.

Figura 3 – Componentes fortemente conexas: cada C_i é fortemente conexo e maximal.

1.2 Arborescências

Uma arborescência enraizada em r é um digrafo acíclico onde existe exatamente um caminho direcionado de r para cada vértice $v \in V \setminus \{r\}$. Equivalentemente, a raiz r tem grau de entrada zero ($d^-(r) = 0$), todo outro vértice tem grau de entrada um ($d^-(v) = 1$ para $v \neq r$) e o número de arcos é $|A| = |V| - 1$.

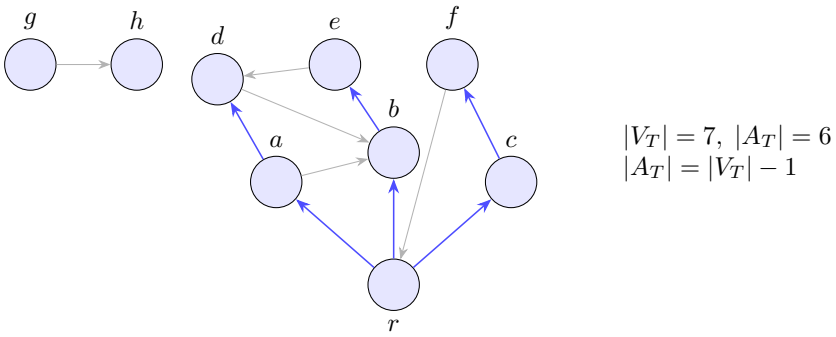
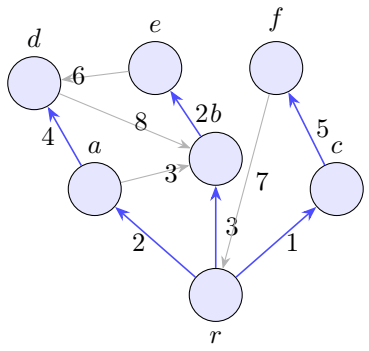


Figura 4 – Arborescência enraizada em r: caminho único da raiz para cada vértice.

Arborescência de custo mínimo

Dado um digrafo $D = (V, A)$ com função de custo $c : A \to \mathbb{R}_{\geq 0}$ e raiz $r \in V$, uma arborescência de custo mínimo é uma arborescência enraizada em r que minimiza o custo total:

$$C(T) = \sum_{a \in A_T} c(a)$$



Custo total da r-arborescência: $2 + 3 + 1 + 4 + 2 + 5 = 17$

Figura 5 – Arborescência de custo mínimo enraizada em r com custo total 17.

1.3 Condições de Otimalidade

Para um conjunto $X \subseteq V \setminus \{r\}$, definimos $\delta^-(X)$ como o conjunto de arcos que entram em X vindos de fora:

$$\delta^-(X) = \{(u, v) \in A : u \notin X, v \in X\}$$

Dada uma família de pesos $y(X) \geq 0$ para cada subconjunto não vazio, o custo reduzido de um arco $a = (u, v)$ é:

$$c'(a) = c(a) - \sum_{\substack{X \subseteq V \setminus \{r\}, \\ u \notin X,\ v \in X}} y(X)$$

Teorema de Fulkerson

Uma arborescência T enraizada em r tem custo mínimo se, e somente se, existem pesos $y(X) \geq 0$ tais que:

1. $c'(a) \geq 0$ para todo arco $a \in A$
2. $c'(a) = 0$ para todo arco $a \in T$
3. Para todo X com $y(X) > 0$, exatamente um arco de T entra em X

Este teorema fornece um certificado de otimalidade verificável: os custos reduzidos conectam a solução primal (arcos escolhidos) ao dual (pesos dos subconjuntos).

1.4 Complexidade Computacional

Expressamos a complexidade de algoritmos usando notação Big-O: $O(f(n))$ denota limite superior (o tempo cresce no máximo proporcional a $f(n)$), $\Omega(f(n))$ denota

limite inferior (o tempo cresce no mínimo proporcional a $f(n)$) e $\Theta(f(n))$ denota limite preciso (o tempo cresce exatamente proporcional a $f(n)$).

Para o problema da arborescência de custo mínimo, o algoritmo de Chu–Liu–Edmonds tem complexidade $O(mn)$ na versão básica e $O(m \log n)$ com heaps, enquanto o algoritmo de András Frank tem complexidade $O(mn)$ na versão básica e $O(m \log n)$ com heaps na Fase II, onde $n = |V|$ e $m = |A|$.

Algoritmos gulosos

Um algoritmo guloso toma decisões locais ótimas em cada etapa, esperando que levem a uma solução globalmente ótima. Suas características incluem escolha local baseada em critério de otimização imediato, decisões definitivas (não reconsidera escolhas anteriores) e eficiência, embora nem sempre garanta otimalidade global.

Os algoritmos de Chu–Liu–Edmonds e Frank são gulosos: selecionam arcos de menor custo (reduzido) e ajustam a estrutura iterativamente até obter a arborescência ótima.

1.5 O problema da Arborescência de Custo Mínimo

O problema da arborescência de custo mínimo é um problema clássico em otimização combinatória e teoria dos grafos. O problema consiste em dado um digrafo enraizado $D = (V, A)$ com custos $c : A \to \mathbb{R}$ nos arcos e um vértice raiz $r \in V$, o objetivo é encontrar uma arborescência de r que minimize a soma dos custos dos arcos selecionados.

Este problema pode ser visto como uma generalização do problema da árvore geradora mínima para digrafos, mas apresenta características distintas devido à natureza direcionada dos arcos.

Dois algoritmos fundamentais resolvem eficientemente o problema da arborescência de custo mínimo: o algoritmo de Chu–Liu–Edmonds e o algoritmo de András Frank.

Algoritmo de Chu–Liu–Edmonds

Proposto independentemente por Chu e Liu (CHU; LIU, 1965) e por Edmonds (EDMONDS, 1967), este algoritmo opera iterativamente selecionando para cada vértice $v \neq r$ um arco de entrada de custo mínimo. Quando ciclos são detectados, o algoritmo contrai o ciclo em um único vértice, ajusta os custos apropriadamente e resolve recursivamente o problema no grafo contraído. Ao final, expande os ciclos contraídos para reconstruir a arborescência ótima no grafo original.

Algoritmo de András Frank

O algoritmo de Frank (FRANK, 1981; FRANK; HAJDU, 2014) adota uma abordagem diferente, operando em duas fases distintas. A Fase I identifica *subconjuntos minimais* via componentes fortemente conexas, reduzindo custos até criar arcos de custo reduzido zero. A Fase II constrói a arborescência a partir desses arcos, garantindo otimalidade.

Esta abordagem processa múltiplos vértices simultaneamente, explorando a estrutura do grafo de forma mais global. A complexidade temporal também é $O(mn)$, com possibilidade de melhoria para $O(m \log n)$ usando filas de prioridade na Fase II.

Embora ambos os algoritmos produzam soluções ótimas, diferem significativamente em sua organização interna. Chu–Liu–Edmonds opera vértice a vértice com contrações imediatas de ciclos, enquanto Frank identifica subconjuntos minimais processando múltiplos vértices simultaneamente.

Ambos contraem ciclos detectados e expandem ao final, com número de contrações limitado por $O(n)$ (SCHRIJVER, 2003). Esta limitação é fundamental para garantir a eficiência computacional dos métodos.

Os capítulos seguintes apresentam implementações detalhadas de ambos os algoritmos, comparando seus desempenhos experimentais e desenvolvendo ferramentas didáticas interativas para facilitar a compreensão de suas operações.

2 Algoritmo de Chu–Liu–Edmonds

Neste capítulo, apresentaremos o algoritmo de Chu–Liu–Edmonds, que determina uma arborescência de custo mínimo em um digrafo ponderado. O algoritmo baseia-se em duas operações fundamentais: (i) a redução gulosa dos custos dos arcos e (ii) a contração de ciclos, de modo a resolver recursivamente uma instância menor do problema e, em seguida, estender a solução para o problema original. O propósito deste capítulo é fornecer uma descrição precisa tanto do algoritmo quanto da implementação desenvolvida neste trabalho.

2.1 O algoritmo

O algoritmo de Chu–Liu–Edmonds recebe uma tripla (D, c, r), em que $D = (V, A)$ é um digrafo, $c\colon A \to \mathbb{R}$ é uma função custo e $r \in V$ é a raiz, sob a hipótese de que D admite ao menos uma r-arborescência. O algoritmo devolve uma r-arborescência c-mínima de D.

Para evitar repetir essa hipótese, introduzimos a seguinte definição. Uma tripla (D, c, r) é um **r-digrafo ponderado** se (D, c) é um digrafo ponderado, r é um vértice de D, $\delta^-(r) = \varnothing$ e D possui uma r-arborescência. Note que a hipótese $\delta^-(r)$ é uma trivialidade, pois uma r-arborescência não contém nenhum arco que entra em r e, portanto, tais arcos podem ser eliminados de D sem nenhum prejuízo.

Vamos tecer algumas considerações para motivar o algoritmo.

Caráter Guloso

Suponha doravante que (D, c, r) é um r-digrafo ponderado. O algoritmo tem um caráter guloso. Note que, se T é uma r-arborescência de D, então, para cada vértice $v \neq r$, existe exatamente um arco de T que entra em v. Isso sugere a seguinte escolha gulosa: para cada vértice $v \neq r$, selecione um arco a_v de custo mínimo que entra em v e forme o conjunto $T := \{a_v : v \in V \setminus \{r\}\}$.

Suponha que T é uma r-arborescência. Não é difícil verificar que T tem custo mínimo. De fato, seja F uma r-arborescência de D. Para cada vértice $v \neq r$, escreva b_v para o *único* arco de F que entra em v. Pela escolha gulosa,

$$c(a_v) \leq c(b_v) \quad \text{para todo } v \neq r.$$

Logo,

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

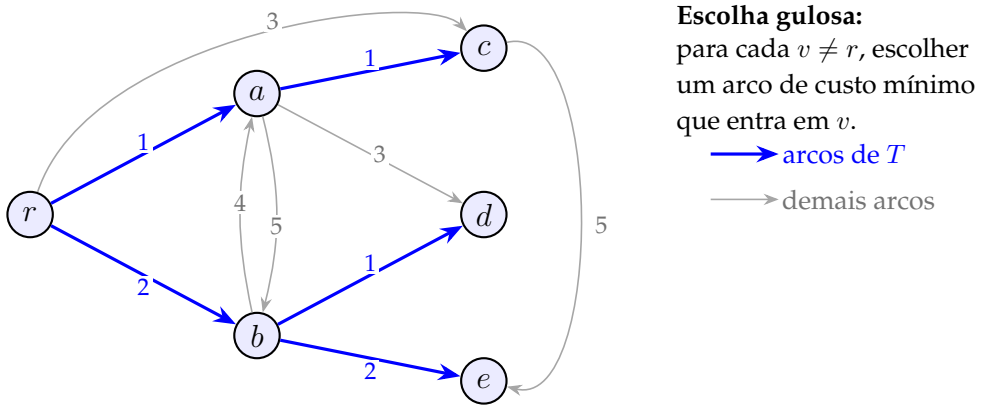


Figura 6 – A figura ilustra a escolha gulosa quando esta produz uma r-arborescência. Os arcos em azul são os escolhidos; os cinza são os demais arcos do digrafo.

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

A seguinte figura ilustra que podemos não ter tanta sorte.

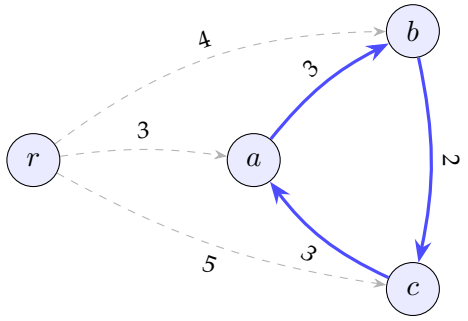


Figura 7 – Os arcos azuis são os da escolha gulosa.

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

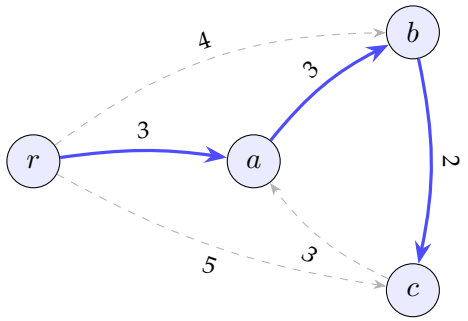


Figura 8 – Os arcos azuis são os da escolha gulosa.

O exemplo acima sugere que devemos formar o subdigrafo H de D com $V(H) = V(D)$ e

$$A(H) := \bigcup_{v \in V \setminus \{r\}} \arg\min\{\, c(a) : a \in \delta^-(v) \,\}.$$

Ou seja, para cada $v \neq r$ incluímos em H *todos* os arcos de custo mínimo que entram em v. Um argumento análogo ao anterior mostra que, se H contém uma r-arborescência, então ela é de custo mínimo.

Infelizmente, só isso não é suficiente, como mostra a próxima figura.

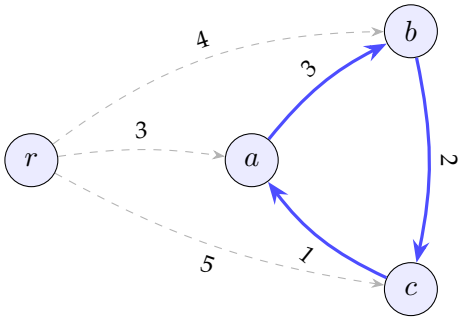


Figura 9 – Os arcos azuis são os da escolha gulosa.

O ideal, do ponto de vista algorítmico, é dispor de uma forma simples de identificar o subdigrafo H. Uma transformação nos custos fornece exatamente isso. Para tanto, introduzimos a noção de custo q-reduzido.

Seja $q : V \setminus \{r\} \to \mathbb{R}$ (convencionamos $q(r) = 0$). Definimos o custo q-reduzido $c_q : A \to \mathbb{R}$ por

$$c_q(a) := c(a) - q\big(\mathrm{head}(a)\big), \qquad a \in A.$$

Para um conjunto $X \subseteq V$, escrevemos $q(X) := \sum_{u \in X} q(u)$.

A próxima proposição mostra que a transformação por custo q-reduzido preserva a otimalidade.

Proposição 2.1. *Para toda função $q : V \setminus \{r\} \to \mathbb{R}$, uma r-arborescência T é c-mínima em D se, e somente se, T é c_q-mínima em D.*

Prova. Seja F uma r-arborescência. Para cada $u \in V \setminus \{r\}$, seja a_u o único arco de F que entra em u. Então

$$\begin{aligned} c_q(F) &= \sum_{u \in V \setminus \{r\}} c_q(a_u) \\ &= \sum_{u \in V \setminus \{r\}} \big(c(a_u) - q(u)\big) \\ &= \sum_{u \in V \setminus \{r\}} c(a_u) - \sum_{u \in V \setminus \{r\}} q(u) \\ &= c(F) - q(V \setminus \{r\}). \end{aligned}$$

Assim, para quaisquer r-arborescências T e F,

$$c(T) \leq c(F) \iff c'(T) = c(T) - q(V \setminus \{r\}) \leq c(F) - q(V \setminus \{r\}) = c_q(F),$$

o que prova a proposição. □

Para cada $v \in V \setminus \{r\}$, defina

$$\lambda(v) := \lambda_c(v) := \min\{\, c(a) : a \in \delta^-(v) \,\}.$$

Note que λ está bem definida uma vez que D possui uma r-arborescência e, portanto, existe ao menos um arco que entra em cada vértice diferente de r. Então, para todo $v \in V \setminus \{r\}$,

$$\min\{\, c_\lambda(a) : a \in \delta^-(v) \,\} = 0,$$

isto é, precisamente os arcos de custo mínimo que entram em v passam a ter custo zero, e os demais ficam com custo positivo. Consequentemente, o subdigrafo H obtém-se simplesmente como o subdigrafo induzido pelos arcos de custo zero de c_λ:

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

A figura a seguir ilustra a redução de custos no digrafo da Figura 9.

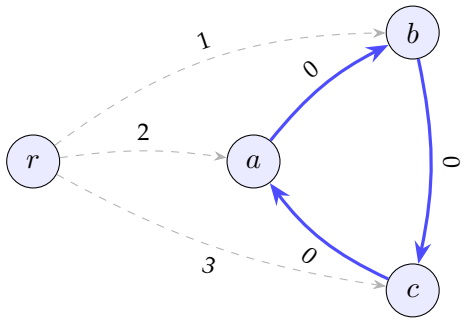


Figura 10 – Os arcos azuis são os da escolha gulosa.

Podemos agora retomar o caso no qual o subdigrafo gerador H de D, cujos arcos são aqueles em que o custo λ-reduzido é zero, não possui uma r-arborescência. Vamos mostrar que H possui um ciclo.

Seja $v \neq r$ um vértice de V que *não* é alcançável a partir de r em H. Considere um caminho simples maximal[1] de H que termina em v. Seja u o início de P. Como v não é atingível a partir de r, temos que $u \neq r$. Logo, existe exatamente um arco, digamos wv, de H que entra em u. Pela maximalidade de P, o vértice w é um dos vértices de P (caso contrário, $w \cdot P$ é um caminho simples, o que contraria a escolha de P). Como P é um caminho simples que começa em u, o vértice w aparece em P *após* u; portanto, P contém um subcaminho S de u até w. Consequentemente, $S \cdot u$ é um ciclo de H.

A solução consiste em *normalizar os custos por vértice*: para cada $v \neq r$, subtraímos de todo arco que entra em v o menor custo entre os arcos que chegam a v. Após esse

[1] Maximal aqui tem o seguinte sentido. Para cada vértice u de H, as sequencias $P \cdot u$ e $u \cdot P$ não são caminhos simples.

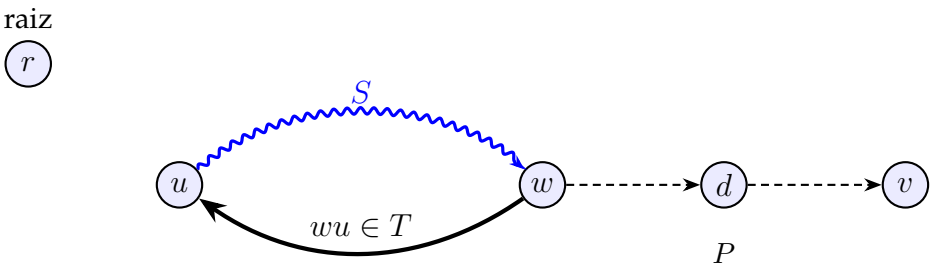


Figura 11 – O caminho simples maximal P inicia em u e termina em v. A porção S de P entre u e w é indicada pelo arco ondulado azul; o caminho $S \cdot u$ é um ciclo.

ajuste (custos reduzidos), cada $v \neq r$ passa a ter ao menos um arco de custo reduzido zero. Se os arcos de custo zero forem acíclicos, já temos a r-arborescência ótima. Se formarem um ciclo C, *contraímos* C em um supervértice x_C, ajustamos os custos dos arcos externos e resolvemos recursivamente no digrafo menor.

A seguir, detalhamos essa operação de contração de ciclos.

Contração de ciclos

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

$$V' := (V \setminus C) \cup \{x_C\}.$$

O conjunto de arcos A' é construído a partir de A da seguinte forma: para cada arco $a = (u, v) \in A$, mantemos a inalterado em A' se ambos u e v estão fora de C; descartamos a se ambos pertencem a C; criamos um arco (u, x_C) se $u \notin C$ e $v \in C$; e criamos um arco (x_C, v) se $u \in C$ e $v \notin C$.

Ajustamos os custos dos arcos que entram e saem do supervértice x_C em D' para refletir a contração do ciclo C da seguinte forma: para cada arco (u, v) com $u \notin C$ e $v \in C$, o custo do arco contraído (u, x_C) é definido como $c_\lambda(u, v)$, onde $\lambda(v) = \min\{c(a) : a \in \delta^-(v)\}$ é o custo mínimo de entrada em v e de forma semelhante, para cada arco (u, v) com $u \in C$ e $v \notin C$, o custo do arco contraído (x_C, v) é definido como $c_\lambda(u, v)$, onde $c_\lambda(u, v) = c(u, v) - \lambda(v)$ e $\lambda(v) = \min\{c(a) : a \in \delta^-(v)\}$ é o custo mínimo de entrada em v.

Agora vamos ilustrar um exemplo de como essa contração é feita e os custos são ajustados.

Considere o digrafo D a seguir, com o ciclo $C = (v_1, v_2, v_3, v_1)$.

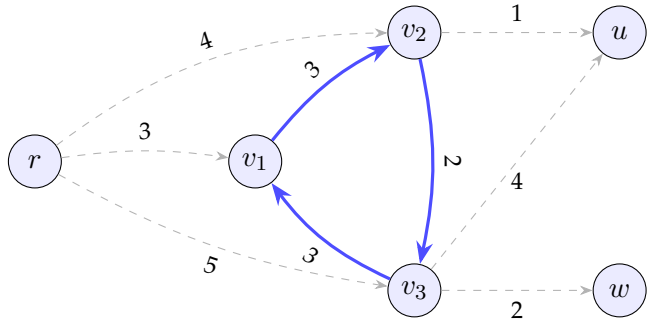


Figura 12 – O digrafo D com o ciclo $C = (v_1, v_2, v_3, v_1)$. Os arcos azuis representam os arcos do ciclo, os tracejados representam os demais arcos do digrafo.

Após a normalização dos custos, os arcos internos do ciclo passam a ter custo reduzido zero e os demais arcos são ajustados conforme a definição de custo λ-reduzido:

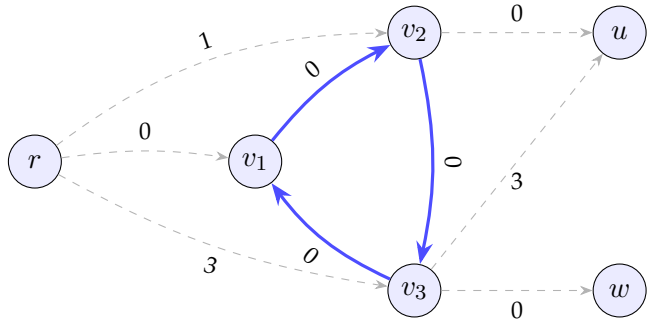


Figura 13 – digrafo com custos λ-reduzidos. Os arcos internos do ciclo C têm custo zero (em azul). Os arcos da raiz para o ciclo têm custos 1, 0 e 3 (tracejados).

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

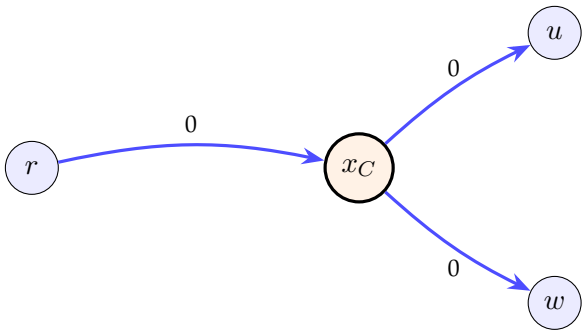


Figura 14 – digrafo D' após a contração do ciclo C. O supervértice x_C substitui todos os vértices do ciclo. Originalmente, havia três arcos paralelos de r para o ciclo: (r, v_1), (r, v_2) e (r, v_3) com custos reduzidos 1, 0 e 3; mantemos apenas o de menor custo 0. Os arcos que saíam do ciclo agora saem de x_C: (x_C, u) com custo 0 e (x_C, w) com custo 0. Note que havia dois arcos de vértices do ciclo para u; mantemos apenas o de menor custo.

Por definição não admitimos gerar arcos paralelos entre um mesmo par de vértices, mantemos apenas o arco de menor custo, conforme ilustrado do vértice r para

o supervértice x_C e de x_C para u e isso não afeta a otimalidade, já que na reexpansão qualquer escolha entre arcos paralelos conduz à mesma solução ótima.

Reexpansão de arborescências

Após resolver o problema no digrafo contraído D', obtemos uma r-arborescência ótima T' em D'. Para reexpandir T' em uma r-arborescência T em D, substituímos o supervértice x_C pelo ciclo C e adicionamos os arcos do ciclo que formam a arborescência dentro de C. Especificamente, se o arco (u, x_C) em T' corresponde a um arco (u, v_i) em D (onde $v_i \in C$), então incluímos esse arco em T. Em seguida, adicionamos os arcos do ciclo C que conectam os vértices de C de forma a manter a estrutura de arborescência. Note que, devemos escolher todos os arcos do ciclo C exceto aquele que entra em v_i, garantindo que cada vértice de C tenha grau de entrada igual a 1, exceto v_i.

Seguindo nosso exemplo anterior, ilustramos a reexpansão da r-arborescência, primeiramente adicionando novamente os vértices que pertenciam ao ciclo C e, em seguida, incluindo os arcos apropriados para formar a r-arborescência T em D:

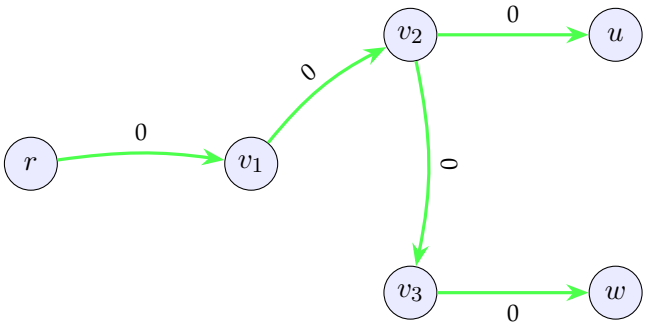


Figura 15 – Reexpansão da r-arborescência ótima T' em D' para obter a r-arborescência T em D

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

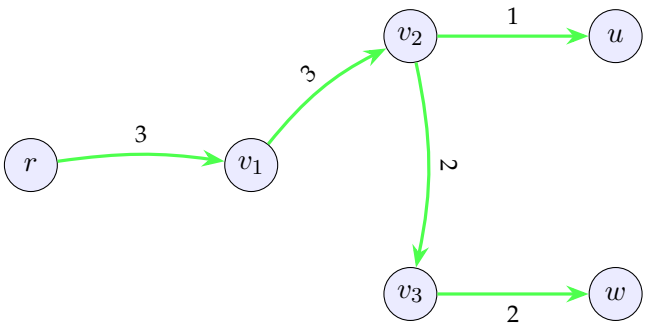


Figura 16 – Reexpansão da r-arborescência ótima T' em D' para obter a r-arborescência T em D. Os arcos selecionados em verde fazem parte de T.

É importante observar que, ao depender da forma com a qual extraímos a arborescência ótima de D' a partir do nosso diagrafo original, podemos obter múltiplas arborescências ótimas em D, o exemplo a seguir ilustra essa situação:

Considere o digrafo a seguir com custos originais:

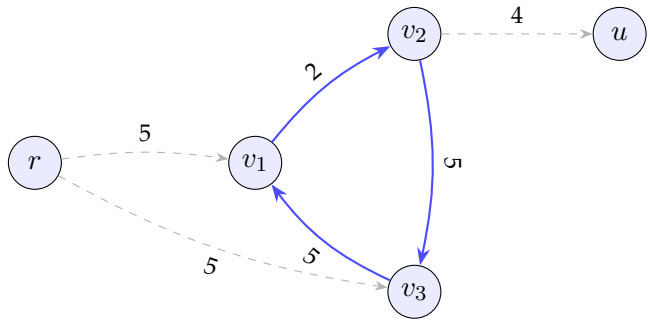


Figura 17 – digrafo D com custos originais. O ciclo $C = (v_1, v_2, v_3, v_1)$ tem arcos com custos 5, 5 e 2. Existem dois arcos da raiz para o ciclo, ambos com custo 5: (r, v_1) e (r, v_3).

Após a normalização dos custos (subtraindo $\lambda(v_1) = 5$, $\lambda(v_2) = 2$, $\lambda(v_3) = 5$ e $\lambda(u) = 4$), obtemos:

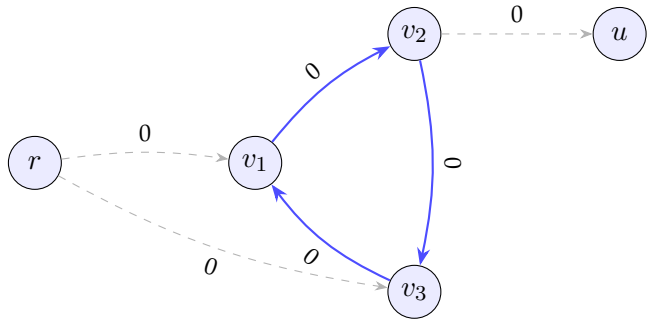


Figura 18 – digrafo D com custos λ-reduzidos. Todos os arcos do ciclo custo mínimo têm agora custo zero.

Após a contração do ciclo C, ambas as arborescências T_1 e T_2 mapeiam para a mesma arborescência T' no digrafo contraído D':

Arborescência T' em D'

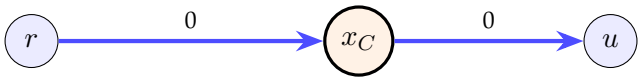


Figura 19 – Arborescência T' no digrafo contraído D'. O arco (r, x_C) pode corresponder ou ao arco (r, v_1) ou ao (r, v_3) em D e o arco (x_C, u) corresponde ao arco normalizado (v_2, u) em D.

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

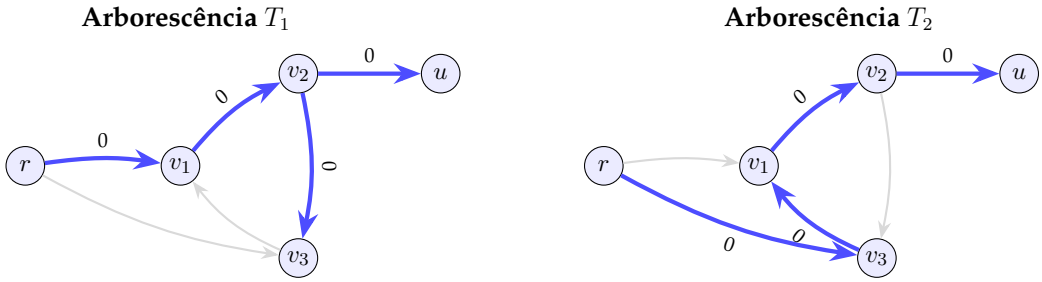


Figura 20 – Duas r-arborescências ótimas distintas em D com custos c_λ-reduzidos. T_1 usa o arco (r, v_1) e os arcos do ciclo (v_1, v_2) e (v_2, v_3). T_2 usa o arco (r, v_3) e os arcos do ciclo (v_3, v_1) e (v_1, v_2). Ambas incluem o arco (v_2, u) e têm custo total zero.

Assim, vemos que a correspondência entre as r-arborescências de D e D' não é bijetiva, pois duas arborescências distintas em D podem corresponder à mesma arborescência em D'. Isso ocorre porque ambos os arcos (r, v_1) e (r, v_3) têm o mesmo custo λ-reduzido (zero) e ambos entram no ciclo C; após a contração, ambos são representados pelo único arco (r, x_C) no digrafo contraído.

A seguir, apresentamos a proposição que estabelece a correspondência entre as r-arborescências de D e D' após a contração do ciclo C.

Proposição 2.2. *Seja C um ciclo dirigido em D com $r \notin C$, e seja D' o digrafo obtido pela contração de C. Para cada vértice $v \in C$, seja a_v o único arco de C que entra em v, e suponha que $c_\lambda(a_v) = 0$ para todo $v \in C$, onde $\lambda(v) = \min\{c(a) : a \in \delta^-(v)\}$. Defina os custos $c' : A' \to \mathbb{R}$ por*

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

Então existe uma correspondência bijetiva entre as r-arborescências de D' com custos c' e as r-arborescências de D com custos c_λ que contêm exatamente um arco entrando em C. Note que, em geral, podem haver múltiplas arborescências ótimas de D que são mapeadas para uma mesma arborescência ótima de D', ou seja, a correspondência entre arborescências ótimas pode não ser bijetiva.

Prova. Seja T' uma r-arborescência de D'. Como x_C é um vértice de D' e $x_C \neq r$, existe exatamente um arco de T' que entra em x_C. Seja (u, x_C) esse arco. No digrafo original D, o arco (u, x_C) corresponde a um arco (u, w) para algum $w \in C$.

Definimos $T \subseteq A$ da seguinte forma: para cada arco $a' \in T'$ que não envolve x_C, incluímos o arco correspondente $a \in A$ em T; para o arco $(u, x_C) \in T'$, incluímos

$(u, w) \in A$ em T; e incluímos todos os arcos de C, com exceção do arco a_w que entra em w.

Afirmamos que T é uma r-arborescência de D. De fato, para cada vértice $v \in V \setminus \{r\}$, se $v \notin C$, então $v \in V'$ e há exatamente um arco de T' entrando em v, logo há exatamente um arco de T entrando em v. Se $v \in C$ e $v \neq w$, então o único arco de T entrando em v é o arco a_v do ciclo C. Finalmente, se $v = w$, o único arco de T entrando em w é precisamente (u, w).

Além disso, como T' é acíclico em D' e os arcos do ciclo C formam um caminho de w até seus predecessores em C (exceto o arco removido a_w), o conjunto T permanece acíclico. Portanto, T é uma r-arborescência de D.

Reciprocamente, seja T uma r-arborescência de D que contém exatamente um arco entrando em C, digamos (u, w) com $u \notin C$ e $w \in C$. Definimos $T' \subseteq A'$ mantendo cada arco de T que não envolve vértices de C, substituindo o arco (u, w) por (u, x_C), e descartando os arcos internos de C presentes em T. É direto verificar que T' é uma r-arborescência de D' e que essa correspondência é bijetiva.

Finalmente, como todos os arcos a_v do ciclo C têm custo c_λ-reduzido zero, temos

$$c_\lambda(T) \;=\; \sum_{a \in T \setminus C} c_\lambda(a) + c_\lambda(u, w) \;=\; c'(T'),$$

o que estabelece a correspondência entre custos. □

Essa proposição justifica a estratégia recursiva do algoritmo: resolver o problema no digrafo contraído D' com custos ajustados c' e, em seguida, expandir a solução para o digrafo original D.

A seguir, apresentamos a implementação completa do algoritmo de Chu–Liu e Edmonds para encontrar uma r-arborescência de custo mínimo em um r-digrafo ponderado (D, c, r).

2.2 Descrição do algoritmo

A seguir apresentamos uma descrição formal do algoritmo de Chu–Liu/Edmonds. Detalhes de implementação serão discutidos na próxima seção.

Algoritmo 2.1: Chu–Liu/Edmonds

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 $A_0 := \{a_v : v \neq r\}$.

2. Se (V, A_0) é acíclico, devolva A_0. Por (KLEINBERG; TARDOS, 2006, Obs. 4.36), trata-se de uma r-arborescência de custo mínimo.

3. Caso contrário, seja C um ciclo dirigido de A_0 (com $r \notin C$). **Contração:** contraia C em um supervértice x_C e defina custos c' por

$$\begin{aligned} c'(u, x_C) &:= c(u, w) - y(w) = c(u, w) - c(a_w) && \text{para } u \notin C,\ w \in C, \\ c'(x_C, v) &:= c(w, v) && \text{para } w \in C,\ v \notin C, \end{aligned}$$

descartando laços em x_C e permitindo paralelos. Denote o digrafo contraído por $D' = (V', A')$.

4. **Recursão:** compute uma r-arborescência ótima T' de D' com custos c'.

5. **Expansão:** seja $(u, x_C) \in T'$ o único arco que entra em x_C. No digrafo original, ele corresponde a (u, w) com $w \in C$. Forme

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

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.

2.2.1 Corretude

A corretude do algoritmo de Chu–Liu/Edmonds baseia-se em três pilares principais:

1. *Normalização por custos reduzidos:* para cada $v \neq r$, defina $y(v) := \min\{c(u, v) : (u, v) \in A\}$ e $c'(u, v) := c(u, v) - y(v)$. Para qualquer r-arborescência T, vale

$$\sum_{a \in T} c'(a) \;=\; \sum_{a \in T} c(a) \;-\; \sum_{v \neq r} y(v),$$

pois há exatamente um arco de T entrando em cada $v \neq r$. O termo $\sum_{v \neq r} y(v)$ é constante (independe de T); assim, minimizar $\sum c$ equivale a minimizar $\sum c'$ (KLEINBERG; TARDOS, 2006, Obs. 4.37). Em particular, os arcos a_v de menor custo que entram em v têm custo reduzido zero e formam A_0.

2. *Caso acíclico:* se (V, A_0) é acíclico, então já é uma r-arborescência e, por realizar o mínimo custo de entrada em cada $v \neq r$, é ótima (KLEINBERG; TARDOS, 2006, Obs. 4.36).

3. *Caso com ciclo (contração/expansão):* se A_0 contém um ciclo dirigido C, todos os seus arcos têm custo reduzido zero.

   Contraia C em x_C e ajuste apenas arcos que *entram* em C: $c'(u, x_C) := c(u, w) - y(w) = c(u, w) - c(a_w)$.

   Resolva o problema no digrafo contraído D', obtendo uma r-arborescência ótima T' sob c'. Na expansão, substitua o arco $(u, x_C) \in T'$ pelo correspondente (u, w) (com $w \in C$) e remova a_w de C.

   Considerando que os arcos de C têm custo reduzido zero e $c'(u, x_C) = c(u, w) - y(w)$, a soma dos custos reduzidos é preservada na ida e na volta; logo, T' ótimo em D' mapeia para T ótimo em D para c'. Pela equivalência entre c e c', T também é ótimo para c. Repetindo o argumento a cada contração, obtemos a corretude por indução (KLEINBERG; TARDOS, 2006; SCHRIJVER, 2003, Sec. 4.9).

Em termos intuitivos, y funciona como um potencial nos vértices: torna "apertados" (custo reduzido zero) os candidatos corretos; ciclos de arcos apertados podem ser contraídos sem perder otimalidade.

2.2.2 Complexidade

Na implementação direta, selecionar os a_v, detectar/contrair ciclos e atualizar estruturas custa $O(m)$ por nível; como o número de vértices decresce a cada contração, temos no máximo $O(n)$ níveis e tempo total $O(mn)$, com $n = |V|$, $m = |A|$.

O uso de memória é $O(m + n)$, incluindo mapeamentos de contração/expansão e as filas de prioridade dos arcos de entrada. A implementação a seguir adota a versão $O(mn)$ por simplicidade e está disponível no repositório do projeto (<https://github.com/lorenypsum/GraphVisualizer>).

2.3 Implementação em Python

Esta seção descreve a implementação do algoritmo de Chu–Liu–Edmonds em Python, estruturada para refletir com precisão as etapas formais discutidas anteriormente. Cada operação fundamental — normalização dos custos, construção do subdigrafo gerador, contração de ciclos e reexpansão — é traduzida em procedimentos sobre digrafos orientados, utilizando a biblioteca `networkx`.

A entrada consiste em um digrafo orientado $D = (V, A)$, com custos dos arcos registrados no atributo `"w"`, e uma raiz $r \in V$. As hipóteses adotadas são: (i) o digrafo é conexo a partir de r, isto é, todo vértice $v \neq r$ é alcançável a partir da raiz; (ii) para todo subconjunto $X \subseteq V \setminus \{r\}$, existe ao menos um arco entrando em X (condições de Edmonds, cf. (SCHRIJVER, 2003)); e (iii) todos os custos são não negativos.

A saída é um subdigrafo T de D com $|A_T| = |V| - 1$ arcos, tal que cada vértice $v \neq r$ possui grau de entrada igual a 1, todos os vértices são alcançáveis a partir de r, e o custo total $\sum_{a \in A_T} c(a)$ é mínimo.

Por limitações da representação com `networkx.DiGraph`, a implementação elimina arcos paralelos durante a contração de ciclos.

A estrutura do código é modular: funções auxiliares tratam cada etapa do algoritmo — normalização dos custos, detecção e contração de ciclos, construção do subdigrafo gerador e reexpansão da solução. Todas operam sobre objetos `nx.DiGraph` e são coordenadas por uma função principal que gerencia o fluxo recursivo. As subseções seguintes detalham cada função auxiliar, abordando lógica, parâmetros, saídas e complexidade.

2.3.1 Representação de digrafos e detecção de ciclos

A implementação utiliza a biblioteca NetworkX[2], especificamente a classe `nx.DiGraph` para representar digrafos $D = (V, A)$. Internamente, usa dicionários aninhados do Python para armazenar vértices, arcos e atributos, garantindo operações eficientes: adicionar/remover arco $O(1)$ amortizado, iterar vizinhos $O(\deg(u))$, percorrer todos os arcos $O(m)$.

Métodos da API NetworkX

Os métodos da API NetworkX utilizados na implementação dividem-se em três categorias funcionais, cada uma correspondendo a uma fase específica do algoritmo:

Consulta de estrutura

- `D.nodes()`: devolve visão iterável sobre V, permitindo percorrer todos os vértices.
- `D.in_edges(v, data="w")`: devolve arcos entrantes em v com pesos, produzindo tuplas (u, v, w).
- `D.out_edges(u, data="w")`: devolve arcos saíntes de u com pesos, análogo a `in_edges`.
- `D[u][v]["w"]`: acessa diretamente o peso do arco (u, v) para leitura ou modificação.

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

Modificação de estrutura

- `D.add_edge(u, v, w=peso)`: adiciona arco (u, v) com peso especificado, criando vértices automaticamente se não existirem.
- `D.remove_edges_from(edges)`: remove múltiplos arcos em lote.
- `D.remove_nodes_from(nodes)`: remove vértices e todos os seus arcos incidentes.

2.3.2 Remoção de arcos que entram na raiz:

Escrevemos essa função como uma etapa de pré-processamento para garantir que a raiz r_0 não possua arcos de entrada antes de iniciar o algoritmo principal.

A remoção é necessária porque, por definição, uma r-arborescência é uma arborescência enraizada em r_0 onde todo vértice $v \neq r_0$ deve ser alcançável a partir de r_0, mas a própria raiz não pode ter predecessores (grau de entrada zero). Se o digrafo original contiver arcos entrando em r_0, esses arcos violariam a definição de arborescência enraizada e poderiam criar ciclos envolvendo a raiz, o que tornaria impossível obter uma estrutura válida. Além disso, a presença de arcos entrando na raiz interfere na normalização: ao tentar normalizar custos de entrada para r_0, criaríamos custos reduzidos artificiais que não fazem sentido no contexto do problema, já que nenhuma solução válida pode incluir tais arcos.

A escolha de implementar esta operação e as demais funções auxiliares fora do escopo da execução principal segue princípios de design de software: (1) *modularidade*, encapsulando uma responsabilidade específica e bem definida (remover entradas na raiz) em uma unidade testável independente; (2) *reutilização*, permitindo que outras partes do código ou implementações alternativas possam chamar esta operação quando necessário sem duplicar lógica; (3) *clareza semântica*, dando um nome descritivo (`remove_edges_to_r0`) que documenta a intenção da operação no ponto de chamada, tornando a função principal mais legível ao abstrair detalhes de implementação; e (4) *facilidade de teste*, possibilitando escrever testes unitários focados exclusivamente nesta operação de pré-processamento, verificando casos extremos (como grafos onde a raiz já não tem predecessores ou onde todos os arcos entram na raiz) sem precisar testar toda a complexidade do algoritmo recursivo.

Em detalhes, ela recebe como entrada um digrafo `D` (objeto `nx.DiGraph`) e o raiz `r0`. A implementação armazena em uma lista todos os arcos que entram em `r0` usando o método `in_edges` (linha 2). Aqui precisamos armazenar os arcos em uma lista porque o método `in_edges` devolve um iterador, que quando sofre remoção direta sofre um erro devido à modificação da estrutura durante a iteração.

Em seguida, na linha 3, remove todos os arcos usando o método `remove_edges_from` da biblioteca NetworkX, o qual recebe como parâmetro uma lista de tuplas representando arcos na forma `(u, v)` e remove cada uma delas do digrafo. O método da NetworkX itera sobre a lista fornecida e, para cada tupla `(u, v)`, remove o arco correspondente da estrutura interna de adjacência. A complexidade de `remove_edges_from` é $O(k)$, onde k é o número de arcos na lista de entrada, pois cada remoção individual tem custo $O(1)$ em média devido ao uso de dicionários aninhados para armazenar arcos.

Por fim, a função devolve o digrafo `D` atualizado no próprio objeto com todos os arcos de entrada em `r0` removidos (linha 4). A complexidade total da função é $O(\deg^-(r_0))$, pois a operação coleta e remove cada arco de entrada uma única vez.

Remoção de arcos que entram na raiz

Remove todos os arcos que entram na raiz `r0`, modificando `D` ao invés de criar uma cópia e devolve o digrafo atualizado.

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

2.3.3 Redução de custos por vértice (normalização):

Criamos uma função auxiliar para realizar a redução de custos por vértice - essa operação é chamada de normalização e calcula $y(v) = \min\{w(u, v)\}$ e substitui cada peso $w(u, v)$ por $w(u, v) - y(v)$, garantindo que ao menos um arco de entrada tenha custo zero. Como cada r-arborescência possui exatamente um arco entrando em cada vértice não-raiz, a soma total dos valores $y(v)$ subtraídos é constante para qualquer solução, preservando assim a ordem de otimalidade entre diferentes arborescências.

Recebe como entrada um digrafo `D` (objeto `nx.DiGraph`) e o vértice `node` a ser normalizado. A implementação armazena em uma variável `incoming_edges` todos os arcos de entrada de `node` com seus pesos usando o método `D.in_edges(node, data="w")`, que devolve uma lista de tuplas $(u, node, w)$ (linha 2) (fazemos isso para evitar repetição de código e deixar o código mais claro). Em seguida, calcula-se o peso mínimo `yv` através de uma compreensão de gerador que extrai o terceiro elemento de cada tupla (linha 3) e, para cada vértice `u`, se houver o atributo "w"subtrai `yv` do peso armazenado em `D[u][node]["w"]` (linha 7), caso contrário inicializa o peso como zero antes de subtrair (linha 6).

A função não devolve nenhum valor, pois a operação é realizada modificando

diretamente a estrutura: o digrafo `D` passado como parâmetro é modificado diretamente, e ao menos um arco de entrada de `node` terá custo reduzido zero após a execução. A complexidade é $O(\deg^-(v))$, pois cada operação percorre os arcos de entrada uma única vez.

Redução de custos por vértice (normalização)

Normaliza os pesos dos arcos que entram em `node`, subtraindo de cada uma o menor peso de entrada. Modifica o digrafo D no próprio objeto.

```
1 def reduce_weights(D: nx.DiGraph, node: str):
2     incoming_edges = D.in_edges(node, data=True)
3     yv = min((data.get("w", 0) for _, _, data in incoming_edges))
4     for u, _, _ in incoming_edges:
5         if "w" not in D[u][node]:
6             D[u][node]["w"] = 0
7         D[u][node]["w"] -= yv
```

A Figura 21 ilustra o funcionamento da normalização:

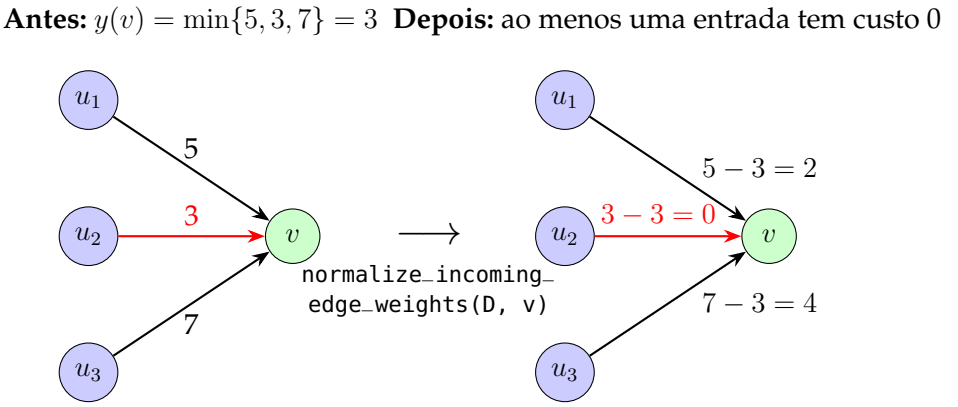


Figura 21 – Exemplo de normalização de custos reduzidos. À esquerda, vértice v com três arcos de entrada (pesos 5, 3 e 7). À direita, após aplicar `reduce_weights(D, v)`: o menor peso $y(v) = 3$ é subtraído de todas as entradas, resultando em custos reduzidos 2, 0 e 4. O arco (u_2, v) (em vermelho) tem custo zero e será selecionado para A_0.

Observe que as diferenças relativas são preservadas: o arco mais caro permanece 4 unidades acima da mais barata, e a intermediária mantém sua posição relativa.

Vale destacar que, quando o vértice recebe apenas um arco de entrada, trivialmente o custo desse arco é reduzido a zero.

Considerando que cada r-arborescência contém exatamente um arco entrando em cada vértice não-raiz, a soma $\sum_{w \neq r} y(w)$ é constante para qualquer solução, garantindo

que a ordem de otimalidade seja preservada.

2.3.4 Construção de A_0:

Esta função constrói o subdigrafo A_0 selecionando, para cada vértice $v \neq r_0$, um único arco de custo reduzido zero que entra em v.

Recebe como entrada um digrafo `D` e a raiz `r0`. A implementação cria um novo digrafo vazio `A_zero` (linha 2) em vez de modificar `D` diretamente; essa escolha de criar uma estrutura separada é fundamental porque A_0 é um subdigrafo usado para detecção de ciclos, e preservar `D` inalterado permite que as operações subsequentes (como contração) trabalhem com o digrafo original completo, evitando perda de informação sobre arcos não selecionados que podem ser necessários após reexpansões.

Em seguida, para cada vértice `v` diferente de `r0` (linha 3), utilizando o método `D.nodes()` para iterar sobre todos os vértices, se v for diferente de `r0` (linha 4), obtém todos os arcos de entrada em v com seus pesos usando `D.in_edges(v, data=True)` (linha 5).

Em seguida, obtém o primeiro predecessor `u` cujo arco (u, v) tem peso zero, aramazenando-o na variável `u` (linha 6) utilizando uma compreensão de gerador combinada com `next`. A escolha de `next` com gerador em vez de uma busca exaustiva é eficiente porque interrompe a iteração assim que encontra o primeiro arco de custo zero, evitando processamento desnecessário dos arcos restantes (embora teoricamente todos os arcos de custo zero sejam equivalentes, na prática apenas um é necessário para A_0). Finalmente, adiciona o arco (u, v) com peso zero a `A_zero` (linha 7).

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

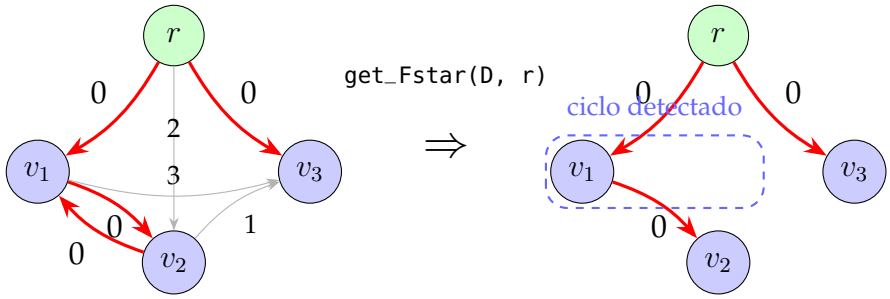
Construção de A zero

Constrói o subdigrafo A_0 a partir do digrafo D, selecionando para cada vértice (exceto a raiz r0) um arco de custo reduzido zero que entra nele.

```
1 def get_Azero(D: nx.DiGraph, r0: str):
2     A_zero = nx.DiGraph()
```

```
    for v in D.nodes():
        if v != r0:
            in_edges = D.in_edges(v, data=True)
            u = next((u for u, _, data in in_edges if data.get("w") == 0))
            A_zero.add_edge(u, v, w=0)
    return A_zero
```

A Figura 22 ilustra a construção de A_0:



Digrafo D (normalizado) **Subgrafo F^***

Figura 22 – Exemplo de construção de A_0 a partir de um digrafo normalizado. À esquerda, o digrafo D após normalização, onde cada vértice não-raiz possui ao menos um arco de entrada com custo zero (em vermelho). À direita, o afo A_0 resultante contém apenas os arcos de custo zero selecionados, um por vértice. Note que A_0 pode conter ciclos (como $\{v_1, v_2\}$) que serão tratados nas etapas subsequentes.

As funções de normalização por vértice e construção de A_0 juntas implementam o passo 1 da descrição do algoritmo de Chu–Liu/Edmonds:

Passo 1 do Algoritmo de Chu–Liu/Edmonds
Passo 1: Para cada $v \neq r$, escolha $a_v \in \arg\min_{(u,v)\in A} c(u, v)$. Defina $y(v) := c(a_v)$ e $A_0 := \{a_v : v \neq r\}$.

2.3.5 Detecção de ciclo:

A detecção de ciclos é crucial, pois a presença de um ciclo em A_0 indica que a escolha de arcos de custo reduzido zero não formou uma arborescência válida.

Logo, a função apresentada a seguir detecta a presença de um ciclo dirigido em A_0 e devolve um subdigrafo que o contém; Não verificamos se o digrafo é acíclico, pois a função principal não deve sequer fazer essa verificação: se não houver ciclo, é porque uma arborecência já foi encontrada.

Recebe como entrada um digrafo `A_zero`. A função inicializa um conjunto vazio `nodes_in_cycle` (linha 2). O laço na linha 3 itera sobre os arcos devolvidos pela função `nx.find_cycle(A_zero, orientation="original")`, que utiliza uma busca em profundidade (DFS) para detectar ciclos dirigidos. Se um ciclo for encontrado, a função devolve um iterador sobre os arcos do ciclo, desempacotando cada uma na forma `(u, v, _)` (ignorando o terceiro elemento com `_`, que contém metadados de orientação), e na linha 4 para cada arco (u, v) adiciona ambos os vértices ao conjunto `nodes_in_cycle` (linha 4); a escolha de usar conjunto em vez de lista garante que cada vértice seja adicionado apenas uma vez mesmo que o ciclo tenha múltiplos arcos incidentes, e a operação de adição tem complexidade $O(1)$.

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

No final, um subdigrafo contendo os vértices e arcos do ciclo detectado é devolvido. O digrafo original `A_zero` não é modificado. A complexidade é $O(m)$, onde m é o número de arcos, pois a DFS visita cada arco no máximo uma vez.

Detecção de ciclo dirigido em A_0

Detecta um ciclo dirigido em A_0 e devolve um subdigrafo contendo seus vértices e arcos, ou `None` se for acíclico.

```
1 def find_cycle(A_zero: nx.DiGraph):
2     nodes_in_cycle = set()
3     for u, v, _ in nx.find_cycle(A_zero, orientation="original"):
4         nodes_in_cycle.update([u, v])
5     return A_zero.subgraph(nodes_in_cycle).copy()
```

A Figura 23 ilustra o processo de detecção de ciclo:

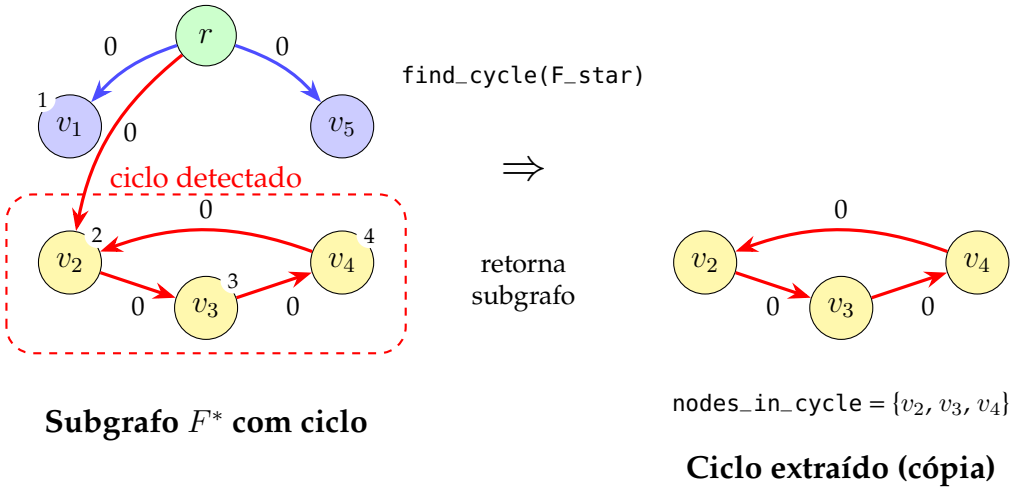


Figura 23 – Exemplo de detecção de ciclo em A_0. À esquerda, o subdigrafo A_0 contém um ciclo formado pelos vértices $\{v_2, v_3, v_4\}$ (destacados em amarelo). A DFS percorre o digrafo e detecta o ciclo ao encontrar o arco (v_4, v_2), onde v_2 já está na pilha de recursão. À direita, a função devolve uma cópia do subdigrafo induzido pelos vértices do ciclo, contendo apenas os três vértices e os três arcos que formam o ciclo.

Ao detectar um ciclo, o código avança para a etapa de contração executando operações concretas sobre o objeto Python: primeiro, coleta os vértices de C num conjunto para consultas em $O(1)$; em seguida, para cada vértice externo $u \notin C$ determina o arco de menor peso que vai de u para algum $w \in C$ e guarda esse par em extttin_to_cycle[u]=(w,weight); análogamente, para cada vértice externo $v \notin C$ determina o arco de menor peso que vai de algum $w \in C$ para v e guarda em `out_from_cycle[v]=(w,weight)`. Depois de coletar estas informações (para evitar mutações durante iterações), o código cria no digrafo `D` arcos redirecionados `(u, label)` e `(label, v)` com os pesos mínimos correspondentes. Por fim, a função devolve os dicionários `in_to_cycle` e `out_from_cycle`, que serão usados na reexpansão para reconstruir corretamente os arcos do ciclo original. Note que a substituição efetiva do ciclo pelo supervértice é realizada pelas operações de remoção/adaptação subsequentes no procedimento principal; aqui apenas são computadas e adicionadas as arestas de ligação e preservadas as informações necessárias para a reexpansão.

2.3.6 Contração de ciclo:

Escrevemos uma função responsável por contrair um ciclo dirigido C a um supervértice x_C, redirecionando arcos incidentes e ajustando custos segundo a regra de custos reduzidos. No final, a função devolve dois dicionários auxiliares com informações dos vértices que incidiam e ascendiam de C, essenciais para a etapa posterior de reexpansão da arborescência.

A função recebe como entrada um digrafo `D`, o ciclo `C` a ser contraído (que fora detectado pela função anterior) e parâmetro de rotulação do novo supervértice `label`. Inicialmente, coletam-se os vértices de `C` em um conjunto (linha 2) para permitir verificações de pertinência em tempo $O(1)$, essencial dado que essa operação é realizada repetidamente nos laços seguintes. Inicializa-se então `in_to_cycle` (linha 3), um dicionário que tem como chave vértices externos ao ciclo e cujo valor são tuplas (v, w), onde v é o vértice do ciclo conectado a u e w é o peso do arco (u, v); essa estrutura preserva não apenas o peso mínimo, mas também o ponto exato de entrada no ciclo.

Para cada vértice `u` no digrafo `D` (linha 4), se `u` não pertence ao ciclo (linha 5), identifica-se o arco de menor peso que sai de `u` e entra em `C` (linhas 6–9) usando a função `min` combinada com uma compreensão de gerador: `((v, data.get("w", float("inf"))) for _, v, data in D.out_edges(u, data=True) if v in cycle_nodes)` a qual itera sobre todos os arcos que saem de `u`, desempacota cada arco na forma `(_, v, data)` (ignorando a origem com `_`, capturando o destino `v` e o peso `data`), filtra apenas aquelas cujo destino `v` pertence ao ciclo, e produz tuplas `(v, w)`; a função `min` (linha 6) então seleciona a tupla de menor peso usando `key=lambda x: x[1]` (linha 10) para comparar pelo segundo elemento (o peso), e devolve `None` se não houver arcos (linha 11). A escolha de selecionar apenas o arco de *menor peso* reflete a propriedade fundamental do algoritmo: qualquer solução ótima que conecta um vértice externo ao ciclo contraído usará necessariamente o arco de custo mínimo, pois todas as outras opções seriam subótimas. Se tal arco existir (linha 12), armazena em `in_to_cycle[u]` (linha 14).

Em seguida, a implementação itera sobre `in_to_cycle` usando o método `items()`, desempacotando cada entrada na forma `(u, (v, w))`, onde `u` é o vértice externo e `(v, w)` é a tupla com o vértice do ciclo e o peso (linha 14). Para cada par, cria um arco de `u` para `label` com peso `w`, efetivamente redirecionando os arcos de entrada para o supervértice (linha 15). A separação entre coleta (linhas 4–10) e criação (linhas 11–12) é necessária porque modificar o digrafo durante a iteração sobre seus vértices causaria comportamento indefinido; ao coletar primeiro todos os dados em estruturas auxiliares, garantimos que as modificações posteriores sejam seguras.

De forma análoga, constrói-se o dicionário `out_from_cycle` (linha 13) para mapear arcos que saem do ciclo.

Finalmente, dois dicionários são devolvidos: `in_to_cycle` mapeia vértices externos aos pontos de entrada no ciclo original, e `out_from_cycle` mapeia vértices externos aos pontos de saída. Esses dicionários são essenciais para a fase de reexpansão, onde será necessário determinar exatamente qual arco interno do ciclo deve ser removido a fim de manter um caminho que conecta todos os vértices. O digrafo `D` é modificado sem criar uma cópia: os vértices de `C` são removidos e substituídos por `label`. A escolha de modificação no próprio objeto (em vez de criar uma cópia) reduz significativamente

o uso de memória e o tempo de execução, especialmente em grafos grandes ou com múltiplos níveis de recursão, embora exija atenção cuidadosa para que informações originais sejam preservadas. A complexidade é $O(m)$, onde m é o número de arcos, pois cada arco incidente ao ciclo é processado uma vez: os laços nas linhas 4–10 e 14–19 examinam cada arco no máximo uma vez, e as operações de adição (linhas 11–12, 20–21) e remoção (linha 22) têm custo proporcional ao número de arcos afetados.

Contração de ciclo

Contrai o ciclo C em um supervértice `label`, redirecionando arcos incidentes e ajustando custos. Modifica D no próprio objeto e devolve dicionários para reexpansão.

```
1  def contract_cycle(D: nx.DiGraph, C: nx.DiGraph, label: str):
2      cycle_nodes: set[str] = set(C.nodes())
3      in_to_cycle: dict[str, tuple[str, float]] = {}
4      for u in D.nodes:
5          if u not in cycle_nodes:
6              min_weight_edge_to_cycle = min(
7                  ((v, data.get("w", float("inf")))
8                      for _, v, data in D.out_edges(u, data=True)
9                      if v in cycle_nodes),
10                 key=lambda x: x[1],
11                 default=None,)
12             if min_weight_edge_to_cycle:
13                 in_to_cycle[u] = min_weight_edge_to_cycle
14     for u, (v, w) in in_to_cycle.items():
15         D.add_edge(u, label, w=w)
16     out_from_cycle: dict[str, tuple[str, float]] = {}
17     for v in D.nodes:
18         if v not in cycle_nodes:
19             min_weight_edge_from_cycle = min(
20                 ((u2, data.get("w", float("inf")))
21                     for u2, _, data in D.in_edges(v, data=True)
22                     if u2 in cycle_nodes),
23                 key=lambda x: x[1],
24                 default=None,)
25             if min_weight_edge_from_cycle:
26                 out_from_cycle[v] = min_weight_edge_from_cycle
27     for v, (u, w) in out_from_cycle.items():
28         D.add_edge(label, v, w=w)
29     return in_to_cycle, out_from_cycle
```

A Figura 24 ilustra o processo de contração de ciclo:

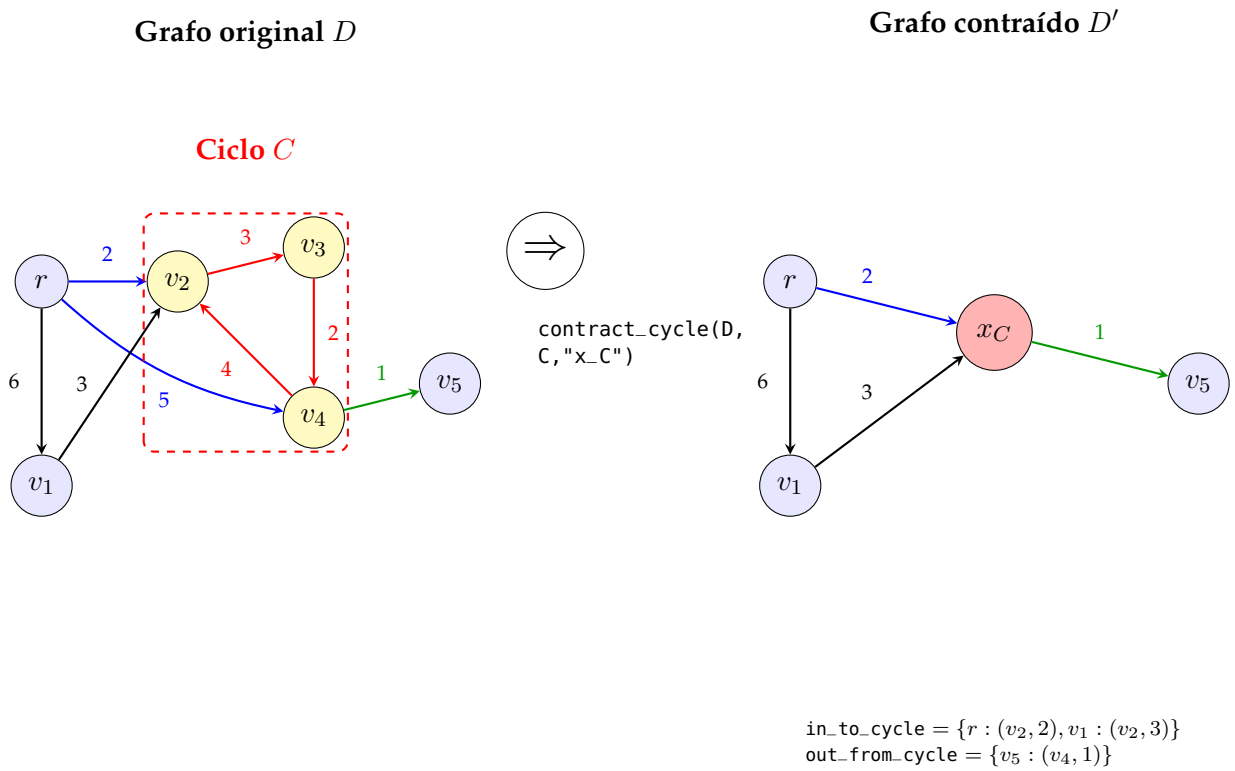


Figura 24 – Exemplo de contração de ciclo. À esquerda, digrafo original D com ciclo $C = \{v_2, v_3, v_4\}$ (em amarelo). Vértices externos r, v_1 e v_5 têm arcos conectando ao ciclo: r envia arco para v_2 (peso 2) e v_4 (peso 5); v_4 envia arco para v_5 (peso 1). À direita, após a contração: o ciclo é substituído pelo supervértice x_C (vermelho). Os arcos de entrada são redirecionados: (r, x_C) recebe peso 2 (menor entre 2 e 5). O arco de saída (x_C, v_5) mantém peso 1. Os dicionários `in_to_cycle` e `out_from_cycle` armazenam os mapeamentos originais para posterior reexpansão.

A função de deteção de ciclo e a de contração juntas implementam os passos 2 e 3 da descrição do algoritmo de Chu–Liu/Edmonds:

Passos 2 e 3 do Algoritmo de Chu–Liu/Edmonds
Passo 2: Se (V, A_0) é acíclico, devolva A_0. Por (KLEINBERG; TARDOS, 2006, Obs. 4.36), trata-se de uma r-arborescência de custo mínimo.

Passo 3: Caso contrário, seja C um ciclo dirigido de A_0 (com $r \notin C$). **Contração:** contraia C em um supervértice x_C e defina custos c' por

$$c'(u, x_C) := c(u, w) - y(w) = c(u, w) - c(a_w) \qquad \text{para } u \notin C,\ w \in C,$$
$$c'(x_C, v) := c(w, v) \qquad \text{para } w \in C,\ v \notin C,$$

descartando laços em x_C e permitindo paralelos. Denote o digrafo contraído por $D' = (V', A')$.

2.3.7 Remoção de arco interno:

Esta função é invocada durante a fase de reexpansão do ciclo contraído, após a chamada recursiva devolver com a arborescência ótima T' do digrafo contraído. Quando o supervértice x_C é expandido de volta para o ciclo original C, um arco externo (u, v) é adicionado conectando um vértice externo u a um vértice v dentro do ciclo. Como o ciclo C originalmente continha exatamente um arco entrando em cada um de seus vértices (formando um ciclo fechado), e agora v recebe um arco adicional vindo do exterior, esse vértice teria grau de entrada 2, violando a propriedade fundamental de arborescência (cada vértice não-raiz deve ter exatamente uma entrada). Para restaurar essa propriedade, a função remove o arco interno que anteriormente entrava em v, mantendo apenas o novo arco externo. Essa remoção "quebra"o ciclo no ponto de entrada, transformando-o em um caminho que se integra corretamente à estrutura de árvore.

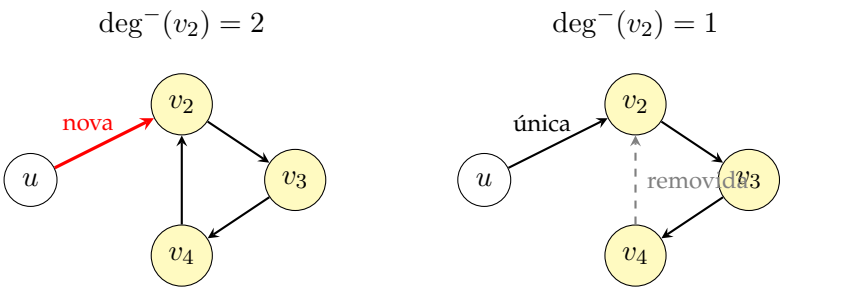
A função recebe como entrada o ciclo `C` e o vértice de entrada `v`. A implementação utiliza uma compreensão de gerador combinada com `next` para encontrar o predecessor de `v` dentro do ciclo (linha 2): a expressão `(u for u, _ in C.in_edges(v))` itera sobre os arcos de entrada de `v`, extraindo apenas o vértice origem `u` (ignorando metadados com `_`), e `next` devolve o primeiro (e teoricamente único) predecessor. Em seguida, remove o arco `(predecessor, v)` do ciclo usando o método `remove_edge` (linha 3).

A função modifica diretamente o subdigrafo `C` e não devolve valor. A complexidade é $O(\deg^-(v))$, dominada pela operação de busca dos arcos de entrada, embora em ciclos simples isso seja tipicamente $O(1)$ pois cada vértice tem exatamente um predecessor.

Remover arco interno na reexpansão
Remove o arco interno que entra no vértice de entrada `v` do ciclo `C` durante a reexpansão, pois `v` passa a receber um arco externo, e manter ambos violaria a propriedade de arborescência.

```
1 def remove_edge_cycle(C: nx.DiGraph, v):
2     predecessor = next((u for u, _ in C.in_edges(v)))
3     C.remove_edge(predecessor, v)
```

A Figura 25 ilustra o objetivo da função:



Ciclo C após adicionar aresta externa (u, v_2)

Após remover aresta interna (v_4, v_2)

Figura 25 – Remoção de arco interno durante reexpansão. À esquerda, ciclo $C = \{v_2, v_3, v_4\}$ após adicionar arco externo (u, v_2) vindouro da arborescência T': o vértice v_2 tem grau de entrada 2 (arco externo vermelho de u e arco interno do ciclo vindo de v_4), violando a propriedade de arborescência. À direita, após remover o arco interno (v_4, v_2): o vértice v_2 passa a ter grau de entrada 1, o ciclo é "quebrado"no ponto de entrada, transformando-se em um caminho que se integra corretamente à estrutura de árvore. O arco removido é mostrado tracejado em cinza.

2.3.8 Procedimento principal (recursivo):

Agora apresentaremos a função principal que orquestra todas as funções auxiliares descritas anteriormente no fluxo completo do algoritmo descrito por Chuliu-Edmonds. A função recebe como entrada um digrafo ponderado `D`, o vértice raiz `r0`, e um parâmetro `level` (padrão 0) usado para rotular supervértices de acordo os distintos níveis recursivos.

A implementação segue a estrutura do algoritmo:

Preservação do digrafo original (linha 2):

Cria uma cópia `D_copy = D.copy()` para preservar os pesos originais e chamamos uma função de cast() para garantir que o tipo seja `nx.DiGraph`, pois o método `copy()`

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

Normalização e construção de A_0 (linhas 3–6):

Itera sobre todos os vértices não-raiz (linhas 3–5), chamando `reduce_weights(D_copy, v)` para cada um. Após normalizar todos os vértices, constrói A_0 (linha 6) chamando `get_Azero(D_copy, r0)`, que seleciona um arco de custo reduzido zero entrando em cada vértice não-raiz.

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

Verifica se A_0 é uma arborescência válida usando `nx.is_arborescence(A_zero)` (linha 7). Caso sim, restaura os pesos originais de `D` para cada arco de A_0 (linhas 8–9) e devolve `A_zero` como solução (linha 10). A função `nx.is_arborescence` testa conectividade, aciclicidade e o grau de entrada correto simultaneamente. Precisamos verificar essa condição porque, se A_0 for acíclico, já encontramos a arborescência ótima e não há necessidade de contração ou recursão. Essa verificação é portanto o caso base da recursão.

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

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

Reexpansão do ciclo contraído (linhas 15–28):

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

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

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

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

O código completo da função principal é apresentado a seguir:

Procedimento principal (recursivo)

Implementa o algoritmo de Chu–Liu/Edmonds de forma recursiva para encontrar a r-arborescência de custo mínimo em um digrafo ponderado `D` com raiz `r0`. Normaliza custos, constrói A_0, detecta ciclos e, se houver, contrai em supervértice, resolve recursivamente no digrafo reduzido e reexpande, restaurando a arborescência ótima no digrafo original. Devolve um `nx.DiGraph` contendo exatamente $|V| - 1$ arcos com grau de entrada 1 para cada vértice exceto a raiz.

```
1  def chuliu_edmonds(D: nx.DiGraph, r0: str,level=0):
2      D_copy = cast(nx.DiGraph, D.copy())
3      for v in D_copy.nodes:
4          if v != r0:
5              reduce_weights(D_copy, v)
6      A_zero = get_Azero(D_copy, r0)
7      if nx.is_arborescence(A_zero):
8          for u, v in A_zero.edges:
9              A_zero[u][v]["w"] = D[u][v]["w"]
10         return A_zero
11     C = find_cycle(A_zero)
12     cl = f"\n n*{level}" # contracted label
13     in_to_cycle, out_from_cycle = contract_cycle(D_copy, C, cl)
14     F_prime = chuliu_edmonds(D_copy, r0,level + 1)
15     in_edge = next(iter(F_prime.in_edges(cl, data=True)))
```

```
16      u, _, _ = cast(tuple, in_edge)
17      v, _ = in_to_cycle[u]
18      remove_edge_cycle(C, v)
19      F_prime.add_edge(u, v)
20      for u_c, v_c in C.edges:
21          F_prime.add_edge(u_c, v_c)
22      for _, z, _ in F_prime.out_edges(cl, data=True):
23          u_cycle, _ = out_from_cycle[z]
24          F_prime.add_edge(u_cycle, z)
25      F_prime.remove_node(cl)
26      for u2, v2 in F_prime.edges:
27          F_prime[u2][v2]["w"] = D[u2][v2]["w"]
28      return F_prime
```

Os passos do algoritmo implementados nesta função em conjunto com a função de remover arco interno do ciclo na reexpansão correspondem diretamente à descrição formal do algoritmo de Chu–Liu/Edmonds da seguinte forma:

Passos 4 e 5 do Algoritmo de Chu–Liu/Edmonds

Passo 4 — Resolução recursiva:

- Para resolver o digrafo contraído D' aplica-se uma chamada recursiva: `F_prime = chuliu(D_copy, r0, level+1)` (linha 14 na implementação).
- Essa chamada executa novamente a normalização, construção de A_0, detecção/contração de ciclos e prossegue até que o caso base (arborescência em D') seja atingido.

Passo 5 — Reexpansão:

- Após obter `F_prime` em D', identifica-se o arco `(u, contracted_label)` que entra no supervértice; no digrafo original esse arco corresponde a `(u,v)` com `v = in_to_cycle[u]`.
- Remove-se o arco interno que entrava em `v` (quebrando o ciclo) — função `remove_internal_edge_to_cycle_entry(C, v)` —, adiciona-se o arco externo `(u,v)` e reintegram-se os demais arcos de C; saídas do ciclo são tratadas via `out_from_cycle` (implementado nas linhas 15–28).
- O resultado é uma arborescência em D com pesos originais restaurados.

Aqui finalizamos a descrição detalhada da implementação do algoritmo de

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

Exemplo de execução do algoritmo

Aqui ilustraremos cada fase do processo: normalização, construção de A_0, detecção de ciclos, contração, resolução recursiva e reexpansão a partir do digrafo inicial mostrado na Figura 26.

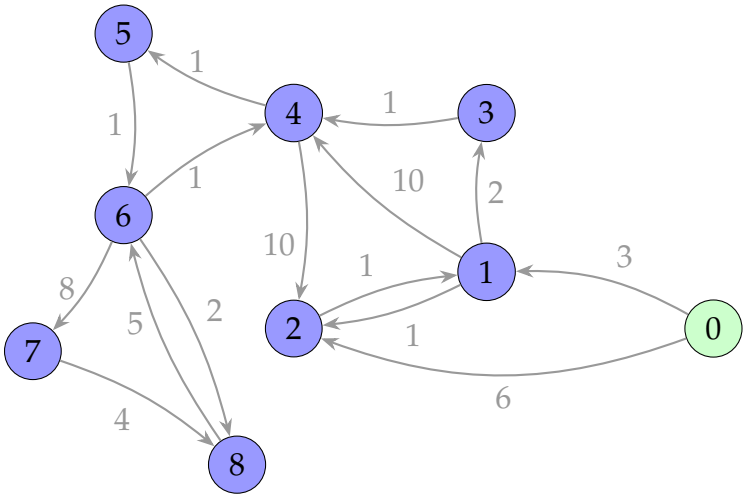


Figura 26 – digrafo direcionado ponderado inicial com raiz no vértice 0. O digrafo contém 9 vértices e múltiplos arcos com pesos variados. O primeiro passo do algoritmo seria remover arcos que entram na raiz, porém não há nenhum neste caso, logo não existe necessidade de alterar o digrafo.

O primeiro passo do nosso algoritmo seria remover os arcos que entram na raiz (vértice 0), porém não há nenhum nesse caso, logo não existe a necessidade de alterar o digrafo.

O próximo passo é normalizar os pesos dos arcos de entrada para cada vértice. Nessa etapa, para cada vértice v (exceto a raiz), o algoritmo encontra o arco de menor peso que entra em v e subtrai esse menor peso de todos os arcos que entram em v (isso serve para zerar o peso do arco mínimo de entrada em cada vértice).

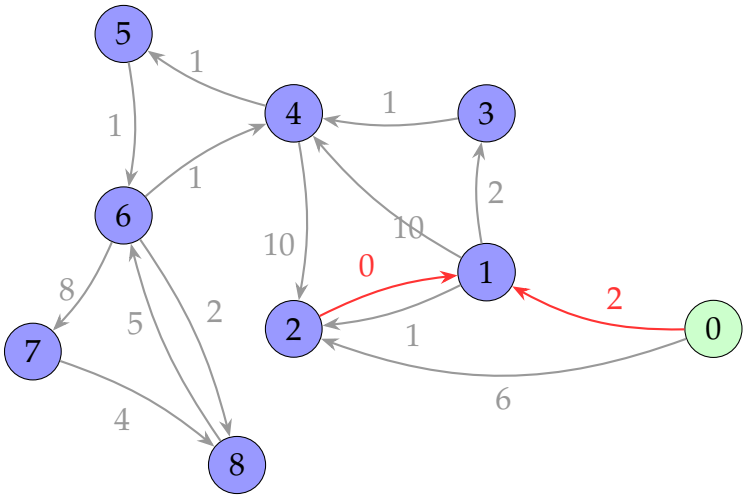


Figura 27 – Normalização parcial dos arcos de entrada para o vértice 1. Os arcos de entrada são $(0 \to 1)$ com peso original 3 e $(2 \to 1)$ com peso original 1. Elegendo o arco $(2 \to 1)$ como o de menor peso (peso mínimo = 1), subtraímos este valor de todos os arcos de entrada: $(0 \to 1)$ passa de peso 3 para 2, e $(2 \to 1)$ passa de peso 1 para 0 (destacadas em vermelho). Esse processo é repetido para todos os demais vértices.

Com os pesos normalizados, o próximo passo é construir A_0: para isso, selecionamos para cada vértice o arco de custo reduzido zero de entrada. Detectamos um ciclo em A_0, formado pelos vértices $\{1, 2\}$. Portanto, precisamos contrair esse ciclo em um supervértice $n * 0$.

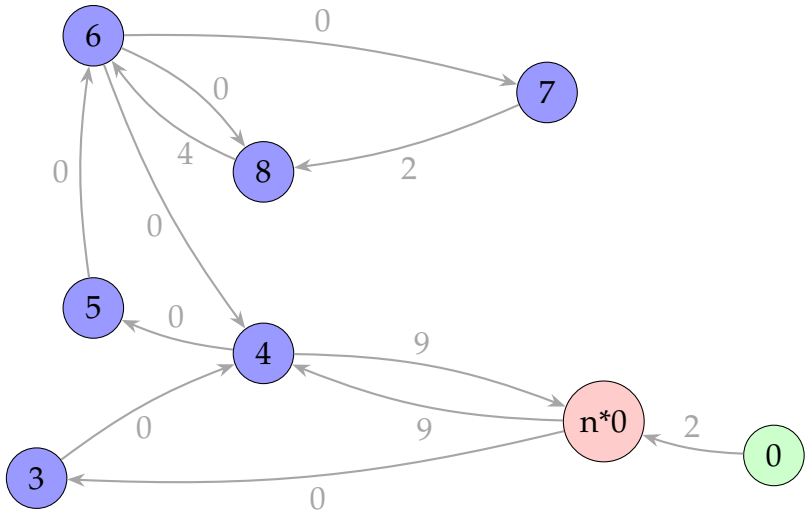


Figura 28 – digrafo contraído após detecção do ciclo $C = \{1, 2\}$ em A_0. O ciclo foi contraído no supervértice $n * 0$ (destacado em vermelho). Os arcos que entravam ou saíam do ciclo foram redirecionados para o supervértice, com custos ajustados segundo as fórmulas $c'(u, x_C) := c(u, w) - y(w)$ para arcos de entrada e $c'(x_C, v) := c(w, v)$ para arcos de saída.

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

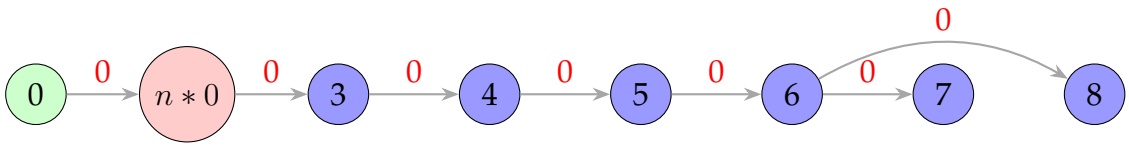


Figura 29 – Arborescência ótima F' obtida no digrafo contraído. todos os arcos selecionados têm custo reduzido 0 (destacados em vermelho), e o digrafo forma uma arborescência válida enraizada em 0: cada vértice (exceto a raiz) tem exatamente um arco de entrada, não há ciclos, e todos os vértices são alcançáveis a partir da raiz. Como F' é acíclico, alcançamos o caso base da recursão.

Após validarmos que A_0 não possui mais ciclos e forma uma arborescência, iniciamos o processo de reexpansão do ciclo contraído para obter a arborescência final no digrafo original. Adicionamos o arco de entrada ao ciclo $(0, 1)$, os arcos internos do ciclo modificado $(1, 2)$, e os arcos de saída $(1, 3)$, chegando a uma arborescência válida.

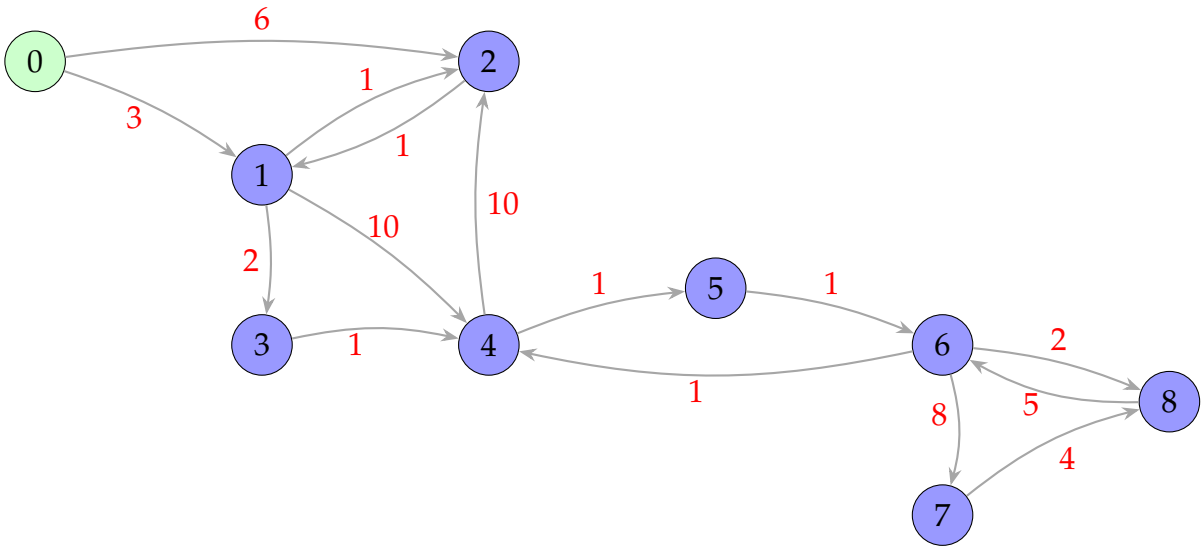


Figura 30 – Arborescência ótima final no digrafo original com pesos restaurados. O supervértice $n * 0$ foi expandido de volta para os vértices 1 e 2, com o arco externo $(0, 1)$ escolhido pela solução recursiva conectando ao ciclo. O arco interno $(2, 1)$ do ciclo original foi removido para manter a propriedade de arborescência ($\deg^-(v) = 1$). O resultado é uma 0-arborescência de custo mínimo com exatamente 8 arcos, onde cada vértice não-raiz tem grau de entrada 1 e todos são alcançáveis a partir da raiz 0.

2.3.9 Correspondência entre teoria e implementação

A implementação em Python segue fielmente os cinco passos da descrição teórica do algoritmo de Chu–Liu/Edmonds apresentada na Seção anterior. A tabela abaixo estabelece o paralelo direto entre cada passo teórico e sua realização no código:

Descrição Teórica	**Implementação Python**

Passo 1: Normalização e construção de A_0
Para cada $v \neq r$, escolha
$a_v \in \arg\min_{(u,v)\in A} c(u,v)$.
Defina $y(v) := c(a_v)$ e $A_0 := \{a_v : v \neq r\}$.
	Linhas 3–6:
	`for v in D_copy.nodes:`
	`  reduce_weights(D_copy, v)`
	`A_zero = get_Azero(D_copy, r0)`
	Calcula $y(v)$ e cria custos reduzidos, depois constrói A_0 selecionando arcos de custo zero.

Passo 2: Verificação de aciclicidade (caso base)
Se (V, A_0) é acíclico, devolva A_0. Por Obs. 4.36 de (KLEINBERG; TARDOS, 2006), trata-se de uma r-arborescência de custo mínimo.
	Linhas 7–10:
	`if nx.is_arborescence(A_zero):`
	`  [restaura pesos originais]`
	`  return A_zero`
	Testa conectividade, aciclicidade e grau de entrada correto simultaneamente.

Passo 3: Contração de ciclo
Caso contrário, seja C um ciclo dirigido de A_0 (com $r \notin C$). Contraia C em supervértice x_C e defina custos c' por:
$c'(u, x_C) := c(u, w) - y(w)$
$c'(x_C, v) := c(w, v)$
Denote o digrafo contraído por $D' = (V', A')$.
	Linhas 11–13:
	`C = find_cycle(A_zero)`
	`label = f"contracted_{level}"`
	`in_to_cycle, out_from_cycle =`
	`  contract_cycle(D_copy, C, label)`
	Implementa as fórmulas de ajuste de custos e modifica `D_copy` para criar D'.

Passo 4: Resolução recursiva
Resolva recursivamente em D', obtendo arborescência ótima F'.
	Linha 14:
	`F_prime = chuliu(`
	`  D_copy, r0, level+1)`
	Chamada recursiva resolve o problema no digrafo contraído.

Passo 5: Reexpansão
Expanda x_C para o ciclo original C. Se $(u, x_C) \in F'$, adicione (u, v) onde v é o vértice do ciclo mapeado por u, remova o arco interno entrando em v, e reintegre demais arcos de C. Restaure custos originais.
	Linhas 15–28:
	`v = in_to_cycle[u]`
	`remove_internal_edge_to_cycle_entry(C,`
	`v)`
	`F_prime.add_edge(u, v)`
	`F_prime.add_edges_from(C.edges)`
	`[processa saídas, remove supervértice]`
	`[restaura pesos originais]`

Tabela 1 – Correspondência entre os cinco passos teóricos do algoritmo de Chu–Liu/Edmonds e sua implementação em Python. Cada linha da coluna direita mostra a tradução direta dos conceitos matemáticos da coluna esquerda em operações concretas sobre grafos.

Esta correspondência demonstra que a implementação não é uma aproximação ou interpretação livre da teoria, mas uma tentativa de traduzir fielmente a descrição teórica. As funções auxiliares (`reduce_weights`, `get_Azero`, `find_cycle`, `contract_cycle`, `remove_edge_cycle`) encapsulam exatamente as operações descritas na formulação teórica, preservando as propriedades de correção e complexidade do algoritmo original.

2.3.10 Transição para a abordagem primal-dual

Embora o algoritmo de Chu–Liu/Edmonds seja elegante e eficiente, sua mecânica operacional — normalizar custos, selecionar mínimos, contrair ciclos — pode ser melhorada em termos de intuição e generalização.

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

3 Algoritmo de András Frank

Neste capítulo, apresentaremos o algoritmo de András Frank, que também determina uma arborescência de custo mínimo em um digrafo ponderado. O algoritmo baseia-se em duas operações fundamentais: (i) a redução gulosa dos custos dos arcos através da identificação de subconjuntos minimais e (ii) a contração de ciclos, de modo a resolver recursivamente uma instância menor do problema e, em seguida, estender a solução para o problema original. A operação de redução é essencialmente a mesma do algoritmo de Chu–Liu–Edmonds — subtrair o menor custo de arco entrando em cada conjunto — mas enquanto Chu–Liu–Edmonds opera vértice a vértice, o algoritmo de Frank identifica *subconjuntos minimais* através de componentes fortemente conexas, processando todos simultaneamente a cada iteração. O propósito deste capítulo é fornecer tanto uma descrição teórica do algoritmo quanto detalhes da implementação desenvolvida neste trabalho.

3.1 O algoritmo

O algoritmo de András Frank também recebe uma tripla (D, c, r), em que $D = (V, A)$ é um digrafo, $c\colon A \to \mathbb{R}$ é uma função custo e $r \in V$ é a raiz, sob a hipótese de que D admite ao menos uma r-arborescência e devolve uma r-arborescência c-mínima de D.

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

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

Na primeira fase, identificamos iterativamente os *subconjuntos minimais* um conjunto $X \subseteq V \setminus \{r\}$ é dito minimal se não há arcos justos entrando em X (ou seja, nenhum arco (u, v) com $u \notin X$ e $v \in X$ possui custo reduzido zero) e para qualquer subconjunto próprio $Y \subset X$, existe ao menos um arco justo entrando em Y, essa segunda condição garante a propriedade de minimalidade, pois não podemos encontrar um subconjunto menor que também não tenha arcos justos entrando.

Assim, para cada conjunto minimal X, calculamos $\delta(X)$, o menor custo entre todos os arcos que entram em X, e subtraímos esse valor de todos esses arcos, criando ao menos um novo arco justo. Esse processo é repetido até que exista exatamente uma

fonte no digrafo atualizado também chamado de digrafo de condensação.

Aqui *fonte* refere-se a um vértice no digrafo de condensação que não tenha arcos justos entrando.

Vamos desenvolver essas ideias utilizando um exemplo: considere o dígrafo D da Figura 31 com seis vértices $\{r, a, b, c, d, e\}$ e custos nos arcos para ilustrar o comportamento completo do algoritmo.

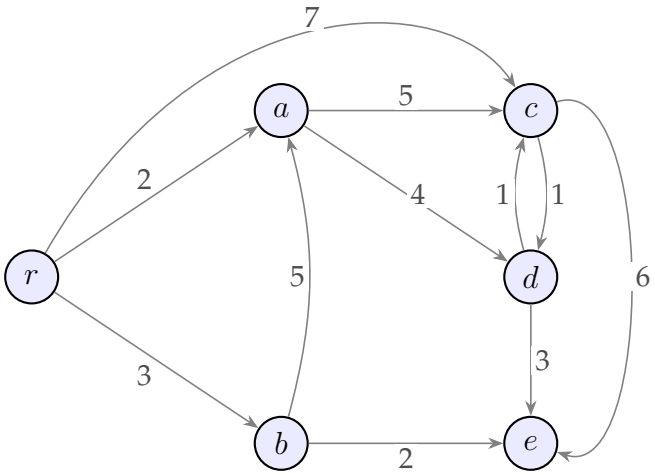


Figura 31 – Dígrafo D com custos originais. Este exemplo ilustrará todas as etapas do algoritmo de András Frank, incluindo formação de ciclos e contração.

Fase 1

Inicialmente, nenhum arco tem custo reduzido zero, cada vértice não-raiz $v \neq r$ forma seu próprio subconjunto minimal $\{v\}$. No dígrafo da Figura 31, os subconjuntos minimais iniciais são portanto $\{a\}$, $\{b\}$, $\{c\}$, $\{d\}$ e $\{e\}$. Para cada um desses conjuntos unitários, encontramos $\delta(\{v\})$, o menor custo entre todos os arcos entrando em v, e subtraímos esse valor de todos esses arcos.

Por exemplo, consideremos o vértice a. Os arcos entrando em a são (r, a) com custo 2 e (b, a) com custo 5. Calculamos $\delta(\{a\}) = \min\{2, 5\} = 2$ e subtraímos este valor de ambos os arcos. Assim, (r, a) passa a ter custo 0 e (b, a) passa a ter custo 3. A Figura 32 ilustra essa operação de redução aplicada ao vértice a.

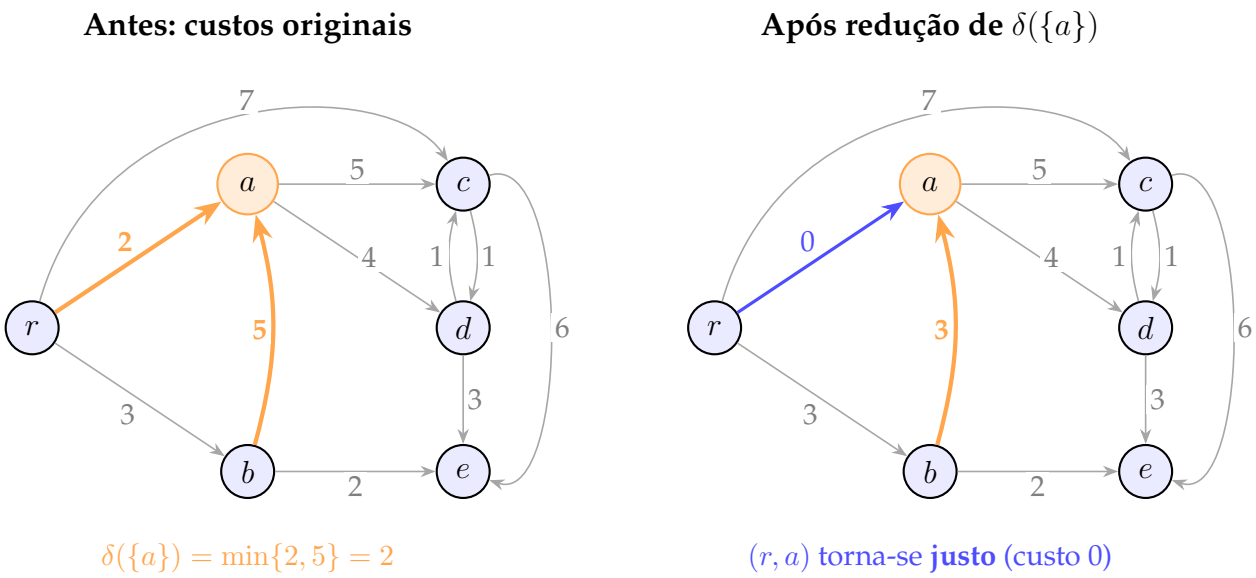


Figura 32 – Exemplo de redução de custo para o vértice a no dígrafo completo. À esquerda, os arcos entrando em a estão destacados em laranja com custos originais 2 e 5. Calculamos $\delta(\{a\}) = 2$ e subtraímos esse valor de ambos os arcos. À direita, após a redução: (r, a) tem custo zero (arco justo, em azul) e (b, a) tem custo $5 - 2 = 3$ (em laranja). Os demais arcos permanecem inalterados.

Essa operação vai sendo repetida para todos os conjuntos minimais iniciais. A Figura 33 mostra o resultado dessa primeira iteração no exemplo da Figura 31, onde aplicamos a redução apenas para os vértices isolados.

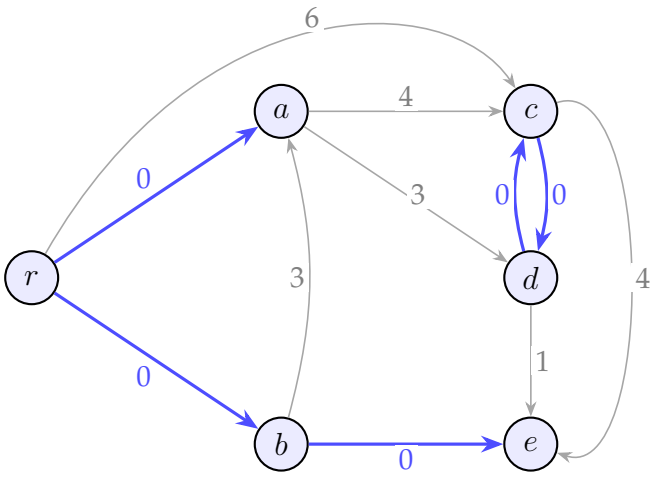


Figura 33 – Dígrafo após a primeira iteração. Os arcos justos (custo 0) são: (r, a), (r, b), (b, e), (c, d) e (d, c). Todos os vértices não-raiz possuem arcos justos entrando: a tem (r, a), b tem (r, b), e tem (b, e), e o conjunto $\{c, d\}$ tem (a, c) (além do ciclo interno). Os arcos (c, d) e (d, c) formam um ciclo justo.

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

zero entrando neles.

Antes de prosseguirmos com a iteração seguinte, precisamos salientar que em uma iteração do algoritmo, pode haver múltiplas fontes no grafo de condensação (excluindo a componente que contém a raiz). Qualquer uma dessas fontes pode ser escolhida como conjunto minimal para modificação de custo. A escolha específica não afeta a corretude do algoritmo, apenas a ordem em que os arcos se tornam justos.

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

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

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

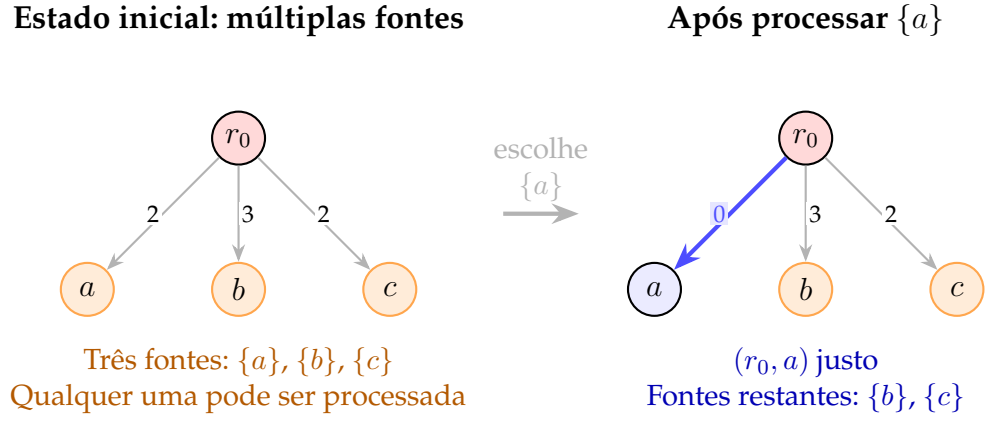


Figura 34 – Exemplo de múltiplas fontes disponíveis para processamento. À esquerda, o estado inicial possui três componentes $\{a\}$, $\{b\}$, $\{c\}$ (em laranja) que são fontes no grafo de condensação. Qualquer uma pode ser escolhida. À direita, após escolher e processar $\{a\}$ (elevando seu potencial por $\Delta(\{a\}) = 2$), o arco (r_0, a) torna-se justo (em azul). Agora $\{a\}$ deixa de ser fonte e as fontes restantes são $\{b\}$ e $\{c\}$. A ordem de processamento não afeta a arborescência ótima final.

No exemplo da Figura 33, após a primeira iteração de redução, temos os vértices $\{a\}$, $\{b\}$, $\{e\}$ que são componentes triviais (cada um forma sua própria CFC). Porém,

o conjunto $\{c, d\}$ forma uma fonte no grafo de condensação e, portanto, um conjunto minimal. Não poderíamos escolher apenas $\{c\}$ ou apenas $\{d\}$ isoladamente, pois esses subconjuntos já possuem arcos justos entrando.

Nesse exemplo da Figura 33, os arcos justos não formam uma r-arborescência, pois nem todos os vértices são alcançáveis a partir de r pelos arcos justos e o conjunto $\{c, d\}$ forma um ciclo com dois arcos justos entre c e d. Temos então as fontes $\{r, a\}$, $\{b\}$, $\{e\}$ e o conjunto $\{c, d\}$. Como não existem arcos justos entrando em $\{c, d\}$, esse conjunto é um subconjunto minimal.

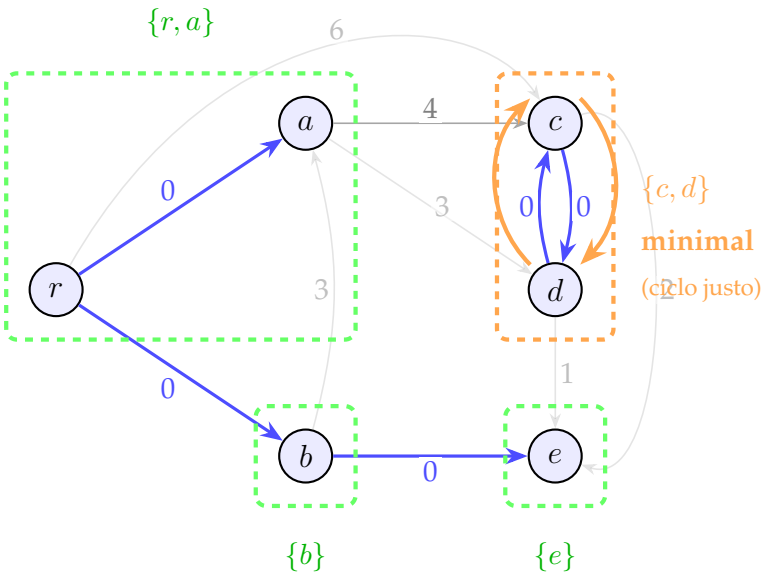


Figura 35 – Identificação de componentes fortemente conexas nos arcos justos após a primeira iteração. As componentes triviais $\{r, a\}$, $\{b\}$ e $\{e\}$ estão em verde. A componente não-trivial $\{c, d\}$ (em laranja) forma um ciclo justo com os arcos (c, d) e (d, c), e é identificada como **minimal** para a próxima iteração. Os arcos justos internos ao ciclo estão destacados, indicando que $\{c, d\}$ deve ser tratado como uma unidade no processo de contração.

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

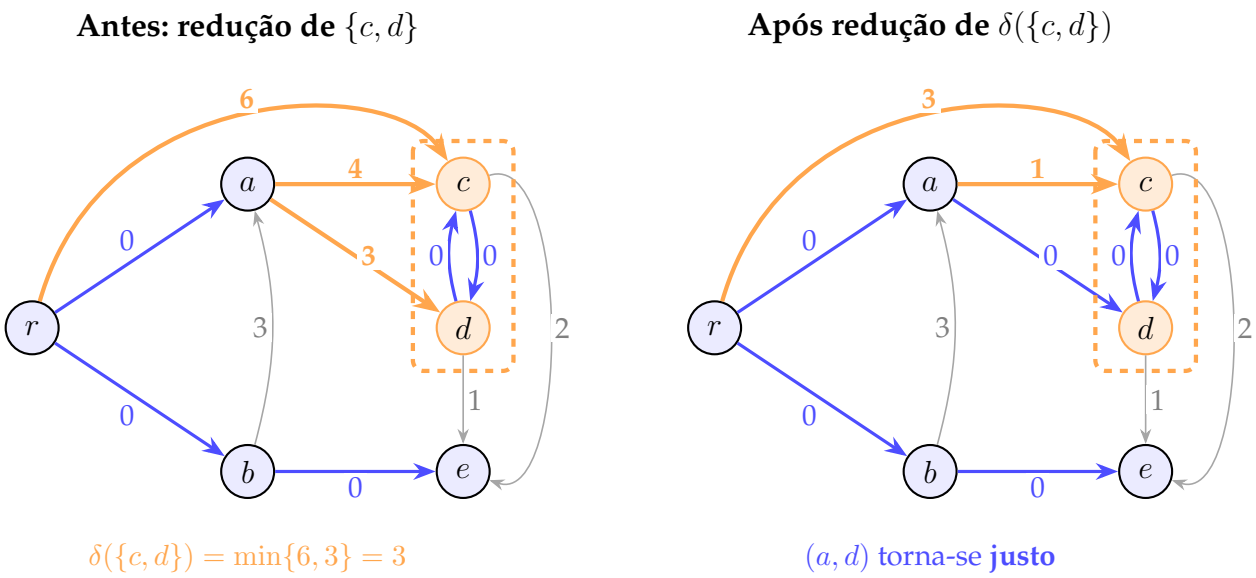


Figura 36 – Redução de custos para o subconjunto minimal $\{c, d\}$. À esquerda, antes da redução: os arcos entrando em $\{c, d\}$ vindos de fora são (r, c) com custo 6, (a, d) com custo 3 e (a, c) com custo 4, destacados em laranja. Calculamos $\delta(\{c, d\}) = 3$ e subtraímos esse valor. À direita, após a redução: (a, d) torna-se justo (custo 0), (r, c) tem custo reduzido para 3 e (a, c) tem custo reduzido para 1. O conjunto $\{c, d\}$ está destacado em laranja para enfatizar que é tratado como uma unidade.

Agora, não existem mais subconjuntos minimais sem arcos justos entrando neles, porém os arcos justos não formam uma r-arborescência, pois existe um ciclo justo entre c e d.

Nesse ponto, o algoritmo procede para a fase de contração, onde o ciclo justo formado pelos arcos (c, d) e (d, c) é contraído em um único vértice. Podemos estender essa ideia de contração de ciclos para abranger a contração de qualquer componente fortemente conexa (CFC) formada por arcos justos.

O algoritmo pode utilizar CFCs para identificar eficientemente os conjuntos minimais. Quando construímos o grafo de condensação das CFCs, cada vértice do grafo de condensação representa uma CFC do grafo original formado pelos arcos justos. Vale destacar que um arco no grafo de condensação conecta duas CFCs se existe um arco justo no grafo original indo de um vértice da primeira CFC para um vértice da segunda e o grafo de condensação é sempre acíclico (DAG).

Uma fonte no grafo de condensação corresponde exatamente aos conjuntos minimais que buscamos.

No exemplo, a CFC $\{c, d\}$ é uma fonte que é contraída em um novo vértice x_C. Todos os arcos que entravam em c ou d agora entram em x_C, e todos os arcos que saíam de c ou d agora saem de x_C. Os custos dos arcos são mantidos conforme estavam antes

da contração.

A Figura 37 ilustra essa contração.

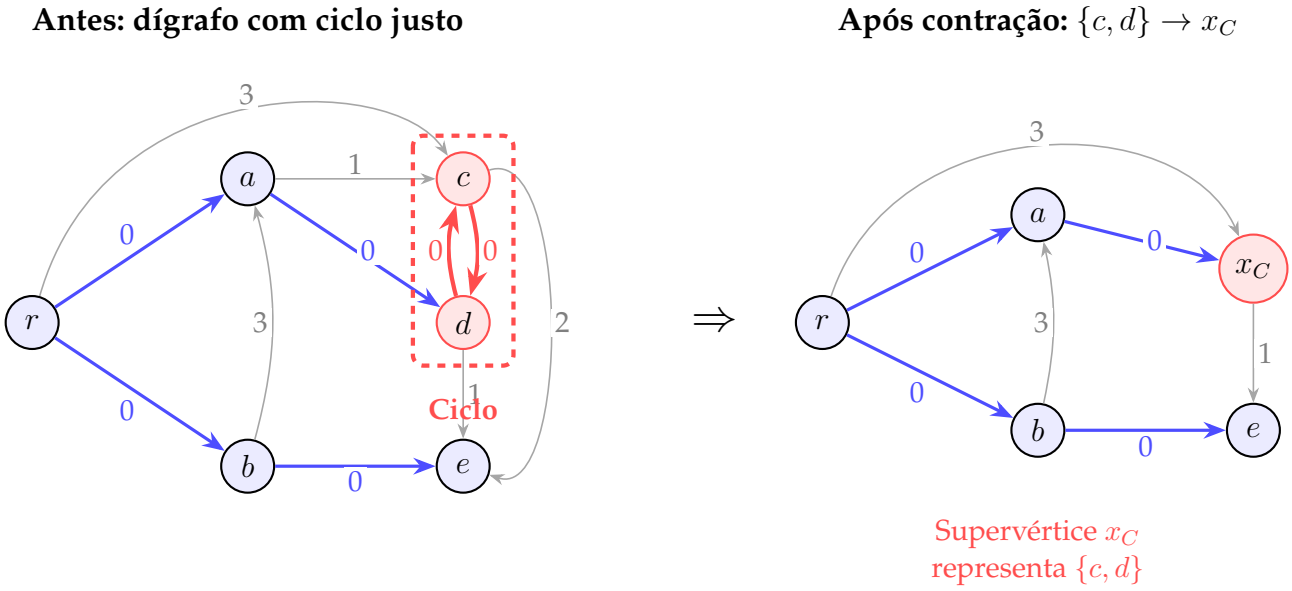


Figura 37 – Contração do ciclo justo $\{c, d\}$. À esquerda, o dígrafo após as reduções de custo mostra o ciclo justo formado pelos arcos (c, d) e (d, c) (em vermelho). À direita, o dígrafo contraído onde os vértices c e d são substituídos pelo supervértice x_C. Os arcos que entravam ou saíam do ciclo são redirecionados para x_C. Note que os arcos justos agora formam uma r-arborescência no dígrafo contraído.

O algoritmo termina quando existe exatamente uma fonte no grafo de condensação das SCCs formadas pelos arcos justos.

Quando essa condição é satisfeita, o grafo (V, A_0) formado pelos arcos justos contém uma r-arborescência, pois todo vértice não-raiz é alcançável a partir de r através de arcos justos (consequência de haver apenas uma fonte).

Assim termina a fase 1 do algoritmo de Frank que pode devolver uma lista de arcos justos A_0 e custos reduzidos $c'(u, v)$ para todos os arcos $(u, v) \in A$ onde uma seleção de $n - 1$ arcos de A_0 formam uma arborescência possivelmente com uma sujeira adicional, no caso do Digrafo do exemplo do acima, A_0 inclui os arcos (c, d) e (d, c) formando um ciclo justo.

Quando A_0 contém um ciclo, a operação de contração (ilustrada na Figura 37) é necessária para eliminar essa ambiguidade. Ao contrair o ciclo em um supervértice e resolver recursivamente, o algoritmo efetivamente escolhe quais arcos do ciclo fazer parte da solução final. Durante a expansão, apenas $|C| - 1$ arcos do ciclo contraído C são incluídos na arborescência (onde $|C|$ é o número de vértices no ciclo), descartando exatamente um arco — aquele que entra no vértice por onde a arborescência alcança o

ciclo.

Outras sujeiras podem ocorrer quando múltiplos arcos justos entram em um mesmo vértice, como no caso de múltiplas fontes sendo processadas em ordens diferentes. A Figura 38 ilustra esse fenômeno no exemplo anterior: após processar os conjuntos $\{a\}$, $\{b\}$ e $\{e\}$, tanto o arco (r, a) quanto o arco (b, a) podem se tornar justos simultaneamente se os custos forem adequados. Nesse caso, A_0 conteria ambos os arcos entrando em a, mas a arborescência final incluirá apenas um deles — o algoritmo escolhe o arco cujo vértice origem já está conectado à raiz na arborescência parcial.

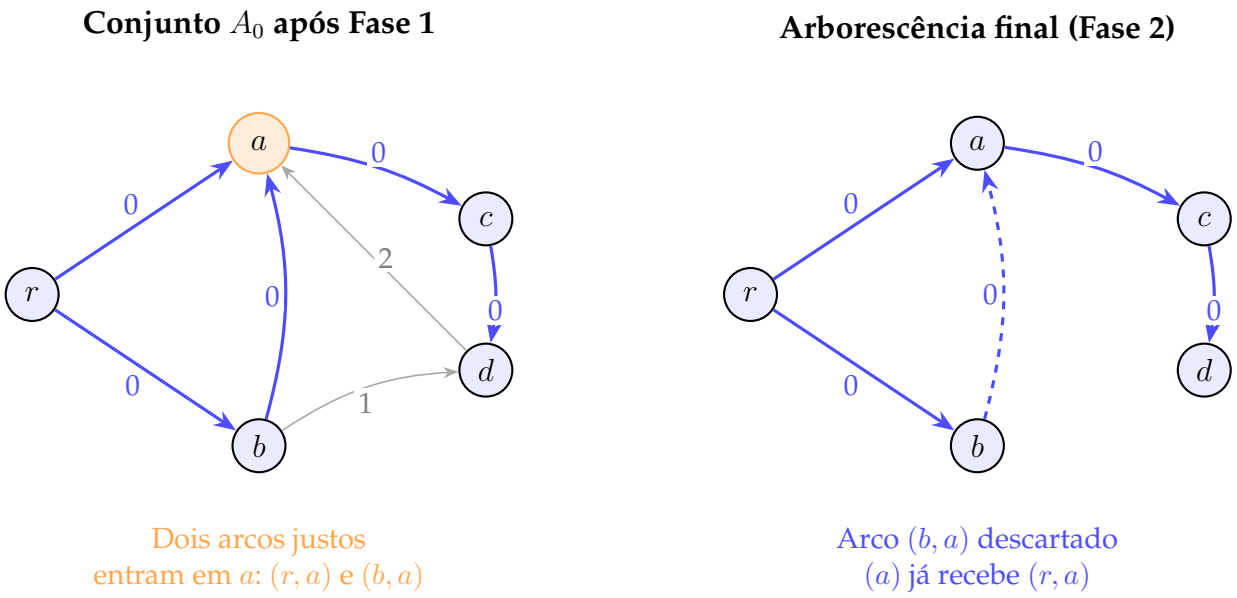


Figura 38 – Exemplo de múltiplos arcos justos entrando em um mesmo vértice. À esquerda, após a Fase 1, o conjunto A_0 contém tanto (r, a) quanto (b, a) como arcos justos (ambos destacados em azul sólido). À direita, durante a construção incremental da arborescência na Fase 2, apenas um arco é escolhido: (r, a) é incluído porque r já está na arborescência parcial, enquanto (b, a) (mostrado tracejado) é descartado, pois a já possui um arco de entrada. A Fase 2 garante que cada vértice não-raiz receba exatamente um arco entrando na arborescência final.

Essa "sujeira" é uma característica do algoritmo. Durante a Fase 1, todos os arcos em A_0 são garantidamente parte de *alguma* arborescência ótima (não necessariamente a mesma), mas apenas um subconjunto de A_0 formará a Ou seja, a Fase 1 do algoritmo de Frank não constrói diretamente a arborescência, mas sim um *certificado de otimalidade* através dos arcos justos e potenciais duais. A Fase 2 então utiliza esse certificado para extrair uma arborescência ótima específica.

Fase 2

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

A Figura 39 ilustra esse processo no exemplo. No dígrafo contraído, a arborescência ótima inclui o arco (a, x_C) entrando no supervértice x_C. Na expansão, esse arco corresponde a (a, d) no dígrafo original. A arborescência final então inclui (a, d) e todos os arcos justos do ciclo exceto o arco que entra em d, ou seja, exclui (c, d) e mantém (d, c). Assim, o vértice c é alcançado através de d.

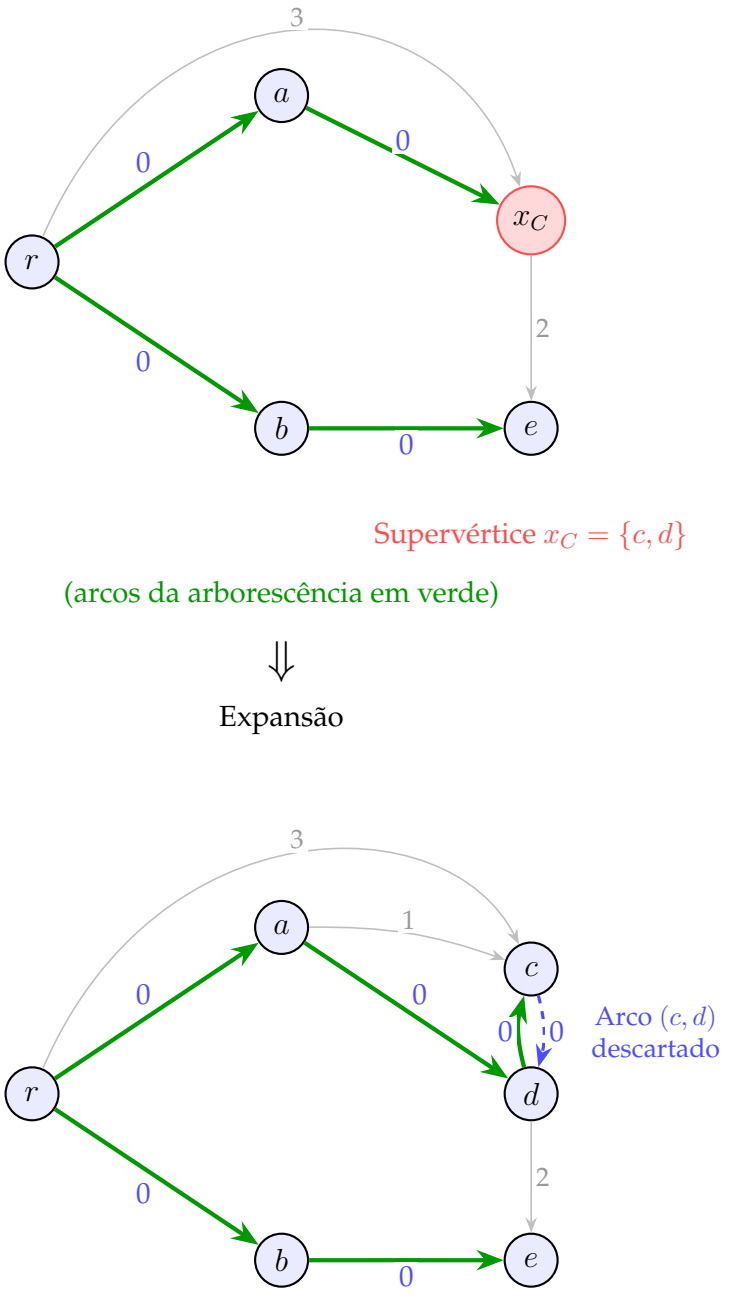


Figura 39 – Fase 2: Expansão do supervértice e construção da arborescência final. No topo, a arborescência ótima no dígrafo contraído (arcos verdes grossos) inclui o arco (a, x_C) entrando no supervértice $x_C = \{c, d\}$. Abaixo, após a expansão no dígrafo original, o arco (a, x_C) é substituído por (a, d), que corresponde ao arco justo que entra no ciclo. O ciclo é "aberto"incluindo apenas $|C| - 1 = 1$ arco interno: (d, c) é incluído na arborescência (verde), enquanto (c, d) é descartado (azul tracejado), pois d já recebe o arco (a, d). O resultado é uma r-arborescência com exatamente $n - 1 = 5$ arcos, todos de custo reduzido zero, portanto ótima.

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

3.2 Descrição do algoritmo

Apresentamos agora uma descrição formal do algoritmo de András Frank.

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

A diferença fundamental em relação ao Chu–Liu–Edmonds está na Fase 1, no algoritmo de Chu–Liu/Edmonds para cada vértice v, subtrai o menor custo entrando em v e processa vértice a vértice, uma vez. Já em Frank, a cada iteração, identifica componentes fortemente conexas no grafo de arcos justos. Para cada componente sem arcos justos entrando (subconjunto minimal X),subtrai $\delta(X)$ de todos os arcos entrando em X. Repete até todos os vértices terem arcos justos entrando.

Algoritmo 3.1: András Frank

Entrada: dígrafo $D = (V, A)$, custos $c : A \to \mathbb{R}_{\geq 0}$, raiz r.[a]
Fase 1: Redução de custos e construção de A_0

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

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

Fase 2: Construção da arborescência

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

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

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

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

3.2.1 Corretude

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

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

   Isso porque, para qualquer r-arborescência F,

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

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

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

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

3.2.2 Complexidade

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

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

O uso de memória é $O(n + m)$, incluindo as estruturas para armazenar o dígrafo, potenciais e componentes. A implementação a seguir adota a versão $O(nm)$ por simplicidade e está disponível no repositório do projeto (<https://github.com/lorenypsum/GraphVisualizer>).

3.3 Implementação em Python

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

A entrada consiste em um dígrafo orientado $D = (V, A)$, com custos dos arcos registrados no atributo `"w"`, e uma raiz $r \in V$. As hipóteses adotadas são: (i) o dígrafo é conexo a partir de r, isto é, todo vértice $v \neq r$ é alcançável a partir da raiz; (ii) para

todo subconjunto $X \subseteq V \setminus \{r\}$, existe ao menos um arco entrando em X; e (iii) todos os custos são não negativos.

A saída é um subdigrafo T de D com $|A_T| = |V| - 1$ arcos, tal que cada vértice $v \neq r$ possui grau de entrada igual a 1, todos os vértices são alcançáveis a partir de r, e o custo total $\sum_{a \in A_T} c(a)$ é mínimo.

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

3.3.1 Identificação de arcos entrando em conjunto X

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

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

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

Identificação de arcos entrando em conjunto X

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

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

A figura a seguir ilustra o funcionamento da função `get_arcs_entering_X` em

um digrafo que vamos denotar por D_{32}. O dígrafo possui uma raiz r_0 conectada aos vértices u_1, u_2, u_3. Os vértices em laranja pertencem ao conjunto $X = \{v_1, v_2, v_3\}$, e a função identifica apenas os arcos em vermelho, que saem de vértices fora de X e entram em vértices dentro de X. Arcos da raiz, arcos internos a X, externos a X, ou saindo de X não são retornados.

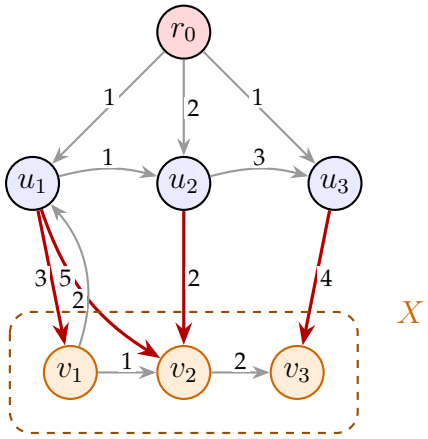


Figura 40 – Ilustração da função `get_arcs_entering_X` em D_{32}. A raiz r_0 (em vermelho claro) conecta-se aos vértices u_1, u_2, u_3. Os vértices em laranja pertencem ao conjunto $X = \{v_1, v_2, v_3\}$. A função identifica apenas os arcos em vermelho: aqueles que saem de vértices fora de X e entram em vértices dentro de X. Arcos da raiz, arcos internos a X, externos a X, ou saindo de X não são retornados.

3.3.2 Cálculo do peso mínimo de corte

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

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

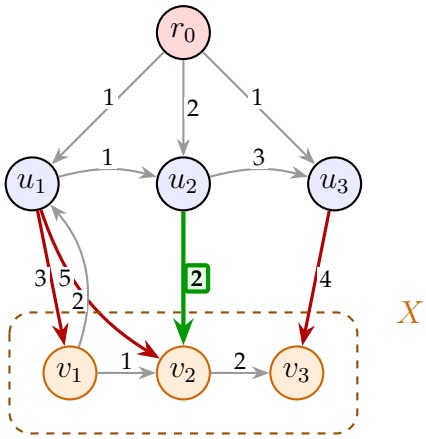
A função devolve o peso mínimo encontrado entre todos os arcos da lista. A complexidade é $O(k)$, onde k é o número de arcos na lista, pois examina cada arco uma vez para encontrar o mínimo.

Cálculo do peso mínimo de corte

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

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

A seguir temos uma ilustração do funcionamento da função `get_minimum_weight_cut` em D_{32}. Considerando os arcos em vermelho que entram em X (identificados pela função anterior), esta função calcula o peso mínimo entre eles. O arco em verde possui o menor peso (2), correspondendo ao valor $\Delta(X) = 2$ que será devolvido pela função.



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

Figura 41 – Ilustração da função `get_minimum_weight_cut` em D_{32}. Considerando os arcos em vermelho que entram em X (identificados pela função anterior), esta função calcula o peso mínimo entre eles. O arco em verde possui o menor peso (2), correspondendo ao valor $\Delta(X) = 2$.

3.3.3 Atualização de pesos em X

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

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

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

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

Atualização de pesos em X

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

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

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

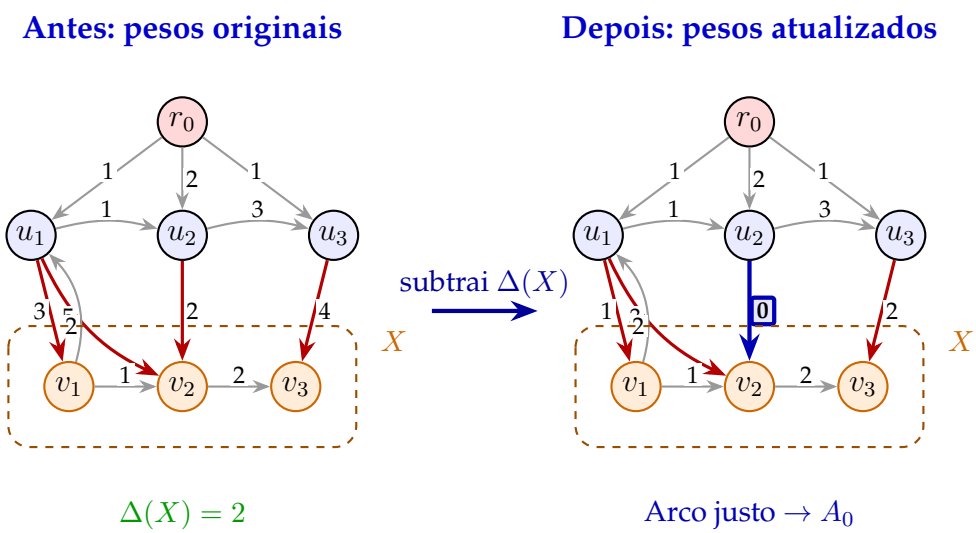


Figura 42 – Ilustração da função `update_weights_in_X` em D_{32}. À esquerda, o dígrafo antes da atualização, com os arcos em vermelho entrando em X e $\Delta(X) = 2$. À direita, após subtrair $\Delta(X)$ de cada arco entrando em X: o peso (u_1, v_1) reduz de 3 para 1, (u_2, v_2) de 2 para **0** (torna-se justo), (u_3, v_3) de 4 para 2, e (u_1, v_2) de 5 para 3. O arco justo é adicionado a A_0 e D_0. Note que os arcos da raiz e arcos internos/externos a X permanecem inalterados.

3.3.4 Verificação de arborescência

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

Recebe como entrada um dígrafo `D` e a raiz `r0`. A implementação constrói uma árvore DFS a partir de `r0` usando `nx.dfs_tree` (linha 2), que devolve um subdigrafo contendo apenas os vértices alcançáveis a partir da raiz seguindo arcos. Em seguida, compara o número de vértices da árvore DFS com o número total de vértices de `D` (linha 3).

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

Verificação de arborescência

Verifica se o dígrafo D contém uma r-arborescência com raiz r0, usando busca em profundidade para testar alcançabilidade de todos os vértices.

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

3.3.5 Fase 1: Elevação de potenciais e construção de A_0

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

Recebe como entrada um dígrafo `D_original` e a raiz `r0`. A implementação cria uma cópia do dígrafo original (linha 2) para preservar a entrada, inicializa estruturas auxiliares `A_zero` (lista de arcos justos), `Dual_list` (lista de pares $(X, \Delta(X))$ para fins de validação dual), e `D_zero` (dígrafo de arcos justos) (linhas 3-5). Um contador de iterações é inicializado na linha 6.

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

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

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

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

Fase 1: Elevação de potenciais e construção de A_0

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

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

As funções auxiliares implementadas nesta seção correspondem diretamente aos

passos da Fase 1 do algoritmo de András Frank da seguinte forma:

Correspondência entre Teoria e Implementação — Fase 1

Passo 1 — Inicialização:

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

Passo 2 — Iteração de elevação de potenciais:

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

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

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

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

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

Passo 3 — Construção de A_0:

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

3.3.6 Fase 2: Construção da arborescência

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

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

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

1. Percorre os arcos (u, v) de `A_zero` (linha 6).
2. Verifica se u já está em `Arb` e v ainda não (linha 7).
3. Se sim, obtém os dados do arco do dígrafo original (linha 8) e adiciona (u, v) a `Arb` (linha 9).

4. Interrompe o loop interno para reiniciar a busca, garantindo descoberta em largura (linha 10).

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

Fase 2: Construção da arborescência

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

```
1  def phase2(D_original, r0, A_zero):
2      Arb = nx.DiGraph()
3      Arb.add_node(r0)
4      n = len(D_original.nodes())
5      for _ in range(n - 1):
6          for u, v in A_zero:
7              if u in Arb.nodes() and v not in Arb.nodes():
8                  edge_data = D_original.get_edge_data(u, v)
9                  Arb.add_edge(u, v, **edge_data)
10                 break
11     return Arb
```

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

Recebe como entrada um dígrafo `D_original`, a raiz `r0`, e a lista `A_zero` de arcos justos. A implementação começa criando um dígrafo auxiliar `Arb` (linha 2) onde cada arco de `A_zero` recebe um peso igual ao seu índice na lista (linhas 3-4), estabelecendo uma ordem de exploração. Inicializa-se o conjunto `V` de vértices visitados contendo apenas a raiz (linha 5) e uma fila de prioridade vazia `q` (linha 6).

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

O loop principal (linhas 10-17) extrai arcos da fila de prioridade em ordem crescente de índice. Para cada arco (u, v) extraído (linha 11), verifica-se se o vértice destino v já foi visitado (linha 12); em caso positivo, o arco é ignorado via `continue`

(linha 13). Caso contrário, adiciona-se o arco (u, v) à arborescência `A` com o peso original de `D_original` (linha 14), marca-se v como visitado (linha 15), e todos os arcos que saem de v em `Arb` são adicionados à fila de prioridade para exploração futura (linhas 16-17).

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

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

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

```
1  def phase2_v2(D_original, r0, A_zero):
2      Arb = nx.DiGraph()
3      for i, (u, v) in enumerate(A_zero):
4          Arb.add_edge(u, v, w=i)
5      V = {r0}
6      q = []
7      for u, v, data in Arb.out_edges(r0, data=True):
8          heapq.heappush(q, (data["w"], u, v))
9      A = nx.DiGraph()
10     while q:
11         _, u, v = heapq.heappop(q)
12         if v in V:
13             continue
14         A.add_edge(u, v, w=D_original[u][v]["w"])
15         V.add(v)
16         for x, y, data in Arb.out_edges(v, data=True):
17             heapq.heappush(q, (data["w"], x, y))
18     return A
```

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

Correspondência entre Teoria e Implementação — Fase 2

Passo 4 — Construção da arborescência (caso acíclico):

- **Descrição teórica:** Se (V, A_0) forma uma r-arborescência, devolva A_0. Por otimalidade dos potenciais duais, trata-se de uma r-arborescência de custo mínimo.

- **Implementação:** Ambas as versões de `phase2` constroem uma r-arborescência a partir do conjunto A_0 de arcos justos obtido na Fase 1. A corretude baseia-se no fato de que todos os arcos em A_0 têm custo reduzido zero, e a Fase 1 garante que existe uma r-arborescência formada exclusivamente por arcos justos.
  - **Versão 1 (`phase2`):** Construção incremental por exploração exaustiva. Em cada uma das $n-1$ iterações, percorre todos os arcos de A_0 procurando um arco (u, v) tal que u já pertence à arborescência parcial e v ainda não. Essa abordagem simples corresponde diretamente à ideia teórica de construir a arborescência adicionando um vértice por vez, conectando-o à estrutura existente através de um arco justo. Complexidade: $O(nm)$.
  - **Versão 2 (`phase2_v2`):** Construção por busca em largura guiada por fila de prioridade. Cria um dígrafo auxiliar onde arcos são indexados, usa a fila de prioridade para explorar arcos sistematicamente a partir da raiz, evitando buscas lineares repetidas. Essa versão otimizada mantém a mesma correção teórica — construir uma r-arborescência usando apenas arcos de A_0 — mas melhora a eficiência prática. Complexidade: $O(m \log m)$.

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

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

3.3.7 Verificação de otimalidade dual

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

Recebe como entrada a arborescência `Arb` e a lista `Dual_list` contendo pares $(X, \Delta(X))$ onde X é um conjunto de vértices cujos potenciais foram elevados e $\Delta(X) > 0$ é o valor da elevação. A implementação itera sobre cada par (X, z) na lista dual (linha 2), e para cada conjunto X, percorre todos os arcos (u, v) da arborescência (linha 3). Inicializa um contador `count` em zero (linha 4) e verifica se o arco cruza a fronteira de X, isto é, se $u \notin X$ e $v \in X$ (linha 5). Quando essa condição é satisfeita, incrementa o contador (linha 6) e imediatamente verifica se já há mais de um arco entrando em X (linha 7). Caso positivo, a condição de otimalidade dual é violada e a função devolve `False` (linha 8).

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

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

Verificação de otimalidade dual

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

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

3.3.8 O algoritmo completo de András Frank

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

Verificação de otimalidade dual

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

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

Fase 2 — Construção da arborescência:

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

Verificação de otimalidade dual:

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

3.3.9 Correspondência entre teoria e implementação

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

Descrição Teórica	Implementação Python

Passo 1: Inicialização
Defina $y(v) := 0$ para todo $v \in V$.
Inicialize $A_0 := \emptyset$.
Construa dígrafo vazio D_0 (arcos justos).
	Função `phase1` — Linhas 2–5:
	`D_copy = D_original.copy()`
	`A_zero = []`
	`D_zero = build_D_zero(D_copy)`
	Potenciais $y(v) = 0$ implícitos, custos $c_y = c$.

Passo 2: Elevação de potenciais
Enquanto $\exists X \subseteq V \setminus \{r\}$ sem arco justo:
Calcule $\Delta(X) := \min\{c_y(u,v) : u \notin X, v \in X\}$
Atualize $y(v) := y(v) + \Delta(X), \forall v \in X$
Adicione arcos com $c_y = 0$ a A_0
	Loop principal — Linhas 7–22:
	`C = nx.condensation(D_zero)`
	`sources = [x for x in C.nodes()]`
	`  if C.in_degree(x) == 0]`
	Para fonte u (exceto raiz):
	`  X = C.nodes[u]["members"]`
	`  arcs = get_arcs_entering_X(D, X)`
	`  min_w = get_minimum_weight_cut(arcs)`
	`  update_weights_in_X(D, arcs, min_w,`
	`  A_zero, D_zero)`

Passo 2(a): Identificar arcos entrando
Determine $\{(u,v) \in A : u \notin X, v \in X\}$
	Função `get_arcs_entering_X`:
	`return [(u, v, data)`
	`  for u, v, data in D.edges(data=True)`
	`  if u not in X and v in X]`

Passo 2(b): Calcular $\Delta(X)$
$\Delta(X) := \min\{c_y(u,v) : u \notin X, v \in X\}$
	Função `get_minimum_weight_cut`:
	`return min(data["w"]`
	`  for _, _, data in arcs)`

Passo 2(c): Atualizar pesos
Para (u,v) entrando em X:
$c_y(u,v) := c_y(u,v) - \Delta(X)$
Se $c_y(u,v) = 0$, adicione a A_0
	Função `update_weights_in_X`:
	`for u, v, _ in arcs:`
	`  D[u][v]["w"] -= min_weight`
	`  if D[u][v]["w"] == 0:`
	`  A_zero.append((u, v))`
	`  D_zero.add_edge(u, v)`

Passo 3: Verificar término Se D_0 contém r-arborescência, encerre.	**Condição — Linhas 11–12:** `if len(sources) == 1: break` Uma fonte $\Rightarrow$ r-arborescência acíclica.

Passo 4: Construir arborescência
Construa F a partir de A_0, conectando vértices incrementalmente.
	Função `phase2` (incremental):
	`Arb = nx.DiGraph(); Arb.add_node(r0)`
	`for _ in range(n - 1):`
	`  for u, v in A_zero:`
	`  if u in Arb and v not in Arb:`
	`  Arb.add_edge(u, v, **data)`
	`  break`
	Complexidade: $O(nm)$.

Passo 4: Versão otimizada
Mesma ideia, com fila de prioridade.
	Função `phase2_v2` (BFS):
	`Arb = nx.DiGraph()`
	`for i, (u, v) in enumerate(A_zero):`
	`  Arb.add_edge(u, v, w=i)`
	`q = [] # fila de prioridade`
	`while q:`
	`  _, u, v = heapq.heappop(q)`
	`  if v in V: continue`
	`  A.add_edge(u, v, w=D[u][v]["w"])`
	Complexidade: $O(m \log m)$.

Otimalidade dual
Para cada X elevado ($\Delta(X) > 0$), exatamente um arco de F cruza $\delta^-(X)$.
	Função `check_dual_optimality`:
	`for X, z in Dual_list:`
	`  count = 0`
	`  for u, v in Arb.edges():`
	`  if u not in X and v in X:`
	`  count += 1`
	`  if count > 1: return False`
	`return True`

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

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

4 Chu-liu / Edmonds vs. Frank

Neste capítulo, apresentamos uma análise comparativa entre os algoritmos de Chu–Liu–Edmonds e András Frank para o problema da arborescência de custo mínimo. Ambos produzem soluções ótimas, mas diferem em sua organização: Chu–Liu–Edmonds opera vértice a vértice com contrações imediatas de ciclos, enquanto Frank identifica subconjuntos minimais via componentes fortemente conexas, processando múltiplos vértices simultaneamente em duas fases distintas. O objetivo é elucidar essas diferenças e avaliar empiricamente seus desempenhos.

O algoritmo de Chu–Liu–Edmonds (CHU; LIU, 1965; EDMONDS, 1967) opera recursivamente selecionando para cada vértice $v \neq r$ um arco de entrada de custo mínimo, contraindo ciclos detectados e ajustando custos, até eliminar todos os ciclos.

O algoritmo de Frank (FRANK, 1981; FRANK; HAJDU, 2014) também reduz custos subtraindo o mínimo de entrada, mas identifica *subconjuntos minimais* via componentes fortemente conexas, processando múltiplos vértices simultaneamente. Opera em duas fases: (i) redução até criar arcos de custo reduzido zero e (ii) construção da arborescência a partir desses arcos.

Ambos contraem ciclos detectados e expandem ao final, com número de contrações limitado por $O(n)$ (SCHRIJVER, 2003).

Logo, os métodos são equivalentes e resolvem o problema de arborescência de custo mínimo, mas adotam estratégias distintas na redução de custos e na construção da solução e nessa sessão experimental buscamos avaliar essas diferenças na prática.

4.1 Análise comparativa dos algoritmos

Conduzimos 2000 experimentos em digrafos enraizados aleatórios com $|V| \in [100, 200]$ vértices, densidade $|A| \in [n, 3n]$ arcos e pesos inteiros em $[1, 20]$, garantindo conectividade por construção incremental (SCHRIJVER, 2003). Para cada instância, executamos Chu–Liu–Edmonds e as duas fases de Frank (com variantes v1 e v2 da Fase II), registrando custos, tempos e métricas estruturais.

Os 2000 testes confirmaram que todos os métodos retornam sempre o mesmo custo ótimo, com 100% de sucesso, validando a corretude das implementações e a equivalência teórica entre Chu–Liu/Edmonds e Frank (FRANK; HAJDU, 2014; SCHRIJVER, 2003). Além disso, as verificações da condição de otimalidade dual foram bem-sucedidas em todos os casos para ambas as variantes de Frank.

Quanto ao desempenho temporal, observamos que Chu–Liu/Edmonds apresenta desempenho computacional superior nas instâncias testadas, com tempo mediano significativamente menor que a Fase I de Frank. A Fase I, responsável pela identificação de subconjuntos minimais e redução de custos, domina o tempo total de execução do método de Frank, enquanto as Fases II representam uma fração residual do processamento.

A comparação entre as duas variantes da Fase II revela ganho expressivo com o uso de *heap* (fila de prioridade). A versão v2 apresenta aceleração (*speedup*) consistente sobre a versão v1, confirmando empiricamente a vantagem da estrutura de dados com complexidade $O(\log n)$ versus $O(n)$ por operação de seleção de mínimo.

As métricas estruturais mostram que o número de contrações em Chu–Liu/Edmonds é pequeno (mediana de 1 e média de 2,29), muito abaixo do limite teórico $O(n)$ (SCHRIJVER, 2003), o que pode indicar que digrafos aleatórios com distribuição uniforme raramente apresentam estruturas cíclicas complexas. O subgrafo D_0 produzido pela Fase I cresce linearmente com o número de vértices, confirmando a proporcionalidade $|A_0| = O(|V|)$ prevista teoricamente.

O consumo de memória na Fase I mantém-se modesto em todas as instâncias testadas, viabilizando a aplicação dos algoritmos mesmo em ambientes com recursos limitados.

As Figuras 43–48 apresentam os resultados experimentais.

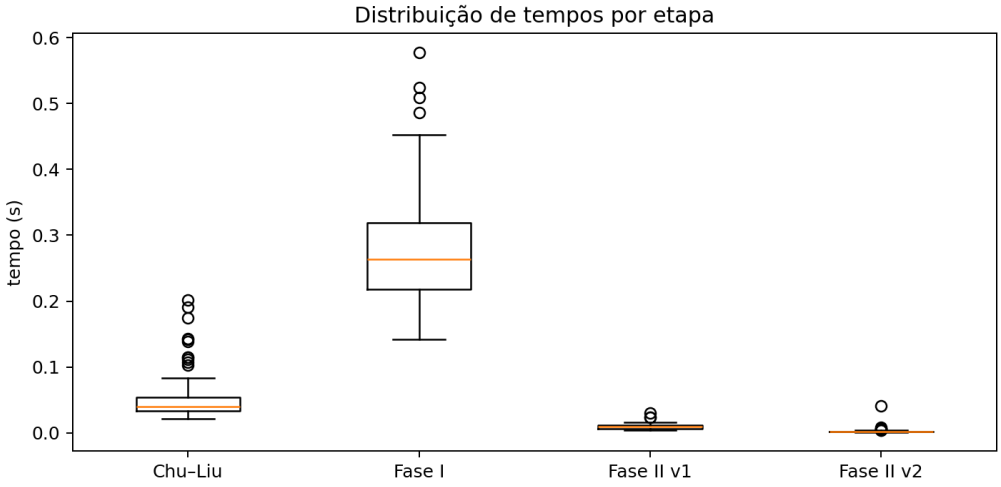


Figura 43 – Distribuição de tempos: Fase I apresenta maior mediana ($0{,}084$ s) e variabilidade.

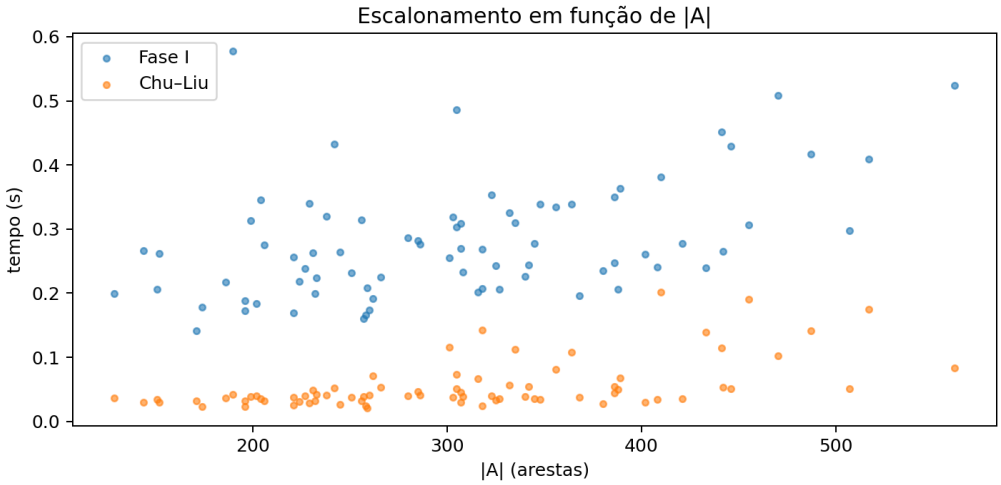


Figura 44 – Escalonamento temporal em função de $|A|$: crescimento aproximadamente linear.

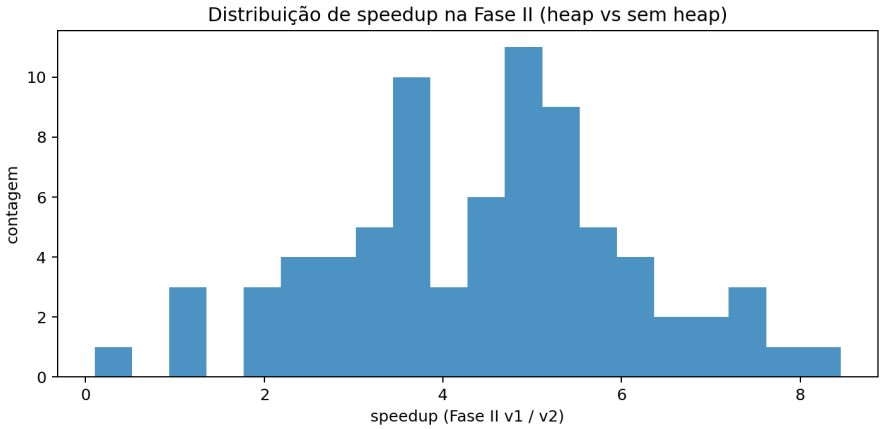


Figura 45 – Aceleração na Fase II: v2 (*heap*) $4{,}59\times$ mais rápida que v1.

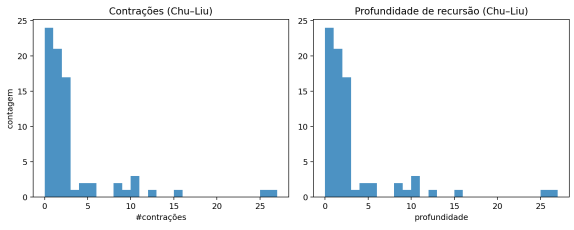


Figura 46 – Contrações e profundidade em Chu–Liu: mediana 1, média $2{,}29$.

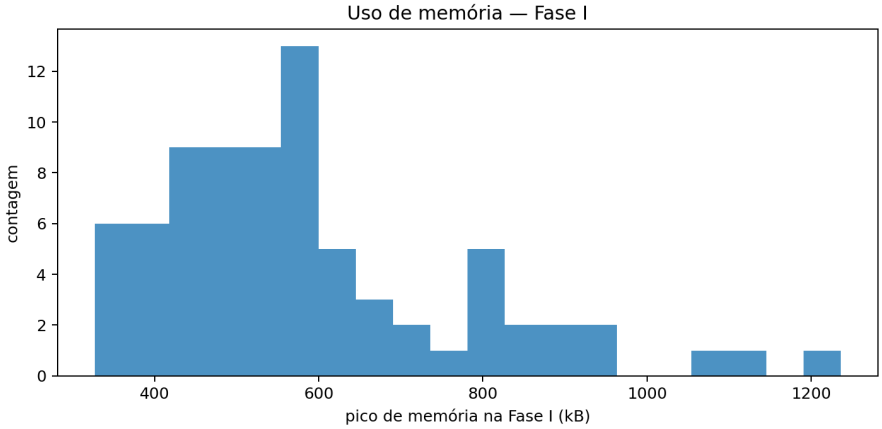


Figura 47 – Pico de memória na Fase I: mediana 539 kB.

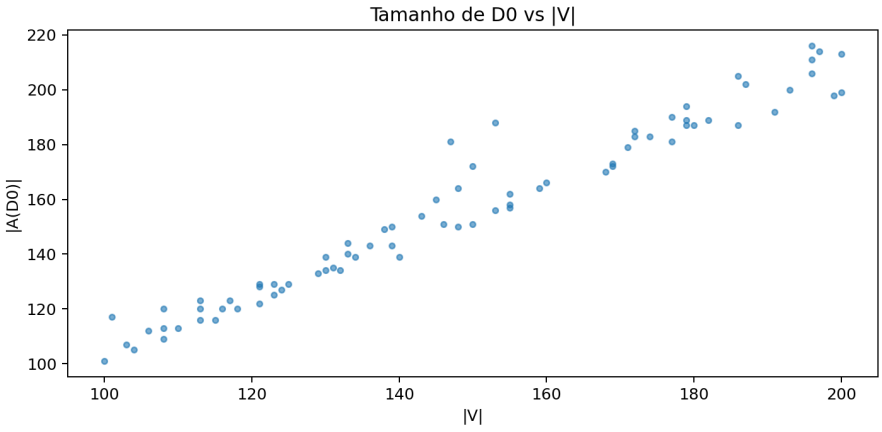


Figura 48 – Tamanho de D_0 versus $|V|$: relação linear confirma $|A_0| = O(|V|)$.

4.2 Conclusões

Os experimentos validam empiricamente as previsões teóricas e revelam características importantes do comportamento prático dos algoritmos. A equivalência de custos em 100% das instâncias, somada à verificação bem-sucedida das condições de otimalidade dual, confirma a corretude das implementações.

Chu–Liu/Edmonds demonstra-se mais eficiente para construção direta de arborescências nas instâncias aleatórias testadas, enquanto o método de Frank apresenta overhead na Fase I devido ao processamento de subconjuntos minimais via componentes fortemente conexas. A versão heap da Fase II, por sua vez, valida os ganhos assintóticos esperados, demonstrando que melhorias algorítmicas fundamentais se traduzem em benefícios práticos mensuráveis.

O comportamento em digrafos aleatórios é significativamente melhor que os limites teóricos de pior caso, especialmente quanto ao número de contrações. O consumo

modesto de memória e a escalabilidade observada viabilizam a aplicação prática de ambos os métodos em contextos com recursos computacionais limitados.

Compreender os algoritmos teoricamente e validá-los empiricamente é fundamental, mas como transformar esse conhecimento em aprendizagem efetiva? Desenvolvemos uma aplicação *web* que permite acompanhar passo a passo o funcionamento de ambos os algoritmos de forma visual e interativa. O próximo capítulo discute os fundamentos didáticos que orientaram esse design.

5 A Didática do Abstrato

Thomás de Aquino, em sua obra *De veritate*, argumenta que o conhecimento humano começa com a percepção sensorial do mundo concreto, mas alcança sua plenitude ao transcender o particular e abraçar o universal através da abstração. Esse processo de abstração é fundamental para a matemática e a ciência da computação, onde conceitos complexos são frequentemente representados por meio de símbolos e estruturas que vão além da experiência direta.

Grafos e digrafos são simultaneamente concretos (nós e arestas) e abstratos (propriedades globais como cortes, conectividade, laminaridade). Essas noções exigem transitar entre níveis de representação (intuitivo, visual, simbólico, formal) (TALL, 1991), o que pode ser desafiador. A abstração é poderosa, mas também pode ser uma barreira: conceitos como "corte ativo" ou "complementaridade primal–dual" são difíceis de visualizar e internalizar sem apoio didático adequado.

Então, como ensinar e aprender conceitos abstratos de forma eficaz? O ensino de matemática no ensino superior, especialmente em áreas como teoria dos grafos parecem sofrer com dificuldades específicas. A seguir, discutimos essas dificuldades e como o uso de ferramentas visuais e interativas pode ajudar a superá-las.

5.0.1 Fundamentos cognitivos e didáticos

O ensino de matemática no ensino superior exige transitar entre registros de representação (intuitivo, visual, simbólico, formal) com intencionalidade didática (TALL, 1991). À luz da teoria da carga cognitiva, é útil distinguir: (i) a *carga intrínseca*, determinada pela complexidade dos esquemas a construir e pelos pré-requisitos ativados; (ii) a *carga extrínseca*, criada pela forma de apresentação; e (iii) a *carga pertinente* (*germane*), isto é, o esforço dedicado à organização e automatização de esquemas (SWELLER, 1988). Em cursos avançados, a extrínseca cresce quando definições, símbolos e figuras não são co-referenciados no tempo e no espaço, dificultando a coordenação entre o que se lê, o que se vê e o que se infere.

5.0.2 Desafios centrais

Aprender conteúdos de alta abstração envolve lidar com sobrecarga cognitiva intrínseca e extrínseca (SWELLER, 1988). Diretrizes de aprendizagem multimídia indicam que combinar representações verbais e visuais pode reduzir carga desnecessária e favorecer integração semântica (MAYER, 2009; PAIVIO, 1990). Em matemática avançada, a transição entre níveis de representação (intuitivo, formal, simbólico) exige mediação

cuidadosa (TALL, 1991) e atenção a como exemplos, contraexemplos e invariantes são apresentados.

No caso específico de algoritmos com provas baseadas em complementaridade primal–dual, é frequente que estudantes compreendam os passos operacionais sem internalizar a estrutura teórica que garante correção e otimalidade.

5.0.3 Lidando com grafos e digrafos

Na prática, o que mais dificulta o ensino de digrafos não é definir vértices e arcos, mas articular o que fazemos localmente com as estruturas globais que sustentam as provas de correção e de otimalidade em arborescências de custo mínimo — em particular, nos métodos de Chu–Liu/Edmonds e de Frank —: cortes ativos, componentes fortemente conexas (SCCs) e famílias laminares de conjuntos (BONDY; MURTY, 2008; DIESTEL, 2017; WEST, 2001). Quando essa articulação não aparece, cresce a *carga intrínseca* (múltiplas dependências simultâneas) e também a *carga extrínseca* (o esforço de alinhar texto, fórmulas e figuras). Nesse contexto específico destacamos três desafios didáticos:

- **Articular o local com o global:** escolher a melhor aresta de entrada para cada vértice não garante coerência global; isso pode criar ciclos. Ver o grafo *condensado* em SCCs (cada ciclo vira um "bloco") torna esse efeito visível e manipulável (Fig. 49). *Dificuldade típica:* estudantes tendem a projetar a heurística local para o todo e se surpreendem com ciclos "inesperados" — uma fonte comum de sobrecarga por conflito entre intuições locais e restrições globais.
- **Acompanhar efeitos de contração/expansão:** contrair um ciclo (substituílo por um supervértice) e depois reexpandir impacta os custos reduzidos c' e os cortes que ficam "ativos". A laminaridade — cortes aninhados, sem interseções conflitantes — fornece uma geometria simples para seguir essas mudanças (Fig. **??**). *Dificuldade típica:* perder o fio entre representações (grafo original, condensado, reexpansão) aumenta a carga extrínseca; sinalizar as diferenças em cada etapa reduz esse atrito.
- **Mapear o "fazer do algoritmo" para a linguagem primal–dual:** ações como elevar potenciais, escolher entradas e contrair ciclos correspondem a garantias teóricas. Em particular, *apertude* (custo reduzido zero) e *complementaridade* (exatamente uma aresta entra em cada conjunto ativo) certificam a otimalidade. Relacionar essas ideias às métricas que coletamos (tempos, número de contrações, pico de memória) ajuda a ligar prática e teoria, via custos reduzidos e reexpansão (Figs. **??** e **??**). *Dificuldade típica:* executar os passos sem ver como eles tornam certas restrições "justas" dificulta a internalização do porquê; explicitar os vínculos entre ação e certificação reduz a distância entre o operacional e o conceitual.

No exemplo da figura abaixo, a condensação do digrafo D em D_0 torna visível a relação entre escolhas locais (entradas por vértice) e estrutura global (ciclos, cortes). Cada SCC (bloco) pode ser tratado como uma unidade, facilitando a compreensão de como ciclos surgem e são resolvidos. Apesar de não ser suficiente para ilustrar situações mais complexas, essa visualização ajuda criar intuição sobre a atualização de custos reduzidos e a dinâmica de contração/expansão.

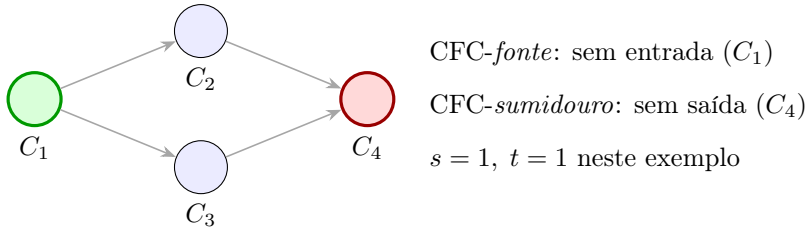


Figura 49 – Condensação de D_0 e fontes do DAG: ver o grafo "como" blocos (SCCs) ajuda a articular o local (entradas por vértice) com o global (cortes e contrações).

5.0.4 Visualização e interação: princípios em uso

Há evidências de que diagramas e animações, quando bem projetados, podem acelerar a compreensão de relações topológicas e causais (LARKIN; SIMON, 1987; WARE, 2012).

A teoria da carga cognitiva sugere que combinar representações verbais e visuais pode reduzir carga extrínseca e favorecer integração semântica (MAYER, 2009; PAIVIO, 1990). Diretrizes de aprendizagem multimídia recomendam evitar excesso de elementos visuais que não contribuam para o entendimento (reduzindo carga extrínseca) e alinhar texto e imagens no tempo e no espaço (reduzindo esforço de coordenação) (MAYER, 2009).

No campo específico de matemática avançada, Tall enfatiza a coordenação entre registros — intuitivo, visual, simbólico e formal — como motor da passagem do pensamento predominantemente procedimental para o conceitual (TALL, 1991). Diagramas não são meros adornos: estruturam inferências espaciais e relacionais de modo mais eficiente que sentenças lineares (LARKIN; SIMON, 1987).

De modo convergente, pesquisas em educação em ciência da computação apontam que visualizações de algoritmos só se traduzem em melhora de aprendizagem quando ativam processos mentais do estudante, promovendo previsão, manipulação e explicação, em vez de mera observação passiva (HUNDHAUSEN et al., 2002; NAPS et al., 2003). Assim, tanto na matemática quanto na computação, o poder das visualizações reside menos no formato gráfico em si e mais na forma como elas integram e articulam o raciocínio.

5.0.5 Disseminação de conteúdos avançados: o ecossistema de ferramentas

Materiais que conectam teoria, evidências empíricas e interatividade têm maior potencial de transferência e retenção.

Tendo em vista esses fundamentos, ferramentas digitais podem ajudar a reduzir carga extrínseca e a integrar registros (visual, simbólico, formal) quando a interação é desenhada para promover *engajamento ativo* (MAYER, 2009; SWELLER, 1988; HUNDHAUSEN et al., 2002; NAPS et al., 2003). A seguir, apresentamos categorias de ferramentas úteis no ensino de grafos, indicando finalidades, forças e limitações, e posicionamos a nossa aplicação nesse ecossistema.

5.0.6 Ferramentas didáticas no ensino de teoria dos grafos

Várias ferramentas digitais podem apoiar o ensino de grafos e digrafos, cada uma com forças e limitações específicas. A seguir, discutimos quatro categorias principais: (i) diagramas programáveis e tipografia matemática, (ii) exploração e edição de grafos, (iii) visualização de algoritmos, e (iv) ambientes programáveis e reprodutibilidade.

Algumas ferramentas permitem criar diagramas de grafos com semântica visual consistente, integrando-os a textos matemáticos. Essas ferramentas são úteis para ilustrar conceitos, definições e provas em materiais didáticos.

Ambientes como Graphviz/dot e TikZ/PGF permitem especificar grafos declarativamente e gerar figuras reprodutíveis com layouts consistentes (GANSNER; NORTH, ; TANTAU, 2015). Benefícios didáticos: (i) semântica visual estável (mesmo conceito, mesma forma), (ii) autoria próxima ao símbolo e ao texto (co-referência), (iii) manutenção e versionamento fáceis. Limitações: a interação costuma ser offline (figuras estáticas) e a curva de aprendizado de sintaxe pode ser um obstáculo inicial. Em contextos de prova e definição, esses recursos ancoram a narrativa formal com diagramas que obedecem às diretrizes de (LARKIN; SIMON, 1987; WARE, 2012).

Contudo, diagramas estáticos não capturam a dinâmica de algoritmos que envolvem mudanças estruturais (elevação de potenciais, contração/expansão, seleção de arestas). Para isso, são necessárias ferramentas interativas que permitam explorar essas transformações em tempo real.

Ferramentas de exploração e edição de grafos permitem que os usuários interajam com representações gráficas de dados, facilitando a manipulação e a análise de estruturas complexas. Essas ferramentas são essenciais para atividades que exigem uma compreensão profunda das relações entre os elementos de um grafo.

Dentre elas destacam-se Gephi, yEd e Cytoscape, que oferecem layouts automáticos, filtros e medidas de rede (BASTIAN et al., 2009; YED. . . , ; SHANNON et al., 2003).

São adequadas para: (i) reconhecer padrões estruturais (componentes, comunidades), (ii) discutir implicações de layouts para percepção de estruturas, (iii) atividades de descoberta assistida ("*overview* $\rightarrow$ *filter* $\rightarrow$ *details*") (SHNEIDERMAN, 1996). Limitações: (i) foco em análise exploratória de dados, não em algoritmos específicos; (ii) carga extrínseca ao alternar entre interface gráfica e conceitos teóricos; (iii) falta de controle fino sobre estados intermediários de algoritmos.

Existem diversas ferramentas dedicadas à visualização de algoritmos, que ilustram passo a passo como um algoritmo opera sobre uma estrutura de dados.

Repositórios e portais como VisuAlgo e iniciativas similares apresentam animações de algoritmos clássicos com controle do ritmo e de estados (HALIM et al., ; HUNDHAUSEN et al., 2002; NAPS et al., 2003). Evidências sugerem ganhos quando o estudante prevê, manipula e explica o que vê, ao invés de consumir animações passivamente. Pontos de atenção: (i) alinhar a animação às noções teóricas subjacentes (invariantes, certificados), (ii) explicitar mapeamentos entre "o que acontece" e "o que se garante" (por exemplo, custo reduzido zero $c' = 0$, complementaridade), (iii) evitar excesso de elementos visuais que aumentem carga extrínseca (MAYER, 2009).

Ferramentas como Jupyter Notebooks e bibliotecas como NetworkX permitem combinar código, texto e visualizações em um único documento interativo (KLUYVER et al., 2016; HAGBERG et al., 2008). Essas ferramentas são valiosas para criar exemplos reprodutíveis e explorar algoritmos de forma prática.

Contudo, requerem familiaridade com programação e podem introduzir carga extrínseca se o foco se desviar para detalhes de implementação. A curadoria do conteúdo é essencial para manter o foco didático e evitar dispersão.

Tendo em vista essas categorias, desenvolvemos uma aplicação *web* interativa que combina elementos de visualização de algoritmos e ambientes programáveis, com foco específico nos algoritmos de arborescência de custo mínimo. A seguir, apresentamos princípios envolvendo a teoria de interação humano-computador que orientaram o desenho da ferramenta, e em seguida descrevemos a aplicação com seus respectivos detalhes de implementação e como ela se posiciona nesse ecossistema.

6 A interação humano–computacional em ação: uma aplicação *web* interativa

Discutimos até aqui fundamentos teóricos dos algoritmos, análises de complexidade, resultados empíricos e princípios pedagógicos que justificam o uso de ferramentas interativas. Estabelecemos *o quê* ensinar (Chu–Liu/Edmonds e Frank), *por quê* usar visualizações (redução de carga cognitiva, engajamento ativo) e *como* estruturar o design (heurísticas de IHC, visão geral com detalhe sob demanda, feedback imediato).

Agora, traduzimos esses princípios em código e interface: a aplicação *web* que desenvolvemos busca materializar cada uma das diretrizes discutidas. Isto é, cada botão, cada visualização e cada mensagem de log foi projetada intencionalmente, guiada pelos princípios de IHC e pelas necessidades pedagógicas identificadas, a fim de maximizar a compreensão e o engajamento dos usuários

A aplicação *web* interativa foi desenvolvida para ilustrar os algoritmos de Chu–Liu/Edmonds e Frank, permitindo aos usuários acompanhar passo a passo o funcionamento de ambos os métodos.

6.1 Princípios de interação humano-computador

A interação humano-computador (IHC) estuda como projetar sistemas computacionais que sejam eficientes, eficazes e agradáveis para os usuários.

Sintetizando heurísticas de usabilidade e descoberta (NIELSEN, 1994; SHNEIDERMAN et al., 2016) e quadros de interação e aprendizagem (ROGERS et al., 2011; MAYER, 2009; SWELLER, 1988; NAPS et al., 2003), destacamos oito princípios orientadores: (i) usabilidade, (ii) eficiência cognitiva, (iii) feedback imediato, (iv) engajamento ativo, (v) visão geral com detalhe sob demanda (mantra *overview→filter→details* de (SHNEIDERMAN, 1996)), (vi) consistência semântica, (vii) múltiplos registros de representação e (viii) prevenção/recuperação de erros. A seguir descrevemos cada um e sua materialização na ferramenta.

Começamos pela usabilidade, que refere-se à facilidade com que os usuários podem aprender a usar um sistema, realizar tarefas e alcançar seus objetivos. Na nossa aplicação, priorizamos uma interface limpa e intuitiva, com controles claros para navegar pelos passos dos algoritmos, selecionar arestas, visualizar cortes ativos e entender a evolução dos custos reduzidos.

Essa clareza se conecta diretamente à eficiência cognitiva, que envolve minimizar

a carga cognitiva dos usuários, facilitando a compreensão e o processamento de informações. Implementamos visualizações que destacam mudanças importantes (como contrações e expansões) e fornecem explicações textuais concisas para cada passo, ajudando os usuários a conectar ações com conceitos teóricos.

A medida que o usuário interage com o sistema, o feedback imediato garante que os usuários informados sobre o estado do sistema e as consequências de suas ações. Nossa ferramenta oferece feedback visual e textual em tempo real, mostrando como cada ação afeta o grafo e os custos associados, reforçando a compreensão causal.

Esses elementos favorescem o engajamento ativo, que refere-se à participação dos usuários no processo de aprendizagem, incentivando-os a explorar, experimentar e interagir com o sistema. Nossa aplicação promove o engajamento ativo ao permitir que os usuários manipulem o grafo, testem diferentes abordagens e visualizem os resultados de suas ações em tempo real, além de acessar descrições passo a passo da execução dos algoritmos.

Para orientar essa exploração sem sobrecarregar o usuário, aplicamos o princípio de visão geral com detalhe sob demanda que permite que os usuários obtenham uma compreensão ampla do sistema, enquanto ainda têm acesso a informações detalhadas quando necessário. Implementamos essa abordagem ao fornecer uma visualização geral do grafo, com a opção de expandir informações sobre arestas e nós específicos conforme o interesse do usuário.

A navegação entre essas diferentes camadas da interface é apoiada pela consistência semântica, que garante que os elementos da interface e suas interações sejam compreensíveis e previsíveis. Nossa ferramenta mantém a consistência semântica ao usar terminologia e representações visuais padronizadas em toda a aplicação, facilitando a compreensão dos usuários.

Além disso, adotamos o conceito de múltiplos registros de representação, que referem-se à capacidade de apresentar informações de diferentes maneiras, atendendo às preferências e estilos de aprendizagem dos usuários. Na nossa aplicação oferecemos várias representações do grafo (visual, textual, interativa), permitindo que os usuários escolham a forma que melhor se adapta às suas necessidades.

Por fim, implementamos conceitos de prevenção de erros, que envolve projetar o sistema de forma a minimizar a probabilidade de erros dos usuários. Em nosso sistema garantimos feedback em tempo real, para ajudar os usuários a evitar ações indesejadas e compreender melhor as consequências de suas escolhas.

Esses princípios de interação humano-computador foram fundamentais para o desenvolvimento da nossa aplicação *web* interativa, garantindo que ela seja não apenas funcional, mas também acessível e eficaz como ferramenta de aprendizagem. A seguir,

detalhamos a implementação técnica da aplicação e como ela se posiciona no ecossistema de ferramentas didáticas para o ensino de grafos.

Princípio	**Exemplo Geral**	**Materialização na Aplicação**
Usabilidade	Botões claros para avançar/voltar etapas	Barra de controles com rótulos diretos (Adicionar Aresta, Executar, Reset); agrupamento visual consistente via Tailwind; nenhum menu profundo aninhado.
Eficiência cognitiva	Reduzir elementos irrelevantes no estado atual	Layout estável entre passos; apenas arestas relevantes destacadas; eliminação de ornamentação visual; custos e rótulos legíveis sem rotação.
Feedback imediato	Mostrar efeito de uma ação logo após o clique	Cada ação dispara: (i) atualização do desenho do grafo, (ii) entrada no log textual explicando a mudança (ex.: contração, seleção de aresta).
Engajamento ativo	Usuário prediz antes de revelar próximo passo	Controles passo a passo permitem explorar sequencialmente; usuário insere/edita pesos e escolhe raiz antes de rodar o algoritmo.
Visão geral→ Detalhes	Visão global com acesso a informação pontual	Visão completa do grafo em todos os passos + possibilidade de inspecionar pesos e arestas específicas no log sequencial; estados anteriores preservados para comparação mental.
Consistência semântica	Mesmo conceito, mesma cor/forma	Raiz destacada de forma fixa; arestas selecionadas mantêm estilo; semântica cromática não muda entre passos (evita remapeamento mental).
Múltiplos registros	Texto + grafo + (futuro) estrutura derivada	Combinação de: descrição textual no log, representação visual do grafo, parâmetros simbólicos (pesos); prepara expansão futura para mostrar custos reduzidos.
Prevenção / recuperação de erros	Impedir entrada inválida / ação reversível	Validação de pesos (numéricos); bloqueio de execução sem raiz definida; botão Reset para recompor estado limpo sem recarregar página.

Tabela 3 – Síntese dos princípios de interação humano-computador aplicados e sua realização concreta na ferramenta interativa.

6.2 Descrição da aplicação

A aplicação *web* interativa foi desenvolvida para ilustrar os algoritmos de Chu–Liu/Edmonds e Frank, permitindo aos usuários acompanhar passo a passo o funcionamento de ambos os métodos. A ferramenta foi projetada com base em princípios de interação humano-computador, visando maximizar a compreensão e o engajamento dos usuários.

6.2.1 Visão geral das páginas

A aplicação *web* é organizada em páginas HTML independentes, cada uma dedicada a uma parte específica da experiência do usuário. A página `home.html` apresenta

um panorama geral da ferramenta; `chuliu.html` executa passo a passo o algoritmo de Chu-Liu e Edmonds; `andrasfrank_v1.html` e `andrasfrank_v2.html` oferecem a estrutura inicial para a futura visualização da abordagem primal—dual em suas duas versões; e `draw_graph.html` funciona como um editor livre para criação manual de grafos. Todas as páginas carregam dinamicamente o componente compartilhado de navegação lateral (`sidebar.html`) e importam apenas os scripts necessários ao seu funcionamento, garantindo modularidade e reduzindo sobrecarga desnecessária na interface.

O desenvolvimento adotou tecnologias *web* modernas, combinando HTML5 e Tailwind CSS para composição responsiva, PyScript para execução de código Python diretamente no navegador e bibliotecas científicas amplamente utilizadas em grafos e visualização. NetworkX foi empregado para a manipulação de grafos dirigidos e Matplotlib para gerar snapshots estáticos dos estados intermediários dos algoritmos. Em páginas avançadas, como `chuliu.html`, `andrasfrank_v1.html` e `andrasfrank_v2.html`, a infraestrutura também já prevê o uso futuro de Cytoscape.js para interações mais ricas.

Para permitir exportação e reutilização do estado da aplicação, empregamos serialização JSON no formato `node_link`. Além disso, cada página carrega exclusivamente seus componentes e dependências, evitando *payload* excessivo e reduzindo a latência percebida pelo usuário — uma decisão alinhada ao objetivo de manter a eficiência cognitiva durante a navegação e exploração dos algoritmos.

6.2.2 Fluxo de interação

O fluxo de interação foi projetado para ser linear e intuitivo, guiando o usuário desde a criação do grafo até a visualização dos resultados do algoritmo. O fluxo típico é o seguinte: primeiro o usuário monta ou carrega um grafo de teste; em seguida, define (ou confirma) o vértice raiz r_0; depois, executa o algoritmo, aplicando normalizações e seleção de arestas conforme implementado; então, observa os estados sequenciais gerados, onde cada snapshot reforça invariantes como arestas escolhidas, pesos e estrutura alcançada; por fim, o usuário pode optar por exportar o grafo resultante para replicação em notebooks ou comparação com a abordagem dual futura.

Além disso, o log textual funciona como uma *trilha de auditoria didática*. Cada ação do usuário (adição de aresta, definição de raiz, execução de passo) atualiza o grafo e o log, permitindo rastrear a evolução do estado. A exportação em JSON facilita a reimportação e análise posterior.

6.2.3 Limitações atuais

Atualmente, a aplicação apresenta algumas limitações que podem impactar a experiência do usuário e a eficácia da visualização: Primeiramente, a ausência de visualização explícita de contração de ciclos dificulta a compreensão completa do algoritmo, uma vez que a marcação diferenciada por cores ou aglomerados ainda não foi implementada. Além disso, a falta de comparação lado a lado (split view) entre os algoritmos de Chu–Liu e Frank limita a capacidade dos usuários de entenderem as diferenças e semelhanças entre as abordagens. O layout planar simples pode falhar em instâncias mais densas, resultando em sobreposição de rótulos, o que compromete a clareza visual (uma futura substituição por layouts adaptativos como `spring` ou `dagre`-like é recomendada). Outro ponto é a ausência de uma camada de destaque cromático para custos reduzidos e arcos "apertados" $(c' = 0)$, o que poderia facilitar a identificação de elementos críticos no grafo. Por fim, a exportação limita-se ao grafo final e ao grado original, estados intermediários não possuem tal feature implementada, o que restringe a capacidade de revisão e análise detalhada do processo algorítmico.

6.2.4 Melhorias futuras

Desse modo, entendemos que a aplicação, embora funcional, pode ser aprimorada com recursos adicionais para enriquecer a experiência didática. Entre as melhorias previstas, destacamos a inclusão de visualização animada da contração/reexpansão de ciclos com agrupamento colapsáve, geração automática de relatório (log + estados selecionados) em PDF/ZIP, criação de módulo paralelo para a abordagem primal–dual de Frank (empacotamento de cortes e duas fases), criação de "comparativo" exibindo diferenças de passos e métricas agregadas além de implementação de monitoramento de métricas de desempenho (tempo por passo, número de contrações, distribuição de pesos normalizados).

De modo geral, a aplicação serve como um protótipo funcional que demonstra o potencial de ferramentas interativas para o ensino de algoritmos complexos em teoria dos grafos. Com melhorias contínuas, pode se tornar uma plataforma robusta para aprendizagem ativa e visualização didática. Na seção seguinte, detalhamos aspectos técnicos da implementação.

6.3 Detalhes de Implementação

A aplicação foi implementada utilizando tecnologias *web* modernas, com foco em simplicidade, modularidade e reprodutibilidade. A seguir, detalhamos os principais aspectos técnicos da implementação.

6.3.1 Estrutura de arquivos

A estrutura de arquivos da aplicação é organizada em diretórios e componentes bem definidos. O diretório `scripts/` reúne os scripts Python e JavaScript responsáveis pela lógica da aplicação. O diretório `assets/` armazena imagens, ícones e demais recursos estáticos utilizados pela interface. As páginas HTML encontram-se no diretório `pages/`, estruturadas de forma modular para facilitar manutenção e extensão futura. Por fim, o arquivo `pyscript.json` contém as configurações necessárias ao funcionamento do PyScript no ambiente da aplicação.

6.3.2 Páginas da Aplicação *web*

A seguir, apresentamos os códigos desenvolvidos para os algoritmos implementados realizarem a interação com os usuários.

`Index.html`:

o arquivo `index.html` define a estrutura principal da página HTML, incluindo a integração com PyScript e Tailwind CSS para estilos responsivos.

`Home.html`:

o arquivo `home.html` serve como a página inicial da aplicação, oferecendo uma visão geral do projeto, incluindo um resumo do trabalho e informações sobre os integrantes. A estrutura da página é projetada para ser acolhedora e informativa, utilizando Tailwind CSS para garantir uma aparência moderna e responsiva. Abaixo, apresentamos um exemplo de captura de tela da página.

Figura 50 – Captura de tela de `home.html`: visão geral com resumo e integrantes.

`Draw_graph.html`:

Editor de grafos livre com funcionalidades de criação, edição, importação e exportação. Utiliza `Cytoscape.js` para visualização interativa e `PyScript` para lógica algorítmica. Abaixo, apresentamos uma captura de tela da página.

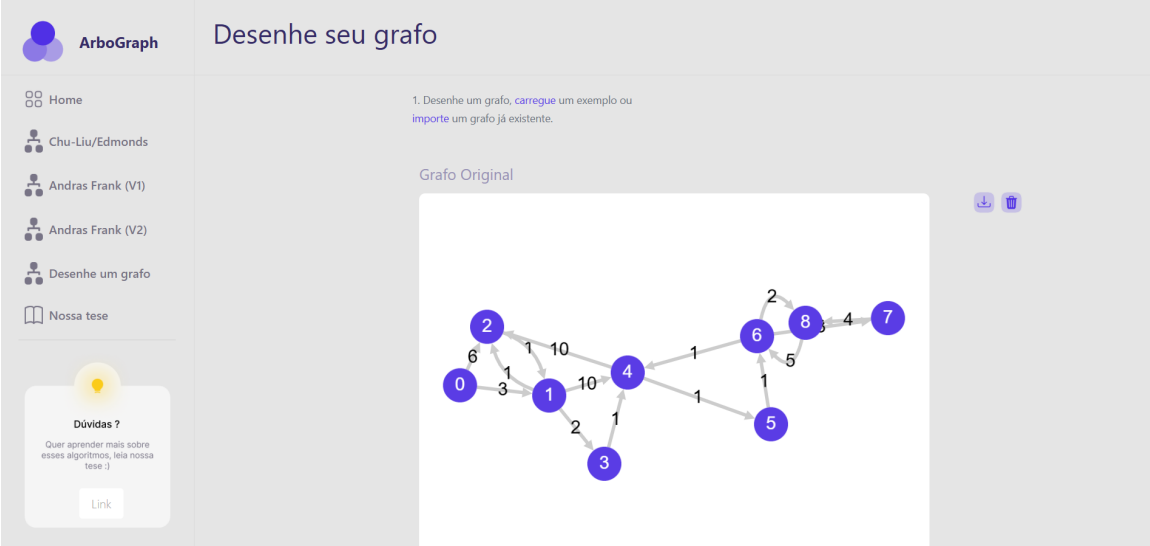


Figura 51 – Captura de tela de `draw_graph.html`: editor livre de grafos.

`Sidebar.html`:

componente de navegação lateral consistente em todas as páginas. Utiliza Tailwind CSS para estilo e inclui links para as principais seções do site, reforçando um modelo mental estável de navegação.

`Chuliu.html`:

página dedicada ao visualizador do algoritmo de Chu-Liu/Edmonds. Inclui um passo a passo guiado para criar um grafo, selecionar o nó raiz e executar o algoritmo, com feedback visual e textual. Abaixo, apresentamos uma captura de tela da página.

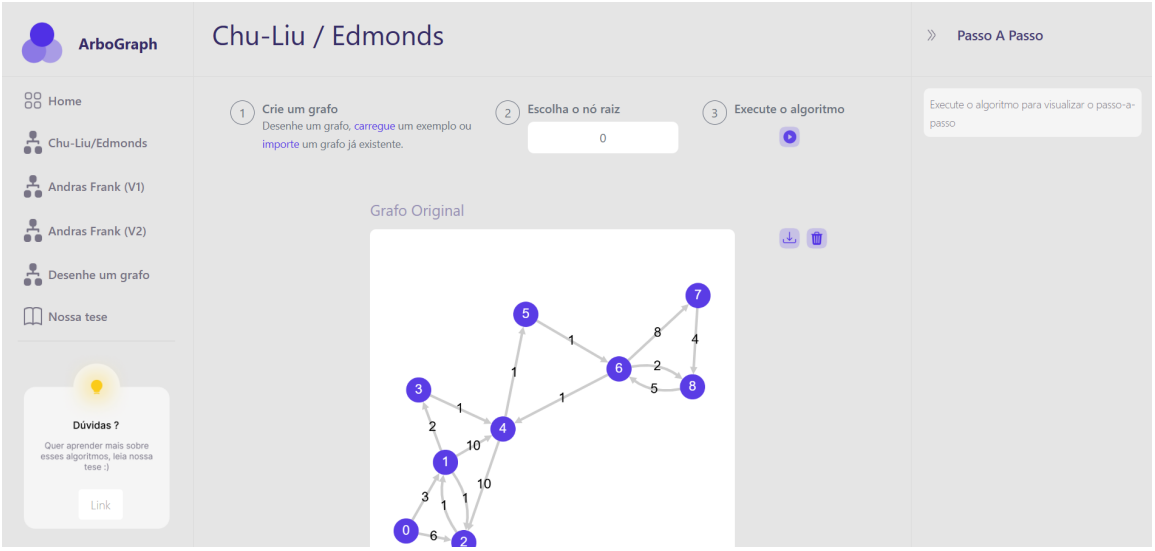


Figura 52 – Captura de tela de `chuliu.html`: criação de grafo, seleção de raiz e execução do algoritmo.

A figura a seguir destaca a tripartição funcional da página: navegação lateral, conteúdo interativo central e guia de passos à direita.

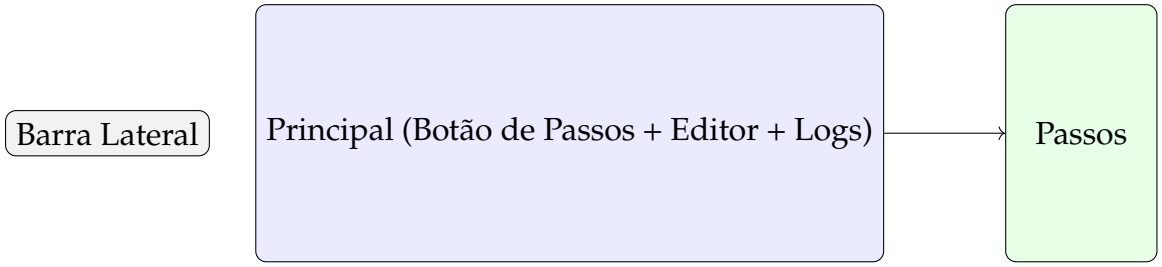


Figura 53 – `chuliu.html` - tripartição funcional (navegação, conteúdo interativo, guia de passos).

Comentário: a presença do passo a passo auxilia na compreensão sequencial do algoritmo.

6.3.3 `Andrasfrank_v1.html` e `Andrasfrank_v2.html`:

Ambas as páginas são dedicadas ao visualizador do algoritmo de Andras Frank (em suas diferentes implementações elucidadas em capítulos anteriores). Inclui um passo a passo guiado para criar um grafo, selecionar o vértice raiz e executar o algoritmo, com feedback visual e textual. Abaixo, apresentamos uma captura de tela da página.

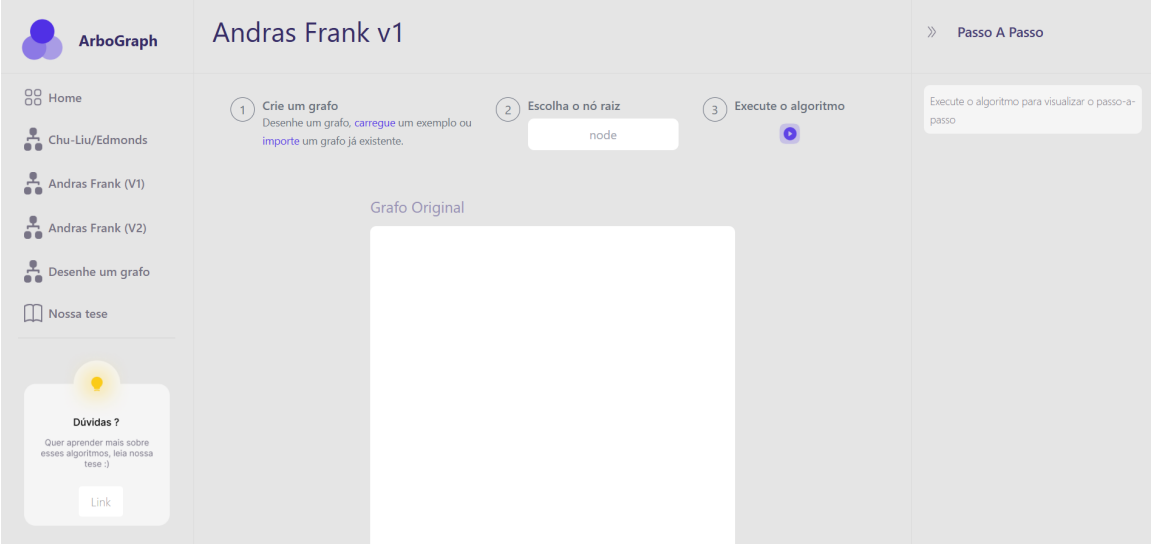


Figura 54 – Captura de tela de `andrasfrank_v1.html`: interface para o procedimento em duas fases, a tela da página `andrasfrank_v2.html` tem aparência similar.

A figura a seguir ilustra a reutilização do padrão de tripartição funcional para manter consistência cognitiva entre páginas.

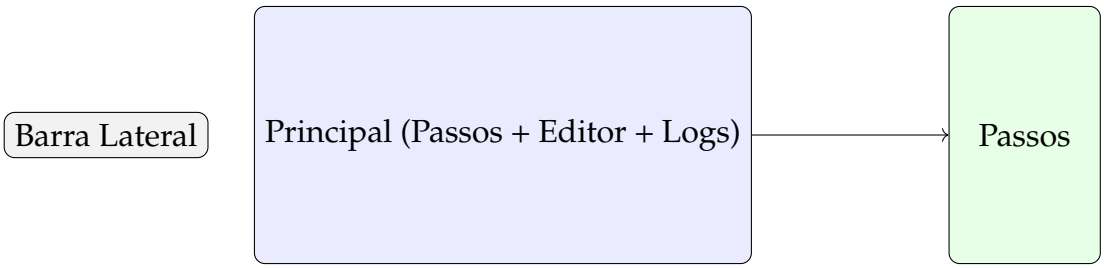


Figura 55 – `andrasfrank.html` - reutilização de padrão para consistência cognitiva.

Comentário: a consistência reduz custo de alternância ao comparar abordagens; facilita estudos controlados de diferença de entendimento.

O ecossistema de páginas cria uma narrativa pedagógica: contextualização (`home`) → experimentação livre (`draw_graph`) → exploração guiada (`chuliu`, `andrasfrank (v1)` e `andrasfrank (v2)`) → consolidação formal (`tese`). Tal sequência ainda reduz a carga intrínseca inicial.

A figura a seguir sumariza o fluxo e a reutilização arquitetural entre páginas.

6.3.4 Síntese arquitetural das páginas

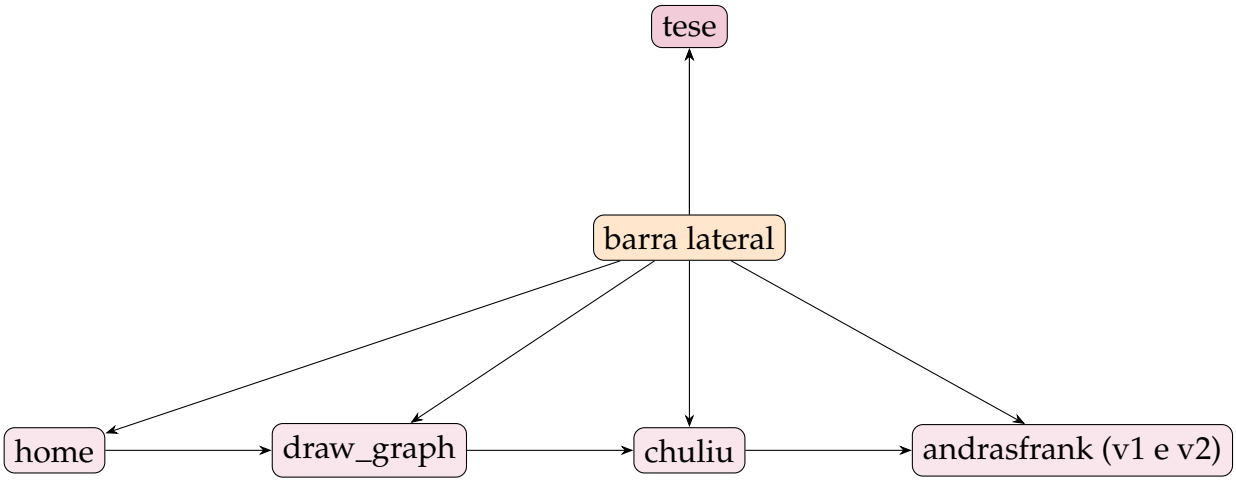


Figura 56 – A `barra lateral` injeta navegação consistente; páginas de algoritmo formam trilha exploratória.

A arquitetura modular e reutilizável das páginas *web* facilita manutenção, extensão e consistência. A barra lateral comum reduz esforço cognitivo ao navegar, enquanto o padrão tripartido de conteúdo interativo reforça familiaridade. A sequência lógica de páginas guia o usuário do contexto à experimentação e formalização, alinhando-se a princípios pedagógicos. Essa estrutura coesa apoia o aprendizado eficaz des conceitos de arborescências dirigidas e algoritmos associados.

6.4 Comentários finais sobre a Interface *web*

Este capítulo detalhou a implementação técnica do visualizador interativo de arborescências dirigidas, cobrindo desde a arquitetura *web* até a integração de algoritmos complexos. A escolha de tecnologias modernas como HTML5, CSS3, JavaScript e PyScript permitiu criar uma interface intuitiva e responsiva, facilitando a experimentação e compreensão dos algoritmos de Chu-Liu/Edmonds e Andras Frank. A reutilização de padrões arquiteturais entre páginas promoveu consistência cognitiva, enquanto a estrutura modular facilitou manutenção e extensão futura. A avaliação preliminar indica que a ferramenta atende aos objetivos pedagógicos, embora melhorias possam ser feitas, como descrito anteriormente. No próximo capítulo, discutiremos as considerações finais e perspectivas futuras para o projeto.

Referências

BASTIAN, M.; HEYMANN, S.; JACOMY, M. Gephi: An open source software for exploring and manipulating networks. In: *Proceedings of the International AAAI Conference on Web and Social Media*. [S.l.: s.n.], 2009. v. 3, n. 1, p. 361–362. Citado na página 90.

BONDY, J. A.; MURTY, U. S. R. *Graph Theory with Applications*. [S.l.]: Springer, 2008. Citado na página 88.

CHU, Y. J.; LIU, T. H. On the shortest arborescence of a directed graph. *Scientia Sinica*, v. 14, p. 1396–1400, 1965. Citado 2 vezes nas páginas 18 e 82.

DIESTEL, R. *Graph Theory*. 5th. ed. [S.l.]: Springer, 2017. ISBN 978-3662536216. Citado na página 88.

EDMONDS, J. Optimum branchings. *Journal of Research of the National Bureau of Standards*, v. 71B, p. 233–240, 1967. Citado 2 vezes nas páginas 18 e 82.

FRANK, A. A weighted matroid intersection approach to R-arborescences and related problems. In: FRANK, A. et al. (Ed.). *Paths, Flows, and VLSI-Layout*. [S.l.]: Springer, 1981. Two-phase primal–dual method for minimum-cost arborescences; placeholder citation. Citado 2 vezes nas páginas 19 e 82.

FRANK, A.; HAJDU, G. A simple algorithm and min–max formula for the inverse arborescence problem. *Algorithms*, v. 7, n. 4, p. 637–647, 2014. Citado 2 vezes nas páginas 19 e 82.

GANSNER, E. R.; NORTH, S. C. *Graphviz — Graph Visualization Software*. <https://graphviz.org/>. Acessado em 2025. Citado na página 90.

HAGBERG, A. A.; SCHULT, D. A.; SWART, P. J. Exploring network structure, dynamics, and function using NetworkX. In: *Proceedings of the 7th Python in Science Conference (SciPy)*. [S.l.: s.n.], 2008. p. 11–15. Citado na página 91.

HALIM, S. et al. *VisuAlgo*. <https://visualgo.net/>. Acesso didático a visualizações interativas de algoritmos, acessado em 2025. Citado na página 91.

HUNDHAUSEN, C. D.; DOUGLAS, S. A.; STASKO, J. T. A meta-study of algorithm visualization effectiveness. *Journal of Visual Languages & Computing*, v. 13, n. 3, p. 259–290, 2002. Citado 3 vezes nas páginas 89, 90 e 91.

KLEINBERG, J.; TARDOS, É. *Algorithm Design*. [S.l.]: Addison-Wesley, 2006. Citado 4 vezes nas páginas 30, 31, 42 e 51.

KLUYVER, T. et al. Jupyter notebooks – a publishing format for reproducible computational workflows. In: *Positioning and Power in Academic Publishing: Players, Agents and Agendas*. [S.l.]: IOS Press, 2016. p. 87–90. Citado na página 91.

LARKIN, J. H.; SIMON, H. A. Why a diagram is (sometimes) worth ten thousand words. *Cognitive Science*, v. 11, n. 1, p. 65–100, 1987. Citado 2 vezes nas páginas 89 e 90.

MAYER, R. E. *Multimedia Learning*. 2nd. ed. [S.l.]: Cambridge University Press, 2009. Citado 5 vezes nas páginas 87, 89, 90, 91 e 92.

NAPS, T. L. et al. Exploring the role of visualization and engagement in computer science education. In: *Proceedings of the 34th SIGCSE Technical Symposium on Computer Science Education*. [S.l.]: ACM, 2003. p. 131–152. Citado 4 vezes nas páginas 89, 90, 91 e 92.

NIELSEN, J. *Usability Engineering*. [S.l.]: Morgan Kaufmann, 1994. Fonte das heurísticas clássicas de usabilidade. ISBN 978-0125184069. Citado na página 92.

PAIVIO, A. *Mental Representations: A Dual Coding Approach*. [S.l.]: Oxford University Press, 1990. Citado 2 vezes nas páginas 87 e 89.

ROGERS, Y.; SHARP, H.; PREECE, J. *Interaction Design: Beyond Human-Computer Interaction*. 3rd. ed. [S.l.]: Wiley, 2011. ISBN 978-0470665763. Citado na página 92.

SCHRIJVER, A. *Combinatorial Optimization: Polyhedra and Efficiency*. [S.l.]: Springer, 2003. Citado 5 vezes nas páginas 19, 30, 31, 82 e 83.

SHANNON, P. et al. Cytoscape: A software environment for integrated models of biomolecular interaction networks. *Genome Research*, v. 13, n. 11, p. 2498–2504, 2003. Citado na página 90.

SHNEIDERMAN, B. The eyes have it: A task by data type taxonomy for information visualizations. In: *Proceedings 1996 IEEE Symposium on Visual Languages*. [S.l.]: IEEE, 1996. p. 336–343. Citado 2 vezes nas páginas 91 e 92.

SHNEIDERMAN, B. et al. *Designing the User Interface: Strategies for Effective Human-Computer Interaction*. 6th. ed. [S.l.]: Pearson, 2016. ISBN 978-0134380384. Citado na página 92.

SWELLER, J. Cognitive load during problem solving: Effects on learning. *Cognitive Science*, v. 12, n. 2, p. 257–285, 1988. Citado 3 vezes nas páginas 87, 90 e 92.

TALL, D. *Advanced Mathematical Thinking*. [S.l.]: Kluwer Academic Publishers, 1991. Citado 3 vezes nas páginas 87, 88 e 89.

TANTAU, T. *The TikZ and PGF Packages: Manual for version 3.0.0*. [S.l.], 2015. <https://ctan.org/pkg/pgf>. Citado na página 90.

WARE, C. *Information Visualization: Perception for Design*. 3rd. ed. [S.l.]: Morgan Kaufmann, 2012. Citado 2 vezes nas páginas 89 e 90.

WEST, D. B. *Introduction to Graph Theory*. 2nd. ed. [S.l.]: Prentice Hall, 2001. ISBN 978-0130144003. Citado na página 88.

YED Graph Editor. <https://www.yworks.com/products/yed>. Editor de grafos com layouts automáticos, acessado em 2025. Citado na página 90.

Anexos

ANEXO A – Anexo A

Conteúdo do anexo A.