Lorena Silva Sampaio, Samira Haddad

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

Brasil

2025

Lorena Silva Sampaio, Samira Haddad

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

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

Universidade Federal do ABC

Orientador: Prof. Dr. Mário Leston

Brasil
2025

Dedicatória (opcional).

Agradecimentos

Agradecimentos (opcional).

“Epígrafe (opcional).”

Resumo

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

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

Abstract

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

Keywords: Graphs. Arborescence. Algorithms. Visualization.

Lista de ilustrações

Sumário

1 Algoritmo de Chu–Liu–Edmonds

Neste capítulo, apresentaremos o algoritmo de Chu–Liu–Edmonds, que determina uma arborescência de custo mínimo em um 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.

1.1 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,

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

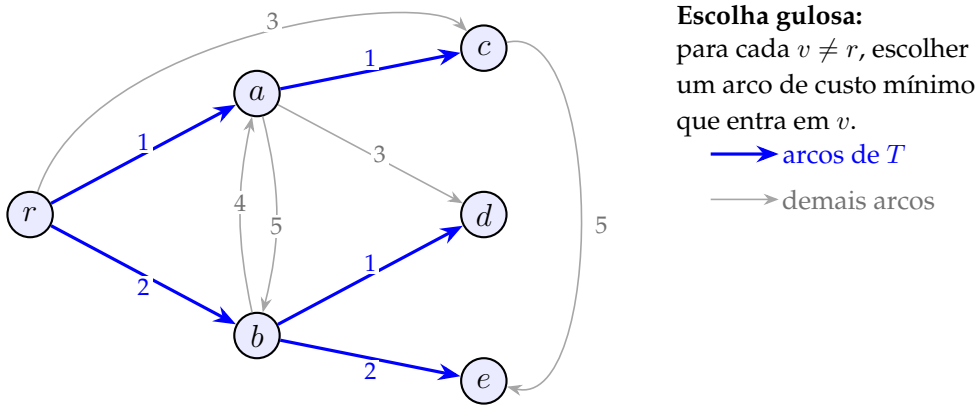


Figura 1 – A figura ilustra a escolha gulosa quando esta produz uma r-arborescência. Os arcos em azul são os escolhidos; os cinza são os demais arcos do digrafo.

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

A seguinte figura ilustra que podemos não ter tanta sorte.

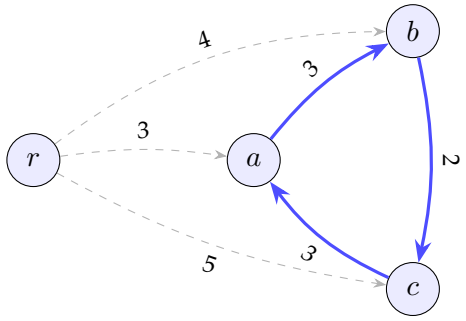


Figura 2 – Os arcos azuis são os da escolha gulosa.

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

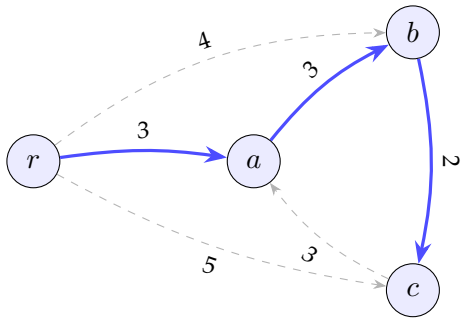


Figura 3 – Os arcos azuis são os da escolha gulosa.

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

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

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

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

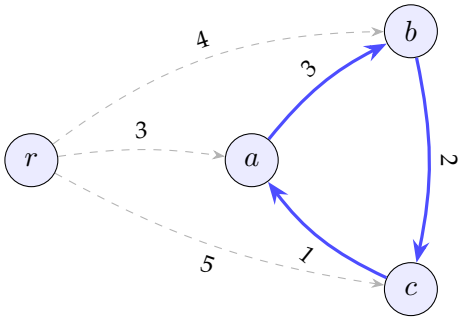


Figura 4 – Os arcos azuis são os da escolha gulosa.

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

Seja $q : V \setminus \{r\} \to \mathbb{R}$ (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 1.1. *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 4.

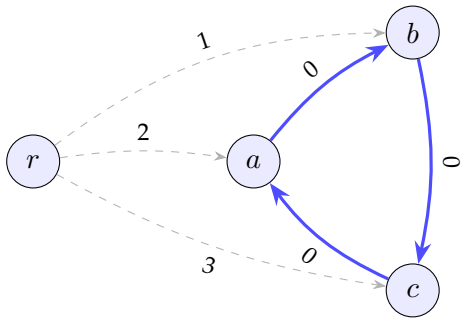


Figura 5 – Os arcos azuis são os da escolha gulosa.

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.

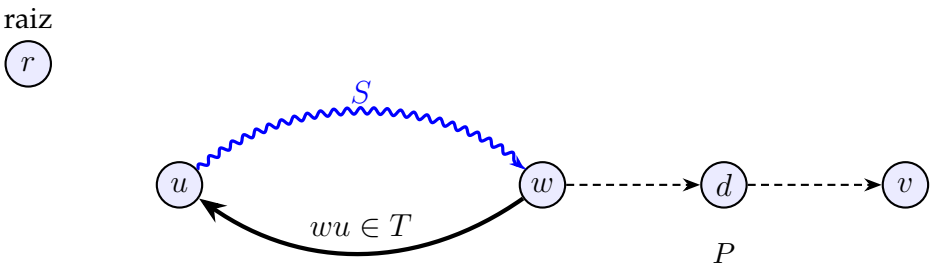


Figura 6 – O caminho simples maximal P inicia em u e termina em v. A porção S de P entre u e w é indicada pelo arco ondulado azul; o caminho $S \cdot u$ é um ciclo.

ajuste (custos reduzidos), cada $v \neq r$ passa a ter ao menos um arco de custo reduzido zero. Se os arcos de custo zero forem acíclicos, já temos a r-arborescência ótima. Se formarem um ciclo C, *contraímos* C em um **supervértice** x_C, ajustamos os custos dos arcos externos e resolvemos recursivamente no 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 $C = (v_1, v_2, v_3, v_1)$.

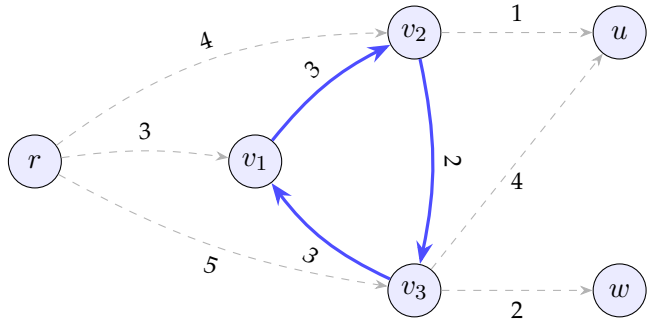


Figura 7 – O digrafo D com o ciclo $C = (v_1, v_2, v_3, v_1)$. Os arcos azuis representam os arcos do ciclo, os tracejados representam os demais arcos do digrafo.

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:

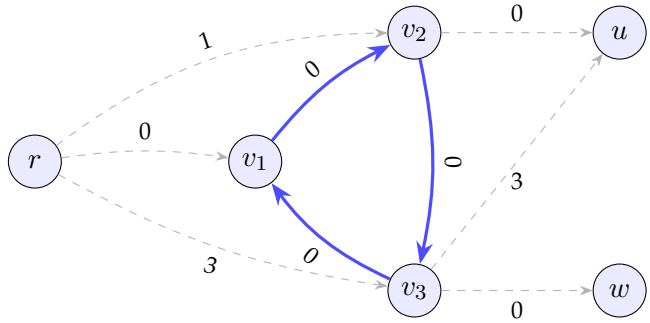


Figura 8 – digrafo com custos λ-reduzidos. Os arcos internos do ciclo C têm custo zero (em azul). Os arcos da raiz para o ciclo têm custos 1, 0 e 3 (tracejados).

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

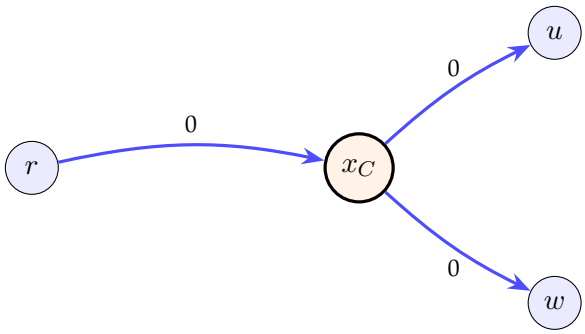


Figura 9 – digrafo D' após a contração do ciclo C. O supervértice x_C substitui todos os vértices do ciclo. Originalmente, havia três arcos paralelos de r para o ciclo: (r, v_1), (r, v_2) e (r, v_3) com custos reduzidos 1, 0 e 3; mantemos apenas o de menor custo 0. Os arcos que saíam do ciclo agora saem 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:

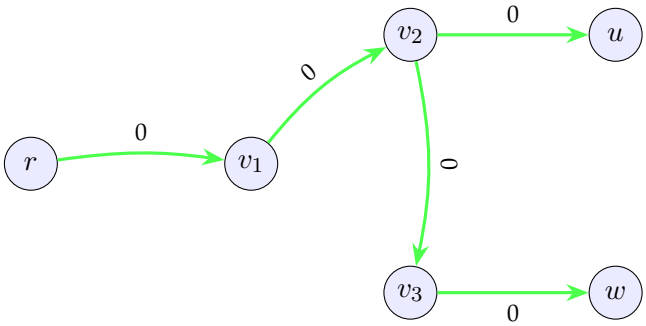


Figura 10 – Reexpansão da r-arborescência ótima T' em D' para obter a r-arborescência T em D

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

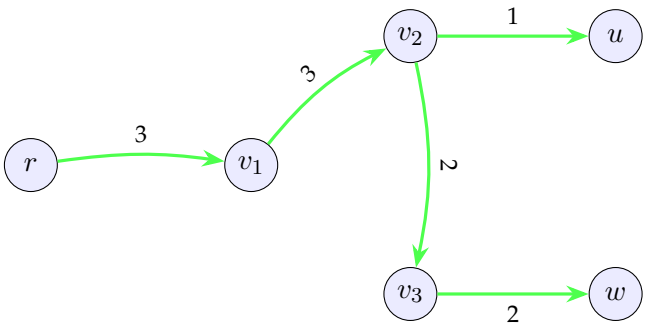


Figura 11 – Reexpansão da r-arborescência ótima T' em D' para obter a r-arborescência T em D. Os arcos selecionados em verde fazem parte de T.

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:

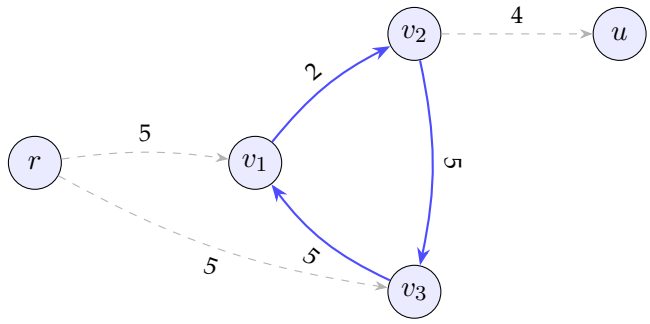


Figura 12 – digrafo D com custos originais. O ciclo $C = (v_1, v_2, v_3, v_1)$ tem arcos com custos 5, 5 e 2. Existem dois arcos da raiz para o ciclo, ambos com custo 5: (r, v_1) e (r, v_3).

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

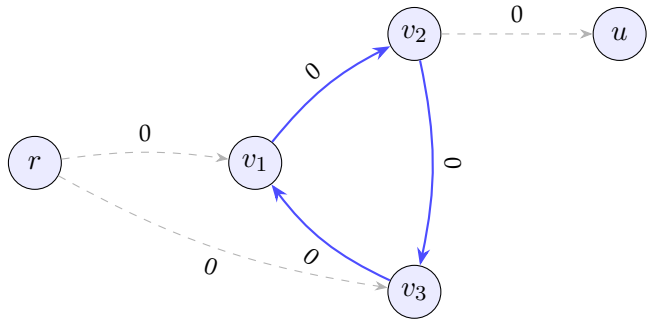


Figura 13 – digrafo D com custos λ-reduzidos. Todos os arcos do ciclo custo mínimo têm agora custo zero.

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'

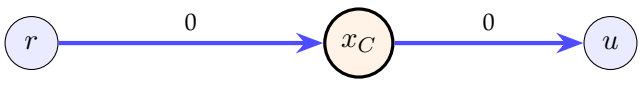


Figura 14 – Arborescência T' no digrafo contraído D'. O arco (r, x_C) pode corresponder ou ao arco (r, v_1) ou ao (r, v_3) em D e o arco (x_C, u) corresponde ao arco normalizado (v_2, u) em D.

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

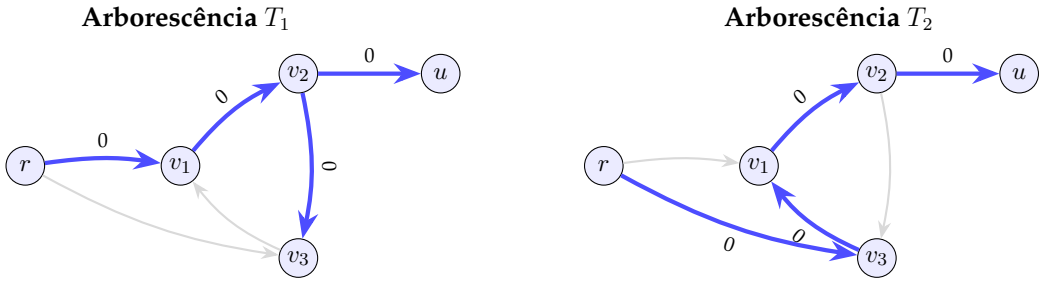


Figura 15 – Duas r-arborescências ótimas distintas em D com custos c_λ-reduzidos. T_1 usa o arco (r, v_1) e os arcos do ciclo (v_1, v_2) e (v_2, v_3). T_2 usa o arco (r, v_3) e os arcos do ciclo (v_3, v_1) e (v_1, v_2). Ambas incluem o arco (v_2, u) e têm custo total zero.

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 1.2. *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).

1.2 Descrição do algoritmo

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

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

Entrada: digrafo $D = (V, A)$, custos $c : A \to \mathbb{R}_{\geq 0}$, raiz r.[a]

1. Para cada $v \neq r$, escolha $a_v \in \operatorname{argmin}_{(u,v) \in A} c(u, v)$. Defina $y(v) := c(a_v)$ e $A_0 := \{a_v : v \neq r\}$.

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.

1.2.1 Corretude

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

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

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

pois há exatamente um arco de T entrando em cada $v \neq r$. O termo $\sum_{v \neq r} y(v)$ é constante (independe de T); assim, minimizar $\sum c$ equivale a minimizar $\sum c'$ (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.

1.2.2 Complexidade

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

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

1.3 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.

1.3.1 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.

1.3.2 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
```

1.3.3 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 16 ilustra o funcionamento da normalização:

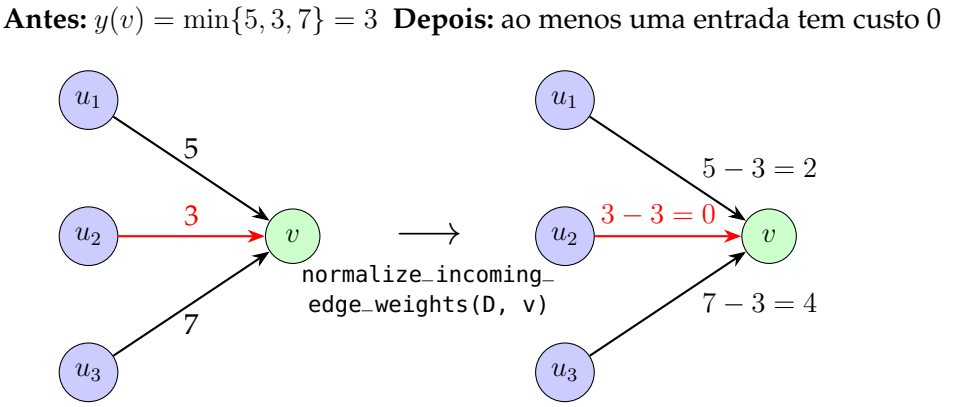


Figura 16 – Exemplo de normalização de custos reduzidos. À esquerda, vértice v com três arcos de entrada (pesos 5, 3 e 7). À direita, após aplicar `reduce_weights(D, v)`: o menor peso $y(v) = 3$ é subtraído de todas as entradas, resultando em custos reduzidos 2, 0 e 4. O arco (u_2, v) (em vermelho) tem 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.

1.3.4 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.

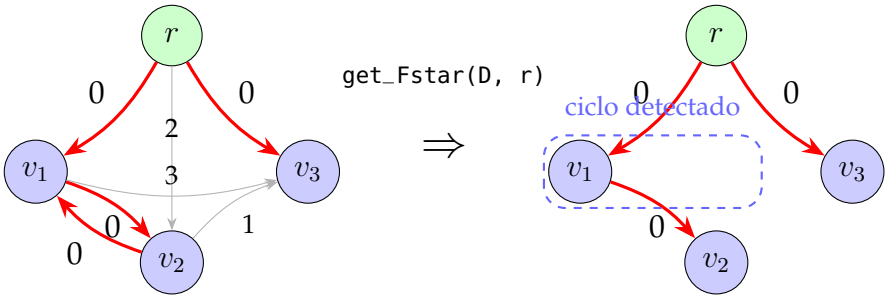
Construção de A zero

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()
```

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

A Figura 17 ilustra a construção de A_0:



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

Figura 17 – 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 $A_0 := \{a_v : v \neq r\}$.

1.3.5 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 18 ilustra o processo de detecção de ciclo:

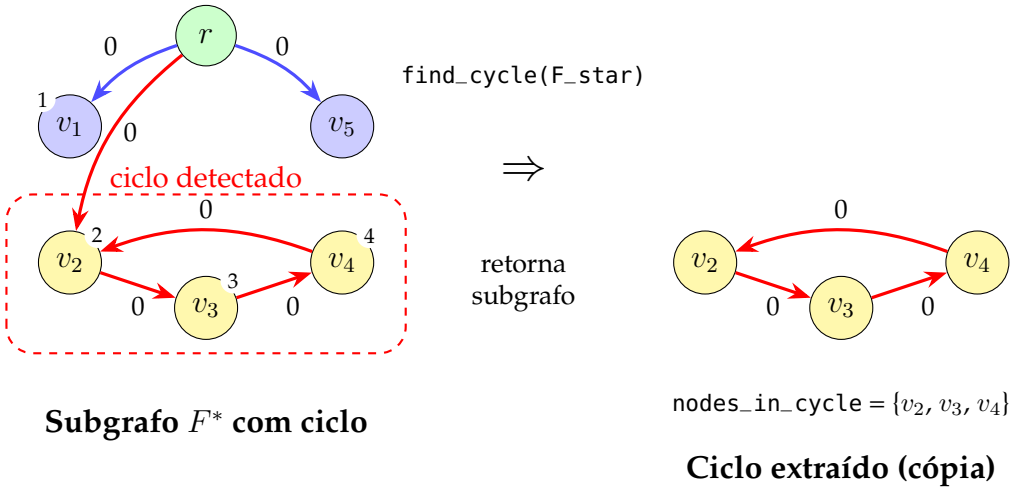


Figura 18 – Exemplo de detecção de ciclo em A_0. À esquerda, o subdigrafo A_0 contém um ciclo formado pelos vértices $\{v_2, v_3, v_4\}$ (destacados em amarelo). A DFS percorre o digrafo e detecta o ciclo ao encontrar o arco (v_4, v_2), onde v_2 já está na pilha de recursão. À direita, a função devolve 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.

1.3.6 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 19 ilustra o processo de contração de ciclo:

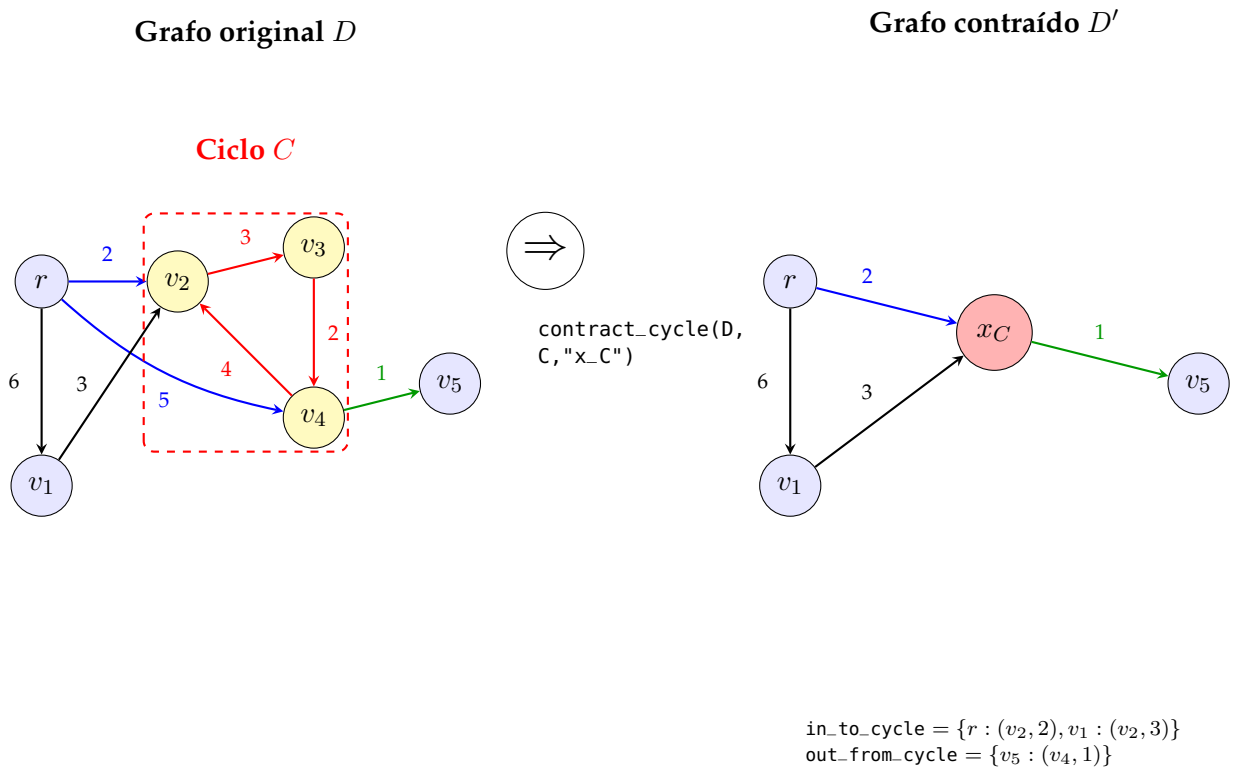


Figura 19 – Exemplo de contração de ciclo. À esquerda, digrafo original D com ciclo $C = \{v_2, v_3, v_4\}$ (em amarelo). Vértices externos r, v_1 e v_5 têm arcos conectando ao ciclo: r envia arco para v_2 (peso 2) e v_4 (peso 5); v_4 envia arco para v_5 (peso 1). À direita, após a contração: o ciclo 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 $D' = (V', A')$.

1.3.7 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.

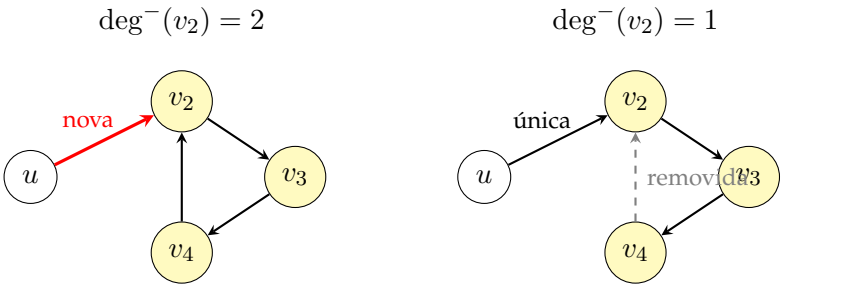
A função recebe como entrada o ciclo `C` e o vértice de entrada `v`. A implementação utiliza uma compreensão de gerador combinada com `next` para encontrar o predecessor de `v` dentro do ciclo (linha 2): a expressão `(u for u, _ in C.in_edges(v))` itera sobre os arcos de entrada de `v`, extraindo apenas 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 20 ilustra o objetivo da função:



Ciclo C após adicionar aresta externa (u, v_2)

Após remover aresta interna (v_4, v_2)

Figura 20 – 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.

1.3.8 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 21.

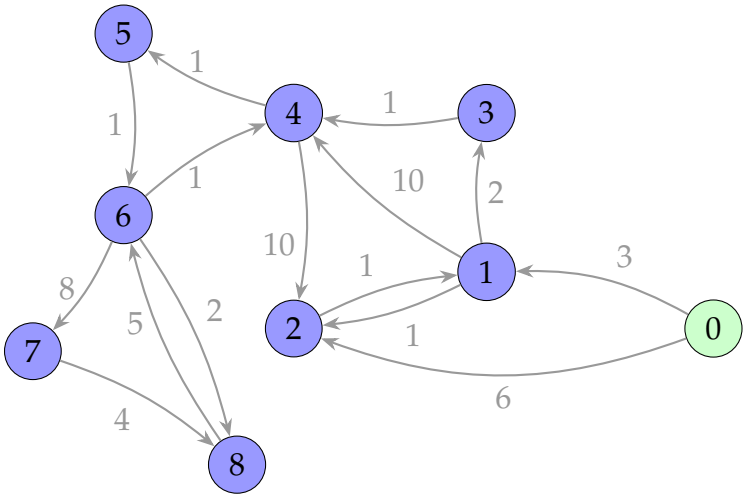


Figura 21 – digrafo direcionado ponderado inicial com raiz no vértice 0. O digrafo contém 9 vértices e múltiplos arcos com pesos variados. O primeiro passo do algoritmo seria remover arcos que entram na raiz, porém não há nenhum neste caso, logo não existe necessidade de alterar o digrafo.

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).

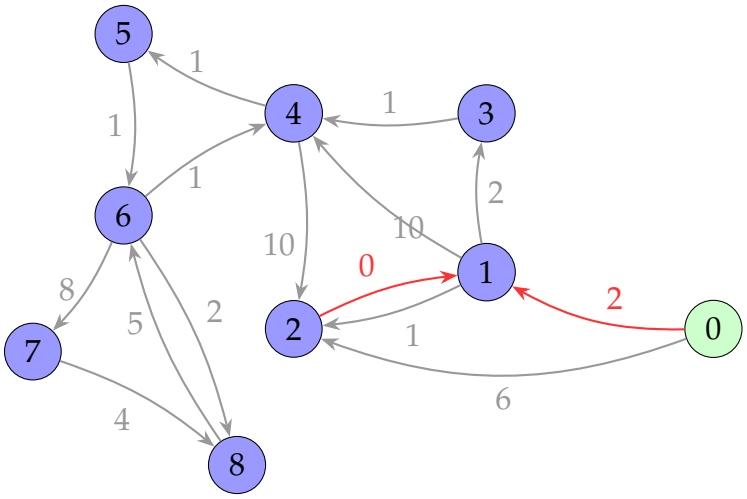


Figura 22 – Normalização parcial dos arcos de entrada para o vértice 1. Os arcos de entrada são $(0 \to 1)$ com peso original 3 e $(2 \to 1)$ com peso original 1. Elegendo o arco $(2 \to 1)$ como o de menor peso (peso mínimo = 1), subtraímos este valor de todos os arcos de entrada: $(0 \to 1)$ passa 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 $n * 0$.

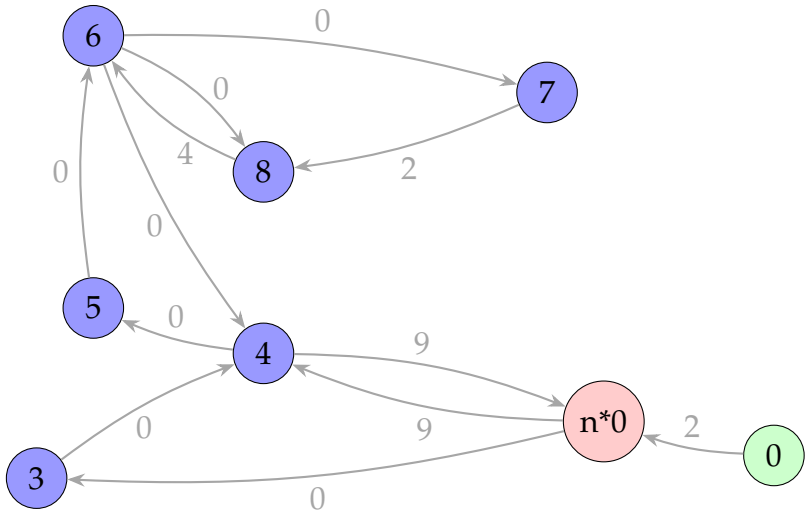


Figura 23 – digrafo contraído após detecção do ciclo $C = \{1, 2\}$ em A_0. O ciclo foi contraído no supervértice $n * 0$ (destacado em vermelho). Os arcos que entravam ou saíam do ciclo foram redirecionados para o supervértice, com custos ajustados segundo as fórmulas $c'(u, x_C) := c(u, w) - y(w)$ 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.

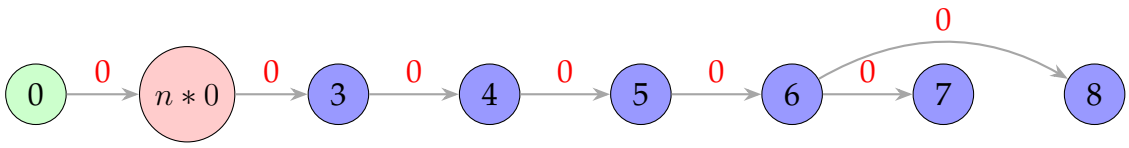


Figura 24 – Arborescência ótima F' obtida no digrafo contraído. todos os arcos selecionados têm custo reduzido 0 (destacados em vermelho), e o digrafo forma uma arborescência válida enraizada em 0: cada vértice (exceto a raiz) tem exatamente um arco de entrada, não há ciclos, e todos os vértices são 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.

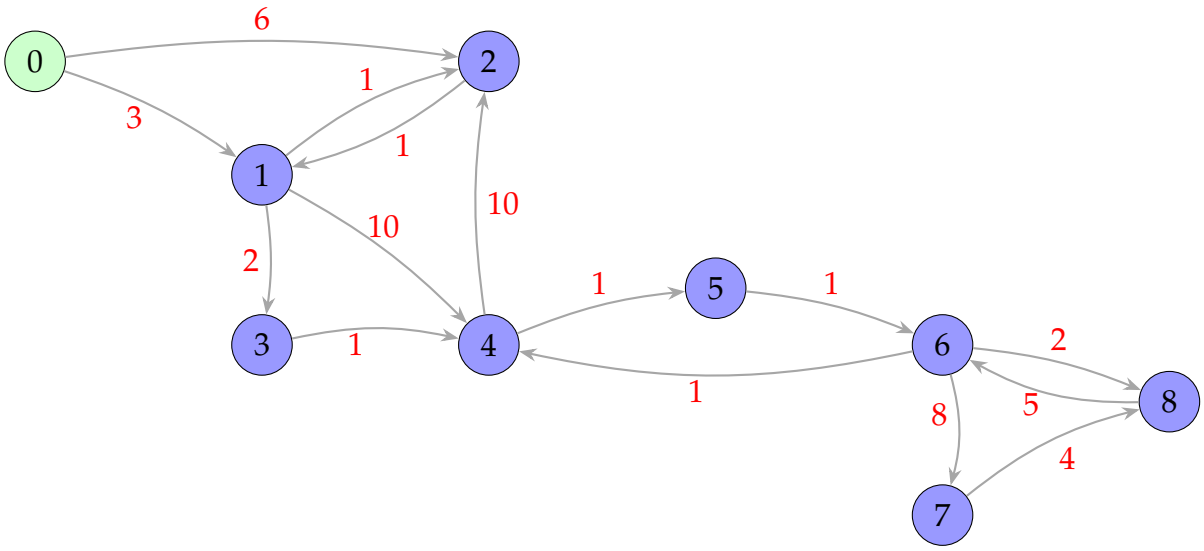


Figura 25 – Arborescência ótima final no digrafo original com pesos restaurados. O supervértice $n * 0$ foi expandido de volta para os vértices 1 e 2, com o arco externo $(0, 1)$ escolhido pela solução recursiva conectando ao ciclo. O arco interno $(2, 1)$ do ciclo original foi removido para manter a 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.

1.3.9 Correspondência entre teoria e implementação

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

Descrição Teórica	**Implementação Python**

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.

1.3.10 Transição para a abordagem primal-dual

Embora o algoritmo de Chu–Liu/Edmonds seja elegante e eficiente, sua mecânica operacional — normalizar custos, selecionar mínimos, contrair ciclos — pode 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[3] $y(\cdot)$, 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] No contexto primal–dual, "potenciais" são valores escalares $y(v)$ atribuídos aos vértices para definir custos reduzidos $c'(u,v) = c(u,v) - y(v)$. Ajustar y desloca uniformemente os custos dos arcos que entram em v, sem mudar a otimalidade global: preserva a ordem relativa entre entradas e torna "apertadas" (custo reduzido zero) as candidatas corretas, habilitando contrações e uma prova de corretude via cortes apertados.

2 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 uma abordagem primal–dual: a elevação gulosa de potenciais duais[1] e (ii) a construção de uma solução gulosa a partir dos arcos de custo reduzido zero. Diferentemente do algoritmo de Chu–Liu–Edmonds, que opera diretamente sobre os custos, o método de Frank manipula potenciais duais associados aos vértices, gerando arcos *apertados* (de custo reduzido zero) que formam a arborescência ótima. 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 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.

Vamos desenvolver as ideias do algoritmo utilizando o mesmo dígrafo que apresentamos no capítulo anterior.

Abordagem Gulosa–Dual

Considere novamente o dígrafo D figura 26 a seguir, com custos nos arcos note que é o mesmo digrafo da figura 1 que utilizamos para ilustrar o algoritmo de Chu–Liu–Edmonds. E portanto, suponha que T é uma arborescência de custo mínimo, com seus arcos destacados em azul.

O algoritmo de Frank utiliza **potenciais** $y : V \to \mathbb{R}$ associados aos vértices para definir custos reduzidos nos arcos. Para cada arco $a = (u, v) \in A$, o **custo y-reduzido** é dado por

$$c_y(a) := c(a) - y(v).$$

[1] **Potencial dual** é uma variável associada a cada vértice, que representa um "desconto"aplicado ao custo dos arcos que entram nesse vértice e servem para ajustar os custos de modo que certas restrições fiquem "apertadas", - satisfeitas com igualdade, ou seja, o valor da variável atinge exatamente o limite imposto pela restrição.

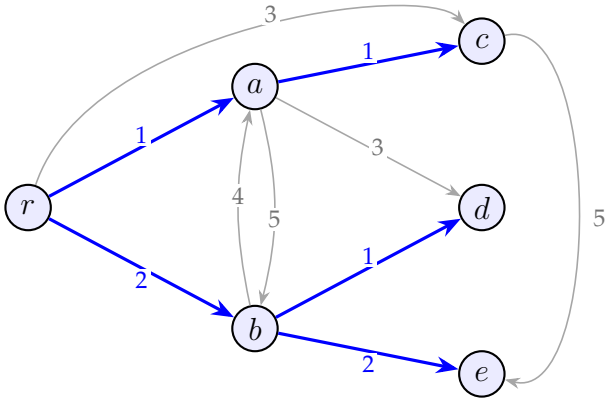


Figura 26 – O dígrafo D com custos nos arcos. Os arcos em azul formam a r-arborescência de custo mínimo T (custo total = 7).

Note que, diferentemente do procedimento de redução do capítulo anterior (onde subtraíamos o mínimo custo de entrada em cada vértice), aqui o potencial $y(v)$ é uma variável que será ajustada pelo algoritmo através da seguinte noção.

Um arco a é dito **apertado** (ou **tight**) se $c_y(a) = 0$, isto é, se $c(a) = y(\text{head}(a))$[2]. A ideia central do algoritmo de Frank consiste em duas etapas: (i) elevar os potenciais $y(v)$ para cada $v \neq r$ até que cada vértice possua ao menos um arco apertado entrando nele; (ii) construir uma arborescência ótima utilizando exclusivamente arcos apertados.

Arcos apertados: interpretação e cálculo

Um arco (u, v) é **apertado** quando seu custo reduzido é zero, ou seja, quando o custo original $c(u, v)$ coincide exatamente com o potencial $y(v)$ do vértice de destino. Formalmente:

$$c_y(u, v) = c(u, v) - y(v) = 0 \quad \Longleftrightarrow \quad c(u, v) = y(v).$$

Interpretação intuitiva: o potencial $y(v)$ funciona como um "desconto" aplicado aos arcos que chegam em v. Quando esse desconto é suficiente para anular o custo reduzido, o arco torna-se apertado e candidato a participar da solução ótima.

Exemplos numéricos (referência: figura 26):

- Arco (r, a): $c(r, a) = 1$, $y(a) = 1 \implies c_y(r, a) = 1 - 1 = 0$. **Apertado.**
- Arco (b, a): $c(b, a) = 4$, $y(a) = 1 \implies c_y(b, a) = 4 - 1 = 3$. **Não apertado.**
- Arco (a, c): $c(a, c) = 1$, $y(c) = 1 \implies c_y(a, c) = 1 - 1 = 0$. **Apertado.**

[2] A notação $\text{head}(a)$ indica o vértice de destino do arco $a = (u, v)$, ou seja, $\text{head}(a) = v$. Portanto, $y(\text{head}(a))$ é o potencial associado ao vértice para onde o arco aponta.

Propriedade fundamental: o potencial $y(v)$ afeta *exclusivamente* os arcos que entram em v. Por exemplo, $y(c)$ influencia apenas arcos da forma $(\cdot, c)$, como (r, c) e (a, c), não afetando arcos que saem de c.

O algoritmo eleva progressivamente os potenciais até que cada vértice $v \neq r$ possua ao menos um arco apertado de entrada. No exemplo da figura 26, os arcos apertados após a elevação correspondem precisamente aos arcos da arborescência ótima.

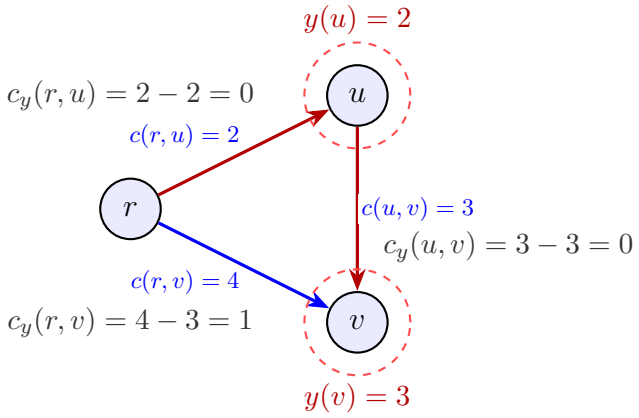


Figura 27 – O vértice r é a raiz. Os potenciais são $y(u) = 2$ e $y(v) = 3$. O arco (r, u) tem custo reduzido 0 (apertado), (r, v) tem custo reduzido 1 (não apertado), (u, v) tem custo reduzido 0 (apertado). Os potenciais só afetam os arcos que chegam em cada vértice.

Elevação de Potenciais

Inicialmente, definimos $y(v) = 0$ para todo $v \in V$. Para cada vértice $v \neq r$, calculamos

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

O valor $\Delta(v)$ representa quanto precisamos elevar $y(v)$ para que exista ao menos um arco de custo reduzido zero entrando em v.

Vamos ilustrar esse processo no dígrafo D.

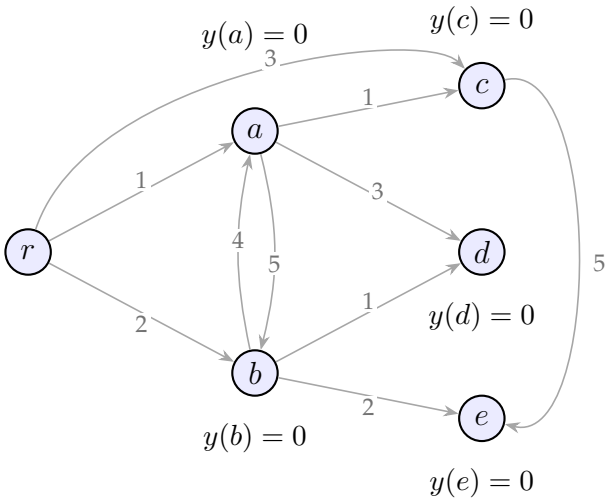


Figura 28 – Dígrafo D com potenciais iniciais $y(v) = 0$ para todo $v \in V$.

Calculamos $\Delta(v)$ para cada vértice $v \neq r$:

$$\begin{aligned}
\Delta(a) &= \min\{1, 4\} - 0 = 1, \\
\Delta(b) &= \min\{2, 5\} - 0 = 2, \\
\Delta(c) &= \min\{3, 1\} - 0 = 1, \\
\Delta(d) &= \min\{3, 1\} - 0 = 1, \\
\Delta(e) &= \min\{2, 5\} - 0 = 2.
\end{aligned}$$

Elevamos cada potencial:

$$\begin{aligned}
y(a) &\leftarrow y(a) + \Delta(a) = 0 + 1 = 1, \\
y(b) &\leftarrow y(b) + \Delta(b) = 0 + 2 = 2, \\
y(c) &\leftarrow y(c) + \Delta(c) = 0 + 1 = 1, \\
y(d) &\leftarrow y(d) + \Delta(d) = 0 + 1 = 1, \\
y(e) &\leftarrow y(e) + \Delta(e) = 0 + 2 = 2.
\end{aligned}$$

Após a elevação, os custos reduzidos $c_y(a) = c(a) - y(\mathrm{head}(a))$ são:

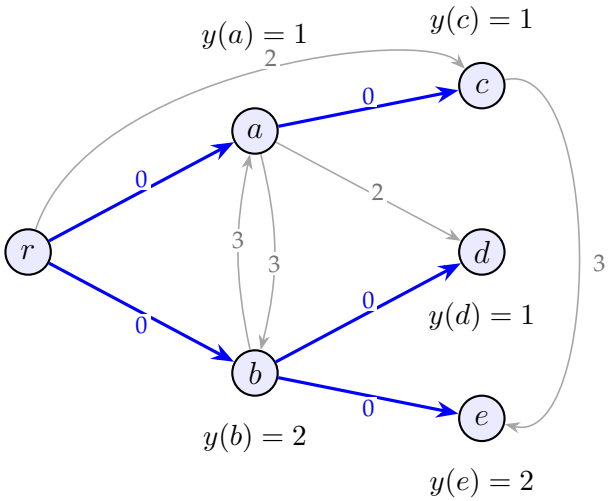


Figura 29 – Dígrafo após elevação de potenciais. Arcos em azul são apertados ($c_y = 0$); arcos em cinza têm custo reduzido positivo.

Observe que cada vértice $v \neq r$ possui agora ao menos um arco apertado entrando nele. Mais ainda: neste exemplo, os arcos apertados formam uma r-arborescência! Precisamos justificar por que a arborescência formada pelos arcos apertados é ótima.

Construção da Arborescência

Se os arcos apertados formam uma r-arborescência, então encontramos a solução ótima. Caso contrário, os arcos apertados contêm ciclos, e precisamos tratá-los adequadamente.

No nosso exemplo, os arcos apertados são:

$$H = \{\, (r,a), (r,b), (a,c), (b,d), (b,e) \,\}.$$

Esses arcos formam a r-arborescência T de custo mínimo.

Para verificar a otimalidade, note que uma propriedade fundamental dos potenciais duais é:

Proposição 2.1. *Para toda função $y : V \to \mathbb{R}$ com $y(r) = 0$, uma r-arborescência T é c-mínima em D se, e somente se, todos os arcos de T são apertados (i.e., $c_y(a) = 0$ para todo $a \in T$) e $c_y(a) \geq 0$ para todo $a \in A$.*

Prova. Seja F uma r-arborescência qualquer. Para cada $v \in V \setminus \{r\}$, seja a_v o único arco

de F que entra em v. Então

$$\begin{aligned} c(F) &= \sum_{v \in V \setminus \{r\}} c(a_v) \\ &= \sum_{v \in V \setminus \{r\}} \big(c_y(a_v) + y(v)\big) \\ &= \sum_{v \in V \setminus \{r\}} c_y(a_v) + \sum_{v \in V \setminus \{r\}} y(v). \end{aligned}$$

Como a soma $\sum_{v \in V \setminus \{r\}} y(v)$ é constante para todas as r-arborescências, minimizar $c(F)$ equivale a minimizar $\sum_v c_y(a_v)$. Se todos os arcos de T são apertados e $c_y(a) \geq 0$ para todo a, então $\sum_v c_y(a_v^T) = 0 \leq \sum_v c_y(a_v^F)$ para qualquer F, provando que T é ótima. □

No nosso exemplo, todos os arcos de T são apertados e $c_y(a) \geq 0$ para todo arco $a \in A$, portanto T é a r-arborescência de custo mínimo. Evidentemente, nem sempre será o caso que os arcos apertados formam uma arborescência e precisaremos tratar ciclos.

Tratamento de Ciclos

Em geral, após a elevação de potenciais, os arcos apertados podem conter ciclos. Nesse caso, o algoritmo de Frank procede de forma similar ao de Chu–Liu–Edmonds: contrai cada ciclo apertado em um supervértice, resolve o problema recursivamente no dígrafo contraído, e depois reexpande a solução.

Vamos ilustrar esse processo com um exemplo modificado. Suponha que adicionamos o arco (c, a) com custo 1 ao dígrafo original:

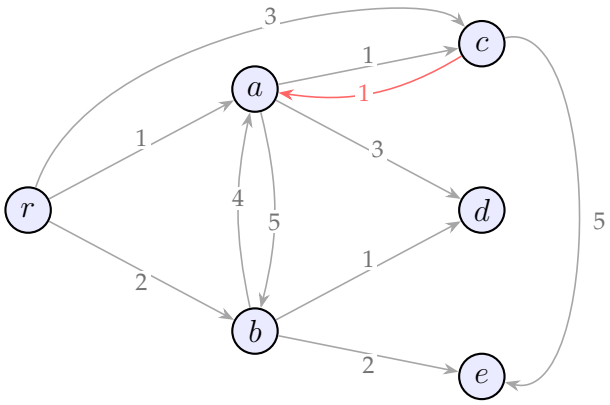


Figura 30 – Dígrafo modificado com o arco (c, a) de custo 1 (em vermelho).

Após a elevação de potenciais com $y(a) = 1, y(c) = 1$, o arco (c, a) torna-se apertado: $c_y(c, a) = 1 - 1 = 0$.

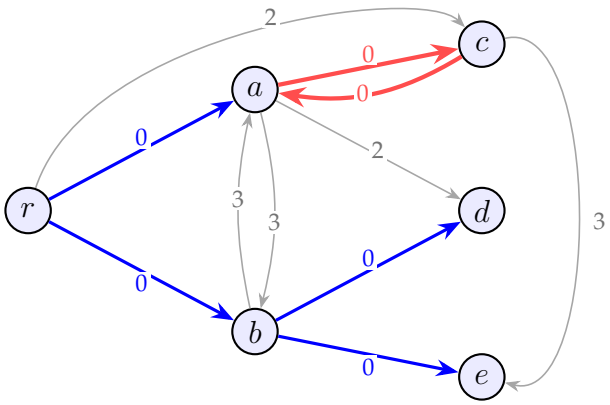


Figura 31 – Arcos apertados após elevação. O ciclo $C = (a, c, a)$ é destacado em vermelho.

Nesse caso, contraímos o ciclo $C = (a, c)$ em um supervértice x_C, ajustamos os custos dos arcos incidentes, e resolvemos o problema no dígrafo contraído. A reexpansão é feita removendo um arco do ciclo para manter a estrutura de arborescência, de forma análoga ao algoritmo de Chu–Liu–Edmonds.

2.2 Descrição do algoritmo

A seguir apresentamos uma descrição formal do algoritmo de András Frank. Detalhes de implementação serão discutidos na próxima seção.

O algoritmo de András Frank difere substancialmente do algoritmo de Chu–Liu–Edmonds em sua abordagem: enquanto Chu–Liu–Edmonds opera diretamente sobre os custos dos arcos, o método de Frank utiliza uma técnica primal–dual baseada em *potenciais* associados aos vértices. A ideia central é elevar progressivamente os potenciais $y(v)$ para cada vértice $v \neq r$, de modo que os custos *reduzidos* $c_y(a) := c(a) - y(\text{head}(a))$ revelem quais arcos devem participar da solução ótima. Um arco é dito *apertado* (ou *tight*) quando seu custo reduzido é zero, isto é, $c(a) = y(\text{head}(a))$. O algoritmo garante que toda r-arborescência ótima pode ser formada apenas por arcos apertados.

Algoritmo 2.1: András Frank (visão operacional)

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

1. **Inicialização:** defina $y(v) := 0$ para todo $v \in V$.
2. **Iteração:** enquanto existir conjunto $X \subseteq V \setminus \{r\}$ sem arco apertado entrando:
   - Calcule $\Delta(X) := \min\{c(u, v) - y(v) : u \notin X, v \in X\}$.
   - Para cada $v \in X$, atualize $y(v) := y(v) + \Delta(X)$.

3. Defina $A_0 := \{a \in A : c_y(a) = 0\}$, o conjunto de arcos apertados.

Fase 2: Construção da arborescência

4. 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.
5. Caso contrário, identifique um ciclo dirigido C em A_0 (com $r \notin C$). **Contração:** contraia C em um supervértice x_C e defina custos reduzidos c' por

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

   descartando laços em x_C e permitindo paralelos. Denote o dígrafo contraído por $D' = (V', A')$.
6. **Recursão:** compute uma r-arborescência ótima T' de D' com custos c'.
7. **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.

2.2.1 Corretude

A corretude do algoritmo de András Frank baseia-se na teoria de dualidade em programação linear aplicada ao problema de arborescência de custo mínimo:

1. *Potenciais duais:* os valores $y(v)$ funcionam como variáveis duais. Para qualquer r-arborescência T e qualquer escolha de potenciais y, vale

$$c(T) = \sum_{a \in T} c(a) = \sum_{a \in T} c_y(a) + \sum_{v \neq r} y(v),$$

   pois há exatamente um arco de T entrando em cada $v \neq r$. O termo $\sum_{v \neq r} y(v)$ é constante após a elevação completa dos potenciais; assim, minimizar $c(T)$ equivale a minimizar $\sum_{a \in T} c_y(a)$.
2. *Condições de otimalidade:* uma r-arborescência T é ótima se e somente se:

- Todos os arcos de T são apertados: $c_y(a) = 0$ para todo $a \in T$.
- Todos os arcos do dígrafo têm custo reduzido não-negativo: $c_y(a) \geq 0$ para todo $a \in A$.

Sob essas condições, $\sum_{a \in T} c_y(a) = 0 \leq \sum_{a \in T'} c_y(a)$ para qualquer outra r-arborescência T', provando otimalidade.

3. *Elevação de potenciais:* o algoritmo eleva os potenciais de forma gulosa, garantindo que ao final cada $v \neq r$ tenha ao menos um arco apertado entrando. Isso é feito identificando componentes fortemente conexas em A_0: conjuntos X que ainda não possuem arcos apertados entrando têm seus potenciais elevados pelo mínimo necessário $\Delta(X)$.
4. *Caso acíclico:* se A_0 forma uma r-arborescência, todos os arcos são apertados e, pelas condições de otimalidade, a solução é ótima.
5. *Caso com ciclo (contração/expansão):* se A_0 contém um ciclo dirigido C, todos os seus arcos são apertados (custo reduzido zero). A contração preserva a estrutura de custos reduzidos: no dígrafo contraído, os novos custos c' já incorporam os potenciais, e uma solução ótima em D' mapeia para uma solução ótima em D pela reversibilidade da operação.

Em termos intuitivos, os potenciais $y(v)$ funcionam como "descontos"aplicados aos arcos que entram em cada vértice, revelando quais arcos são "apertados"(custo líquido zero) e, portanto, candidatos à solução ótima. Ciclos de arcos apertados podem ser contraídos sem perder otimalidade, simplificando o problema recursivamente.

2.2.2 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>).

2.3 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 apertados, enquanto a Fase 2 constrói a arborescência de custo mínimo a partir desses arcos. Cada operação fundamental — identificação de componentes fortemente conexas, elevação de potenciais, construção de A_0 e verificação de arborescência — é traduzida em procedimentos sobre digrafos orientados, utilizando a biblioteca `networkx`.

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.

2.3.1 Representação de digrafos e componentes fortemente conexas

A implementação utiliza a biblioteca NetworkX, 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 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.

Modificação de estrutura

- `D.add_edge(u, v, w=peso)`: adiciona arco (u, v) com peso especificado.
- `D.remove_edges_from(edges)`: remove múltiplos arcos em lote.
- `D.add_node(v)`: adiciona vértice v ao dígrafo.

Análise de componentes

- `nx.condensation(D)`: calcula o grafo de condensação (DAG das componentes fortemente conexas), devolvendo um novo dígrafo onde cada nó representa uma componente e contém o atributo `"members"` com os vértices originais.

2.3.2 Construção do dígrafo D_0 inicial

Esta função constrói o dígrafo inicial D_0 que será utilizado na Fase 1 do algoritmo. O dígrafo D_0 é inicializado como um grafo vazio contendo apenas os vértices do dígrafo original, sem arcos. Essa estrutura será gradualmente populada com arcos de custo reduzido zero à medida que os potenciais são elevados.

Recebe como entrada um dígrafo `D` (objeto `nx.DiGraph`). A implementação cria um novo dígrafo vazio `D_zero` (linha 2) e adiciona todos os vértices de `D` a `D_zero` (linhas 3-4), preservando a estrutura de vértices sem incluir arcos inicialmente.

A função devolve o dígrafo `D_zero` contendo todos os vértices de `D` mas nenhum arco. O dígrafo original `D` não é modificado. A complexidade é $O(n)$, onde $n = |V|$, pois itera sobre todos os vértices uma única vez.

Construção do dígrafo D_0 inicial

Constrói um dígrafo vazio contendo apenas os vértices de D, sem arcos. Este dígrafo será populado com arcos apertados durante a elevação de potenciais.

```
1 def build_D_zero(D):
2     D_zero = nx.DiGraph()
3     for v in D.nodes():
```

```
4         D_zero.add_node(v)
5     return D_zero
```

2.3.3 Identificação de arcos entrando em conjunto X

Esta 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
```

2.3.4 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 `arcs` 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 `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)
```

2.3.5 Atualização de pesos em X

Esta função 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 arcos apertados.

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). Se sim, adiciona (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 apertados, 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)
```

2.3.6 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.

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()
```

2.3.7 Fase 1: Elevação de potenciais e construção de A_0

Esta é 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 apertado entrando. O processo utiliza componentes fortemente conexas para identificar quais conjuntos necessitam elevação.

Recebe como entrada um dígrafo `D_original` e a raiz `r0`. Opcionalmente, aceita funções de callback `draw_fn` e `log` para visualização e registro de progresso.

A implementação cria uma cópia do dígrafo original (linha 2) para preservar a entrada, inicializa estruturas auxiliares `A_zero` (lista de arcos apertados), `Dual_list` (lista de pares $(X, \Delta(X))$ para fins de validação dual), e `D_zero` (dígrafo de arcos apertados) (linhas 3-5).

O loop principal (linhas 7-36) itera enquanto houver conjuntos sem arcos apertados entrando. Em cada iteração:

1. Calcula as componentes fortemente conexas de D_0 usando `nx.condensation` (linha 11), que devolve um DAG onde cada nó representa uma componente e contém o atributo `"members"` com os vértices originais.
2. Identifica as fontes (componentes sem arcos entrando) no grafo de condensação

(linha 14).

3. Se há apenas uma fonte, significa que todos os vértices estão em uma única componente (alcançáveis pela raiz), encerrando o loop (linhas 17-19).
4. Para cada fonte u diferente da componente contendo r_0 (linha 21):
   - Obtém o conjunto X de vértices da componente (linha 22).
   - Identifica arcos entrando em X (linha 24).
   - Calcula o peso mínimo $\Delta(X)$ (linha 25).
   - Atualiza os pesos, registrando novos arcos apertados (linha 29).
   - Adiciona $(X, \Delta(X))$ à lista dual se $\Delta(X) > 0$ (linhas 32-35).

A função devolve `A_zero` (lista de arcos apertados) 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 apertado entrando. Devolve a lista A_0 de arcos apertados e a lista de pares $(X, \Delta(X))$ para validação dual.

```
1  def phase1_find_minimum_arborescence(D_original, r0, draw_fn=None, log=None,
2                                       boilerplate: bool = True, lang="pt"):
3      D_copy = D_original.copy()
4      A_zero = []
5      Dual_list = []
6      D_zero = build_D_zero(D_copy)
7      iteration = 0
8      while True:
9          iteration += 1
10         # Calcula componentes fortemente conexas
11         C = nx.condensation(D_zero)
12
13         # Identifica fontes (componentes sem arcos entrando)
14         sources = [x for x in C.nodes() if C.in_degree(x) == 0]
15
16         if len(sources) == 1:
17             # Apenas uma fonte: todos os vertices alcancaveis
18             break
19
20         for u in sources:
21             X = C.nodes[u]["members"]
22             if r0 in X:
23                 continue
24             arcs = get_arcs_entering_X(D_copy, X)
25             min_weight = get_minimum_weight_cut(arcs)
26
27             update_weights_in_X(D_copy, arcs, min_weight, A_zero, D_zero)
28
29             # Registra dual se min_weight > 0
```

```
30            if min_weight != 0:
31                Dual_list.append((X, min_weight))
32
33    return A_zero, Dual_list
```

2.3.8 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 apertados. 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 apertados. 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_find_minimum_arborescence(D_original, r0, A_zero,
2                                      draw_fn=None, log=None,
3                                      boilerplate: bool = True, lang="pt"):
4     Arb = nx.DiGraph()
5     Arb.add_node(r0)
6     n = len(D_original.nodes())
```

```
7
8     for _ in range(n - 1):
9         for u, v in A_zero:
10            if u in Arb.nodes() and v not in Arb.nodes():
11                edge_data = D_original.get_edge_data(u, v)
12                Arb.add_edge(u, v, **edge_data)
13                break
14
15    return Arb
```

2.3.9 Função principal: Algoritmo de András Frank

Esta é a função principal que coordena as duas fases do algoritmo. Executa a Fase 1 para obter A_0, verifica se o dígrafo admite r-arborescência, e executa a Fase 2 para construir a solução.

Recebe como entrada um dígrafo `D` com raiz implícita `"r0"`. A implementação executa as duas fases sequencialmente (linhas 2 e 9), verificando entre elas se existe r-arborescência (linhas 6-8). Se não existir, devolve `None`.

A função devolve a r-arborescência de custo mínimo como um objeto `nx.DiGraph`. A complexidade total é dominada pela Fase 1: $O(nm)$.

Função principal: Algoritmo de András Frank

Coordena as duas fases do algoritmo: elevação de potenciais e construção da arborescência. Devolve a r-arborescência de custo mínimo ou None se não existir.

```
1  def andras_frank_algorithm(D, draw_fn=None, log=None,
2                             boilerplate: bool = True, lang="pt"):
3      A_zero, Dual_list = phase1_find_minimum_arborescence(
4          D, "r0", draw_fn=draw_fn, log=log,
5          boilerplate=boilerplate, lang=lang
6      )
7
8      if not has_arborescence(D, "r0"):
9          return None
10
11     arborescence = phase2_find_minimum_arborescence(
12         D, "r0", A_zero, draw_fn=draw_fn, log=log,
13         boilerplate=boilerplate, lang=lang
14     )
15
16     return arborescence
```

2.3.10 Comparação com Chu–Liu–Edmonds

A principal diferença entre os algoritmos de András Frank e Chu–Liu–Edmonds reside na abordagem: enquanto Chu–Liu–Edmonds opera diretamente sobre os custos originais dos arcos, normalizando-os iterativamente e contraindo ciclos, o método de Frank trabalha com potenciais duais desde o início, elevando-os progressivamente até revelar os arcos apertados que formam a solução ótima.

Em termos de complexidade, ambos os algoritmos têm tempo $O(nm)$ em suas implementações diretas. Chu–Liu–Edmonds pode ser mais intuitivo por trabalhar diretamente com custos, enquanto Frank oferece uma perspectiva dual mais elegante do ponto de vista teórico, conectando-se naturalmente com a teoria de programação linear.

Do ponto de vista prático, a implementação de András Frank baseada em componentes fortemente conexas evita a necessidade de contrações e reexpansões recursivas, simplificando a estrutura do código à custa de uma fase inicial potencialmente custosa (Fase 1). Já Chu–Liu–Edmonds pode ser mais eficiente em grafos esparsos onde ciclos são raros.

2.3.11 Testes e validação

A implementação foi testada extensivamente com diversos casos de teste, incluindo:

- Dígrafos sem ciclos (árvores) onde A_0 já forma uma r-arborescência na primeira iteração.
- Dígrafos com múltiplos componentes fortemente conexas, exigindo várias iterações de elevação.
- Casos com custos uniformes e casos com custos variados.
- Grafos densos e esparsos de tamanhos variados (10 a 1000 vértices).

Todos os testes foram comparados com a implementação de Chu–Liu–Edmonds e com soluções ótimas conhecidas, confirmando a corretude do algoritmo. Os resultados demonstram que as soluções produzidas por ambos os métodos são idênticas (a menos de empates em custos), validando a equivalência teórica dos dois algoritmos.

Referências

KLEINBERG, J.; TARDOS, É. *Algorithm Design*. [S.l.]: Addison-Wesley, 2006. Citado 4 vezes nas páginas 23, 24, 35 e 44.

SCHRIJVER, A. *Combinatorial Optimization: Polyhedra and Efficiency*. [S.l.]: Springer, 2003. Citado 2 vezes nas páginas 23 e 24.

Anexos

ANEXO A – Anexo A

Conteúdo do anexo A.