Lorena Silva Sampaio, Samira Haddad

Algoritmos para o problema da arborescência geradora miníma: Uma aplicação didática interativa

Brasil

2025

Lorena Silva Sampaio, Samira Haddad

Algoritmos para o problema da arborescência geradora miníma: Uma 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 Preliminares

Neste capítulo, reunimos as noções básicas necessárias para compreensão completa do texto.

1.1 Digrafos

Começamos por introduzir a noção de digrafo. Um **digrafo** D é um par (V, A), em que V é um conjunto finito de elementos chamados **vértices** e A, chamado de conjunto dos **arcos**, é um subconjunto de

$$\{(u, v) \in V \times V : u \neq v\}.$$

Escrevemos $V(D)$ e $A(D)$ para denotar, respectivamente, o conjunto dos vértices e o conjunto dos arcos de D.

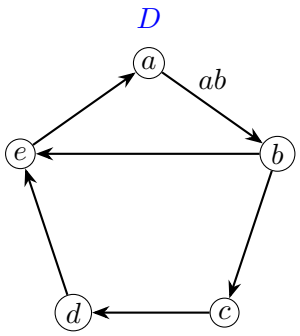


Figura 1 – A figura ilustra um digrafo D cujo conjunto dos vértices é $\{a, b, c, d, e\}$ e cujo conjunto dos arcos é $\{ab, bc, cd, de, ea, be\}$.

Para um arco $a := (u, v)$ de D — o que costumamos abreviar como uv — dizemos que u é a **cauda** (ou **ponta inicial**) de a e v é a **cabeça** (ou **ponta final**) de a.

Seja X um subconjunto de $V(D)$. Dizemos que um arco a **entra** em X se a ponta final de a está fora de X e a inicial está em X. Por outro lado, um arco a sai de X se a ponta inicial de a está em X e a final está fora de X. De forma análoga, dizemos que um subconjunto B de $A(D)$ **entra** em X se existe um arco de B que entra em X. O conjunto dos arcos de D que entram em X é denotado por $\delta_D^-(X)$ (ou $\delta^-(X)$ quando o contexto permitir). De forma similar, o conjuntos dos arcos que saem de X é denotado por $\delta_D^+(X)$. Por brevidade, para cada vértice $v \in V(D)$, escrevemos $\delta_D^-(v)$ no lugar de $\delta_D^-(\{v\})$. A mesma convenção é usada para δ_D^+.

Um **digrafo ponderado** é um par (D, w), em que D é um digrafo e $w : A(D) \to \mathbb{R}$ é uma função que associa a cada arco $a \in A(D)$ um **custo** (ou **peso**) real $w(a)$.

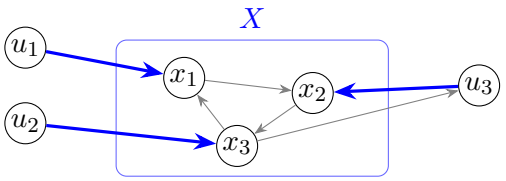


Figura 2 – Os arcos em azul entram em um subconjunto $X \subseteq V(D)$.

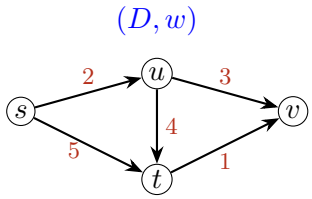


Figura 3 – Exemplo de digrafo ponderado (D, w): cada arco recebe um custo real $w(a)$.

Um digrafo H é um **subdigrafo** de um digrafo D se $V(H) \subseteq V(D)$ e $A(H) \subseteq A(D)$.

Seja D um digrafo e seja $B \subseteq A(D)$. O subdigrafo de D **gerado** por B, denotado $D[B]$, é o par (W, B), em que W é o conjunto dos vértices que são pontas de arcos de B, isto é,

$$W = \{v \in V(D) : \text{existe } u \in V \text{ tal que } uv \in A(D) \text{ ou } vu \in A(D)\}.$$

1.2 Caminhos

Um **caminho** P em um digrafo D é uma sequência de vértices de D

$$(u_0, u_1, \ldots, u_k),$$

em que $k \geq 0$ e, para cada $i \in \{0, 1, \ldots, k-1\}$, $u_i u_{i+1}$ é um arco de D. Dizemos que P é um caminho **de** u_0 **até** u_k para destacar a **origem** u_0 de P e o **destino** u_k de P. Um caminho P é dito **simples** se seus vértices são dois a dois distintos. Um caminho é **fechado** se sua origem e seu destino coincidem. Finalmente, um caminho fechado $P := (u_0, u_1, \ldots, u_k)$ é um **ciclo** se $(u_0, u_1, \ldots, u_{k-1})$ é um caminho simples.

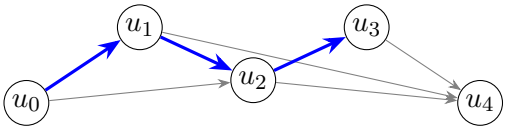


Figura 4 – A figura ilustra um caminho simples (u_0, u_1, u_2, u_3) em um digrafo.

A noção de caminho permite definir o território de um vértice em um digrafo. Seja D um digrafo e seja $r \in V(D)$. O **território** de r em D é o conjunto dos vértices $v \in V(D)$ tais que existe um caminho de r até v em D. O seguinte fato é bem conhecido. Para enunciá-lo é conveniente introduzir a seguinte definição. Um r**-conjunto** de D é um subconjunto não vazio X de $V(D)$ tal que $r \notin V(D)$.

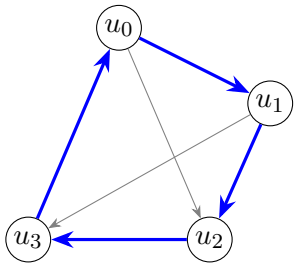


Figura 5 – A figura ilustra um ciclo $(u_0, u_1, u_2, u_3, u_0)$ em um digrafo.

Proposição 1.1. *Seja D um digrafo e $r \in V(D)$. O território de r em D é igual a $V(D)$ se, e somente se, para cada r-conjunto X existe ao menos um arco de D que entra em X.* □

1.3 Arborescências

Podemos agora introduzir um dos objetos fundamentais deste trabalho: as arborescências. Dizemos que um digrafo D é uma **arborescência** se existe um vértice $r \in V(D)$ tal que, para cada vértice $v \in V(D)$, existe um único caminho em D de r até v. Nesse caso, chamamos r de **raiz** de D. Para destacar o papel de r nesta definição, dizemos que D é uma **r-arborescência**.

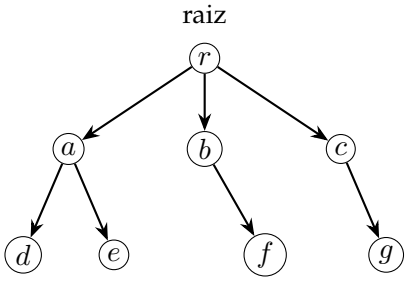


Figura 6 – A figura ilustra uma r-arborescência.

Um subdigrafo T de um digrafo D é uma **arborescência de** D se T é uma arborescência. Dizemos que T é uma arborescência **geradora** de D se, além disso, $V(T) = V(D)$.

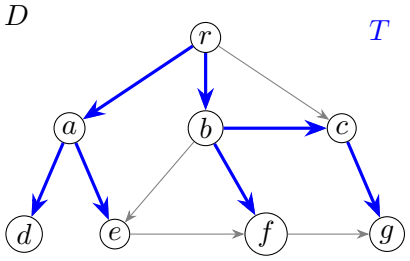


Figura 7 – O subdigrafo T (em azul) é uma arborescência geradora do digrafo D, que inclui tanto os arcos azuis quanto os arcos cinza.

Seja (D, w) um digrafo ponderado. Para todo subconjunto $B \subseteq A(D)$, definimos

$$w(B) := \sum_{b \in B} w(b).$$

Quando H é um subdigrafo de D, escrevemos $w(H)$ para abreviar $w(A(H))$.

No contexto de arborescências de um digrafo D, é comum identificarmos um subconjunto $B \subseteq A(D)$ com o subdigrafo $D[B]$. Assim, dizemos que $B \subseteq A(D)$ é uma **arborescência de** D se $D[B]$ é uma arborescência de D.

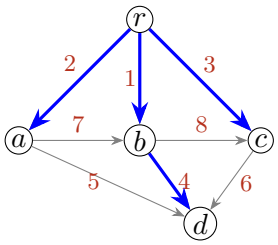


Figura 8 – A figura ilustra um subconjunto B (de arcos azuis) de custo $w(B) = 2 + 1 + 3 + 4 = 10$ visto como uma arborescência geradora de D.

Seja D um digrafo e seja r um vértice de D. Uma **cobertura de r-conjuntos** é um subconjunto $B \subseteq A(D)$ tal que B entra em todo r-conjunto de D. É claro que uma r-arborescência geradora de D é uma cobertura de r-conjuntos. A próxima proposição afirma que $B \subseteq A(D)$ é uma r-arborescência geradora de D se, e somente se, B é uma cobertura *minimal* de r-conjuntos de D. Isto significa que: (i) B é uma cobertura de r-conjuntos e (ii) para cada $a \in B$, o conjunto $B \setminus \{a\}$ não é uma cobertura de r-conjuntos de D.

Proposição 1.2. *Seja D um digrafo e seja r um vértice de D. Um subconjunto $B \subseteq A(D)$ é uma r-arborescência geradora de D se, e somente se, B é uma cobertura* minimal *de r-conjuntos de D.* □

1.4 Problema da arborescência de custo mínimo

Finalmente, podemos enunciar o problema que constitui o objeto de estudo deste trabalho.

Problema da r-arborescência geradora de custo mínimo

Dado um digrafo ponderado (D, w) e um vértice $r \in V(D)$, deseja-se encontrar, se existir, uma r-arborescência geradora T de D tal que

$$w(T) \leq w(F)$$

para toda r-arborescência geradora F de D.

É uma chateação lidar com a posssibilidade de que uma r-arborescência geradora de D pode não existir. Além disso, decidir se uma r-arborescência existe é tarefa simples: basta determinar o território do vértice r em D, o que pode ser feito por meio de qualquer algoritmo de busca. Assim, o problema só se coloca quando existe ao menos uma r-arborescência. Nesse caso, há ainda uma hipótese que pode ser adotada sem perda de generalidade: podemos supor que nenhum arco de D entra em r, uma vez que nenhuma r-arborescência contém um arco entrando em r. Para evitar essas repetições, introduzimos a seguinte definição. Dizemos que uma tripla (D, w, r) é um **r-digrafo ponderado** se

- (D, w) é um digrafo ponderado;
- r é um vértice de D;
- $\delta^-(r) = \varnothing$; e
- D possui uma r-arborescência.

Para um r-digrafo ponderado (D, w, r), dizemos que uma r-arborescência geradora é de **custo mínimo** em (D, w) se

$$w(T) \leq w(F)$$

para toda r-arborescência geradora F de D.

1.5 Contração

A operação de contração de um conjunto de vértices é fundamental para o algoritmo de Chu–Liu–Edmonds e é o assunto que passaremos a tratar agora.

Seja D um digrafo e seja $\mathscr{P}$ uma partição de $V(D)$. Definimos o digrafo obtido de D pela **contração** de $\mathscr{P}$, denotado por $D/\mathscr{P}$, como segue. Seu conjunto de vértices é

$$V(D/\mathscr{P}) := \mathscr{P},$$

e seu conjunto de arcos é

$$A(D/\mathscr{P}) := \{\, XY \in \mathscr{P} \times \mathscr{P} : X \neq Y \text{ e existem } x \in X,\ y \in Y \text{ tais que } xy \in A(D) \,\}.$$

Em outras palavras, $D/\mathscr{P}$ é o digrafo cujo conjunto de vértices é $\mathscr{P}$ e em que há um arco de X para Y, com $X, Y \in \mathscr{P}$ e $X \neq Y$, se, e somente se, existe um arco $xy \in A(D)$ com $x \in X$ e $y \in Y$.

Definimos agora em que consiste contrair um conjunto não vazio de vértices de um digrafo. Seja D um digrafo e seja $\varnothing \neq X \subseteq V(D)$. A **contração** de X em D, denotada por D/X, é o digrafo $D/\mathscr{P}$, em que

$$\mathscr{P} := \big\{\{u\} : u \in V(D) \setminus X\big\} \cup \{X\}.$$

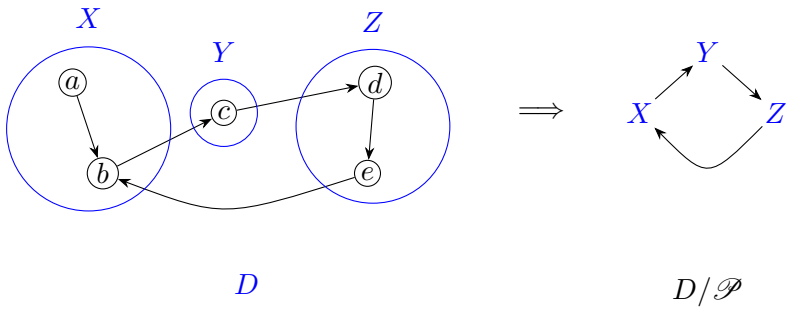


Figura 9 – Contração de um digrafo D pela partição $\mathscr{P} = \{X, Y, Z\}$, em que $X := \{a, b\}$, $Y := \{c\}$ e $Z := \{d, e\}$.

Informalmente, em D/X todos os vértices de X são identificados em um único vértice, enquanto os vértices fora de X permanecem inalterados.

Nesse caso, no contexto do digrafo D/X, vamos identificar o conjunto $\{u\}$ com o próprio elemento u, para cada $u \in V(D) \setminus X$. Também vamos identificar o conjunto X com um vértice $v \notin V(D)$. Para explicitar essa identificação, escreveremos $D/X \mapsto v$ em vez de simplesmente D/X.

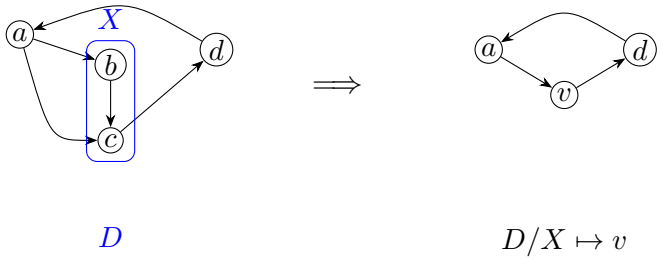


Figura 10 – Contração de um subconjunto $X \subseteq V(D)$ em um digrafo D. À esquerda, o subconjunto $X = \{b, c\}$ é destacado; à direita, os vértices de X foram identificados em um único vértice v, obtendo-se o digrafo $D/X \mapsto v$.

Considere agora um digrafo ponderado (D, w) e uma partição $\mathscr{P}$ de $V(D)$. Definimos o digrafo ponderado $(D/\mathscr{P}, w/\mathscr{P})$ pondo

$$(w/\mathscr{P})(XY) := \min\{\, w(xy) : x \in X,\ y \in Y \,\}$$

para cada $XY \in A(D/\mathscr{P})$. Ou seja, o custo de cada arco XY de $D/\mathscr{P}$ é o menor dos custos dos arcos $xy \in A(D)$ tais que $x \in X$ e $y \in Y$.

Quando X é um subconjunto não vazio de vértices de D, escrevemos $(D/X \mapsto v, w/X \mapsto v)$ para denotar a contração de X na qual o conjunto X é identificado com o vértice $v \notin V(D)$.

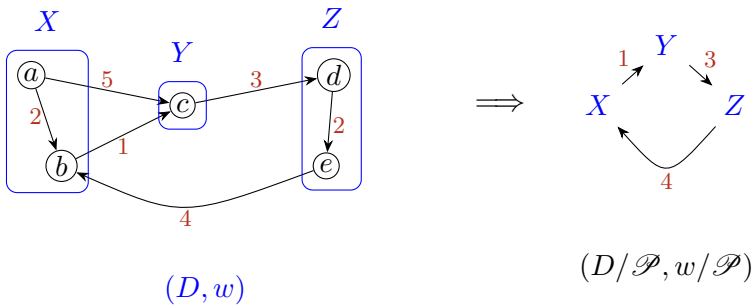


Figura 11 – Contração de um digrafo ponderado (D, w) pela partição $\mathscr{P} = \{X, Y, Z\}$, em que $X := \{a, b\}$, $Y := \{c\}$ e $Z := \{d, e\}$. O peso de cada arco XY em $D/\mathscr{P}$ é o mínimo dos pesos dos arcos de D que vão de um vértice de X para um vértice de Y.

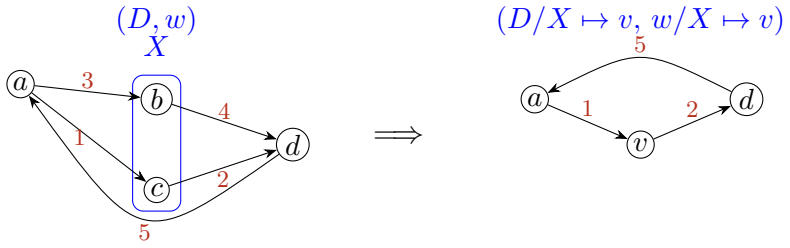


Figura 12 – Contração ponderada de um subconjunto $X \subseteq V(D)$. À esquerda, o digrafo (D, w) com $X = \{b, c\}$ destacado. À direita, o digrafo $(D/X \mapsto v, w/X)$, em que os vértices de X foram identificados em um único vértice v e cada peso w/X é o mínimo dos pesos dos arcos de D entre X e os demais vértices.

1.6 Componentes fortes

Seja D um digrafo. Uma **componente forte** de D é um subconjunto maximal $S \subseteq V(D)$ tal que, para cada $s, t \in S$, existe um caminho de s até t em D e um caminho de t até s em D. Nesse caso, qualquer caminho de s até t (por $s, t \in S$) tem todos os seus vértices contidos em S.

O conjunto das componentes fortes de D é denotado por $\mathscr{C}(D)$, ou simplesmente por $\mathscr{C}$ quando não houver risco de ambiguidade. A **condensação** de D é o digrafo $D/\mathscr{C}$, cujos vértices são as componentes fortes de D e em que há um arco do vértice S para o vértice T de $D/\mathscr{C}$ se, e somente se, existe um arco de D que sai de um vértice de S e entra em um vértice de T.

Dizemos que $S \in \mathscr{C}$ é uma **fonte** de $\mathscr{C}$ se nenhum arco de $D/\mathscr{C}$ entra em S. É bem sabido que $D/\mathscr{C}$ é um digrafo livre de ciclos (isto é, um digrafo acíclico).

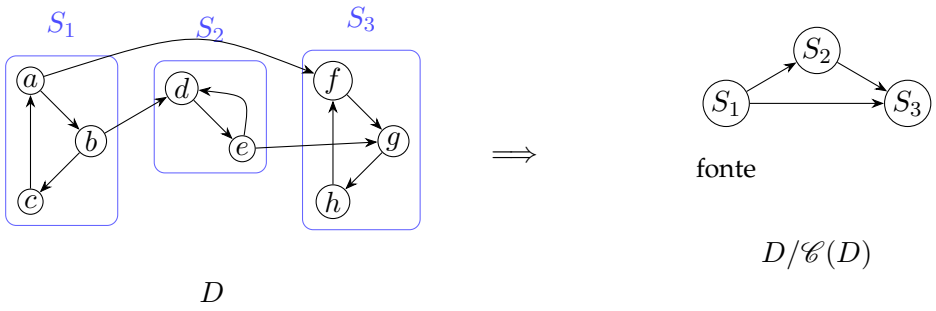


Figura 13 – Componentes fortes S_1, S_2, S_3 de um digrafo D e sua condensação $D/\mathscr{C}(D)$, um digrafo acíclico em que S_1 é fonte.

2 Algoritmo de Chu–Liu–Edmonds

Neste capítulo apresentaremos o algoritmo de Chu–Liu–Edmonds (**??**) (**??**), que determina uma r-arborescência geradora de custo mínimo em um r-digrafo ponderado. O algoritmo baseia-se em duas operações fundamentais: (i) a redução gulosa dos custos dos arcos e (ii) a contração de ciclos. Essas operações permitem resolver recursivamente uma instância menor do problema e, em seguida, estender a solução obtida para o problema original.

O propósito deste capítulo é fornecer uma descrição precisa tanto do algoritmo quanto da implementação desenvolvida neste trabalho.

2.1 O algoritmo

O problema que nos interessa consiste em, dado um r-digrafo ponderado (D, w, r) (veja a página 17), encontrar uma r-arborescência geradora de custo mínimo de D.

O algoritmo de Chu–Liu–Edmonds recebe um r-digrafo ponderado (D, w, r) e devolve uma r-arborescência geradora de custo mínimo de D.

Vamos primeiro fornecer uma visão geral do algoritmo. O algoritmo é recursivo. Inicialmente, ele faz uma escolha gulosa de um certo conjunto de arcos. Se esse conjunto forma uma arborescência, o algoritmo pára e devolve esse conjunto. Caso contrário, identifica um ciclo especial no digrafo e o contrai, produzindo um novo digrafo. Esse novo digrafo é então submetido recursivamente ao algoritmo, que devolve uma arborescência de custo mínimo. Por fim, utilizamos o ciclo contraído para construir uma arborescência de custo mínimo no digrafo original. Essa construção é detalhada a seguir.

Escolha gulosa

Suponha doravante que (D, w, 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 dentre aqueles que entram 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,

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

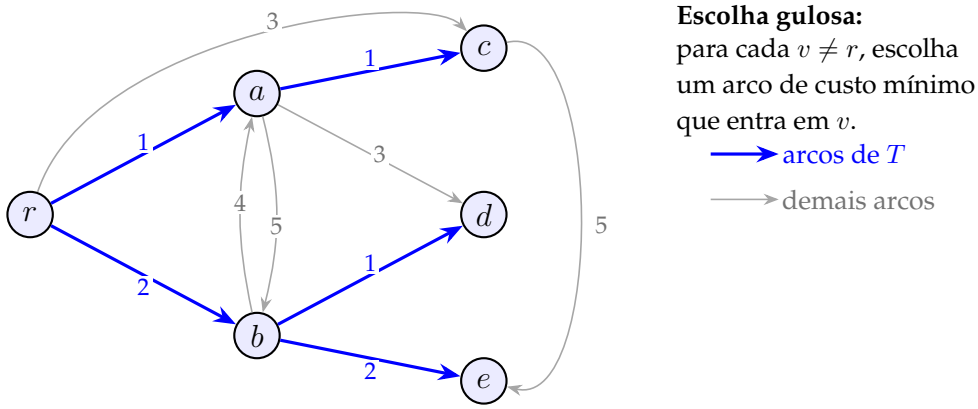


Figura 14 – A figura ilustra a escolha gulosa quando esta produz uma r-arborescência. Os arcos em azul são os escolhidos; os cinza são os demais arcos do digrafo.

Logo,

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

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

A Figura 15 ilustra que podemos não ter tanta sorte com T.

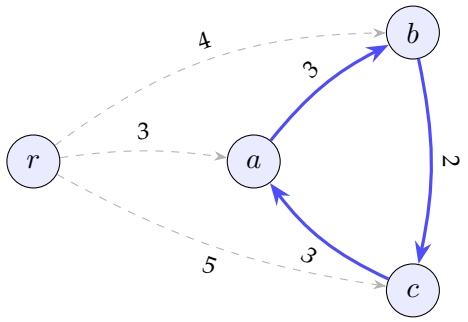


Figura 15 – Os arcos azuis são os da escolha gulosa.

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

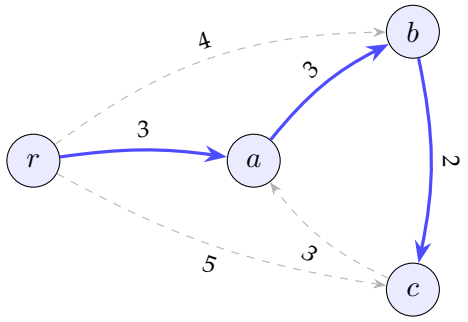


Figura 16 – Os arcos azuis são os da escolha gulosa.

2.2 Custos reduzidos

Vamos introduzir agora a noção de custo reduzido. Essa noção permite fazer uma tranformação nos custos que preserva a otimalidade.

Seja $q : V \setminus \{r\} \to \mathbb{R}$ uma função. Definimos o **custo q-reduzido** $w_q : A \to \mathbb{R}$ por[1]

$$w_q(uv) := w(uv) - q(v), \qquad uv \in A.$$

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

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

Proposição 2.1. *Para toda função $q : V \setminus \{r\} \to \mathbb{R}$, uma r-arborescência T é de custo mínimo em (D, w) se, e somente se, T é de custo mínimo em (D, w_q).*

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

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

Assim, para quaisquer r-arborescências T e F,

$$w(T) \leq w(F) \iff w_q(T) = w(T) - q(V \setminus \{r\}) \leq w(F) - q(V \setminus \{r\}) = w_q(F),$$

o que prova a proposição. □

O custo reduzido de interesse é o dado pela função λ definida a seguir. Para cada $v \in V \setminus \{r\}$, definimos

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

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. Consequentemente, para todo $v \in V \setminus \{r\}$,

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

[1] Recorde que nenhum arco entra em r; logo, a função w_q está bem definida.

isto é, precisamente os arcos de custo mínimo que entram em v passam a ter custo zero, e os demais ficam com custo positivo.

Definimos o subdigrafo gerador D_0 de D escolhendo, para cada $v \neq r$, exatamente um arco a_v que entra em v e satisfaz $w_\lambda(a_v) = 0$. Assim,

$$V(D_0) := V(D) \quad \text{e} \quad A(D_0) := \{\, a_v : v \in V(D) \setminus \{r\} \,\}.$$

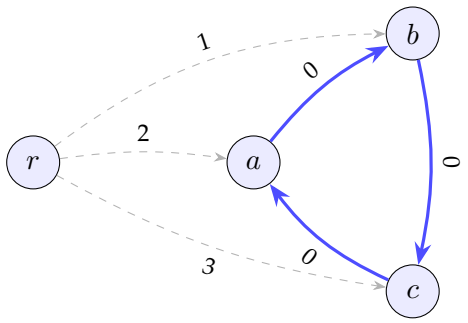


Figura 17 – Os arcos azuis são os arcos de D_0.

Como vimos, se D_0 é uma r-arborescência, então D_0 tem custo mínimo. Podemos então supor que D_0 não é uma r-arborescência. Vamos mostrar que, nesse caso, D_0 possui um ciclo.

Seja $v \neq r$ um vértice de V que *não* é alcançável a partir de r em D_0; um tal vértice existe uma vez que estamos admitindo que D_0 não possui uma r-aborescência. Observe que, por construção, para cada vértice s de D_0 existe exatamente um arco de D_0 que entra em s. Considere um caminho simples maximal[2] de D_0 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 tu, de D_0 que entra em u. Pela maximalidade de P, o vértice t é um dos vértices de P (caso contrário,[3] $t \cdot P$ é um caminho simples, o que contraria a escolha de P). Como P é um caminho simples que começa em u, o vértice t aparece em P *após* u; portanto, P contém um subcaminho S de u até t. Consequentemente, $S \cdot u$ é um ciclo de D_0. Isso prova que D_0 contém um ciclo.

Devemos agora mostrar o que fazer com um desses ciclos de D_0.

Contração de ciclos

A próxima operação do algoritmo é a contração de ciclos (veja a Seção 1.5). É conveniente identificar um ciclo com o conjunto de seus vértices. Suponha que o digrafo D_0 possua um ciclo, digamos C. Observe que $r \notin V(C)$. Recorde que, ao *contrair* o ciclo

[2] Maximal aqui tem o seguinte sentido. Para cada vértice u de D_0, as sequencias $P \cdot u$ e $u \cdot P$ não são caminhos simples.

[3] Para sequências α e β, escrevemos $\alpha \cdot \beta$ para denotar a concatenação de α e β. Para simplificar a notação, uma sequência de comprimento 1 é identificada com o seu único elemento.

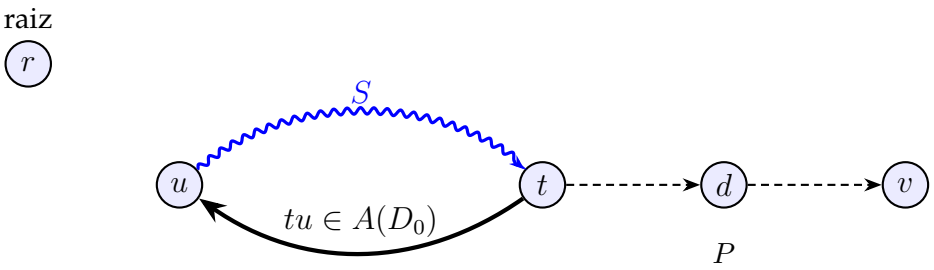


Figura 18 – O caminho simples maximal P inicia em u e termina em v. A porção S de P entre u e t é indicada pelo arco ondulado azul; o caminho $S \cdot u$ é um ciclo.

C em D e obter o digrafo $D/C \mapsto x_C$, identificamos todos os vértices de C em um único vértice, denotado por x_C e visto como um supervértice, e redirecionamos os arcos incidentes: arcos da forma uv, em que $v \in C$ (isto é, que entravam em C), passam a ser da forma ux_C em $D/C \mapsto x_C$, e arcos da forma vu, em que $v \in C$ (isto é, que saíam de C), passam a ser da forma $x_C u$ em $D/C \mapsto x_C$. Assim, em $D/C \mapsto x_C$, o ciclo C é tratado como um único vértice.

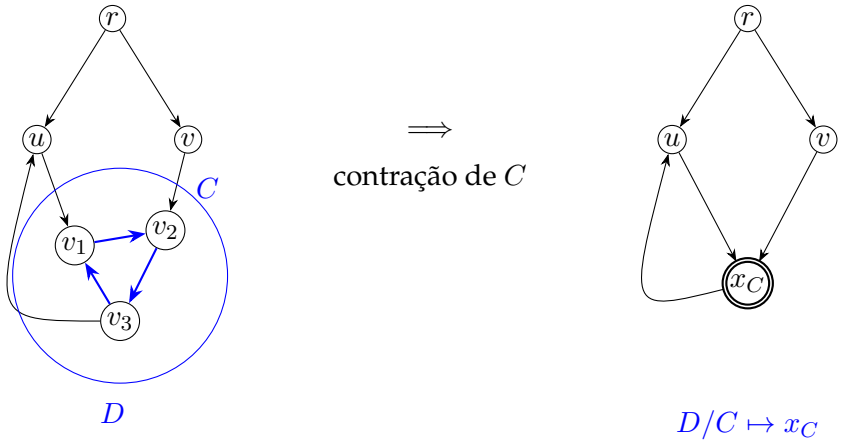
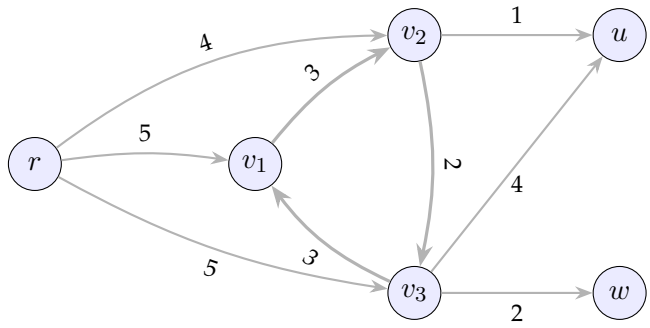


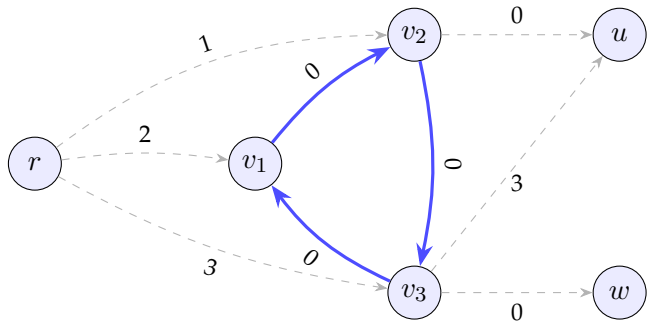
Figura 19 – Contração de um ciclo C em um vértice x_C.

Precisamos agora definir um novo r-digrafo ponderado. Ei-lo: (D', w', r), onde $D' := D/C \mapsto x_C$ e $w' := w_\lambda/C \mapsto x_C$. Note que r é um vértice de D' e que, além disso, D' possui uma r-arborescência.

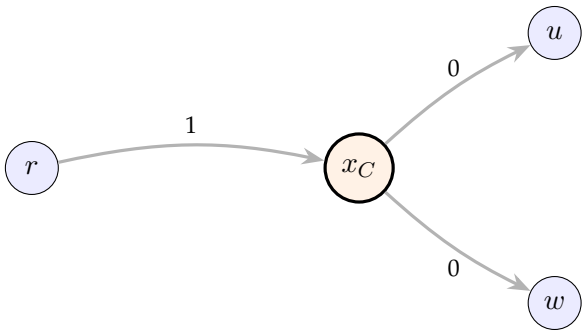
Agora vamos ilustrar um exemplo de como essa contração é feita e os custos são ajustados. Considere o digrafo a seguir.



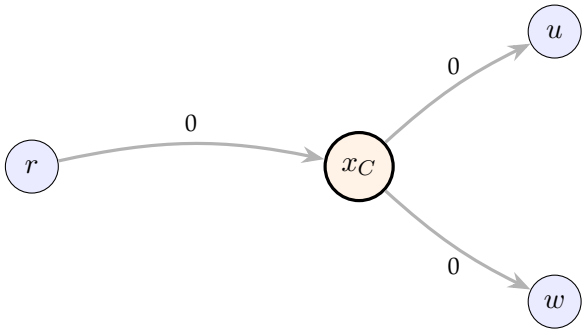
Após a redução dos custos, obtemos um ciclo $C := (v_1, v_2, v_3, v_1)$ cujos arcos têm custo reduzido igual a zero.



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



O digrafo contraído é submetido recursivamente ao algoritmo. Assim, o próximo passo consiste em reduzir os custos dos arcos dessa nova intância, obtendo assim o seguinte r-digrafo ponderado.

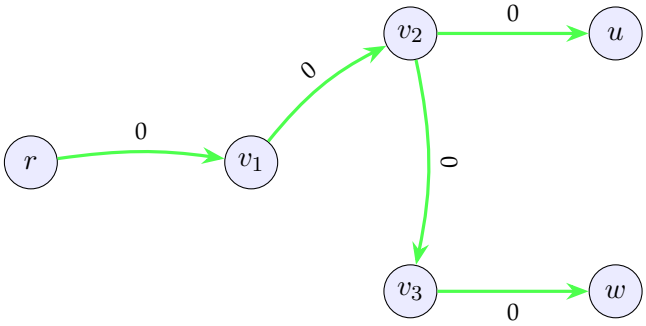


O subdigrafo gerador obtido a partir desse digrafo, usando apenas os arcos de custo reduzido igual a zero, possui uma r-arborescência. O algoritmo devolve essa r-arborescência, que agora deverá passar por um processo de expansão.

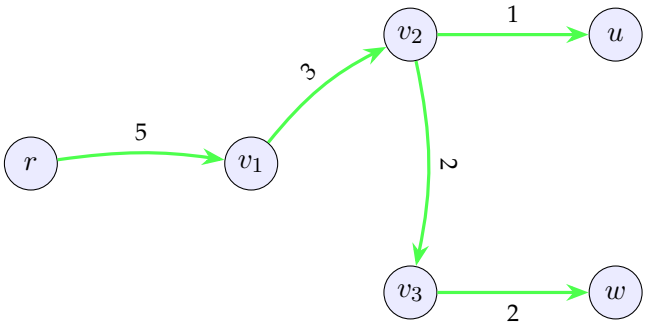
Reexpansão de arborescências

Após resolver o problema no digrafo contraído D', obtemos uma r-arborescência geradora T' de custo mínimo em (D', w'). Observe que tão somente um arco de T' entra em x_C. 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, o arco ux_C de T' corresponde a um arco uv de D, onde $v \in C$. Esse arco uv é incluído 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, garantindo que cada vértice de C tenha grau de entrada igual a 1 em T.

Retomemos o exemplo para ilustrar a expansão da r-arborescência geradora do digrafo contraído. Primeiro, adicionamos os vértices do ciclo C e, em seguida, incluímos os arcos apropriados para formar a r-arborescência geradora do digrafo original:



No r-digrafo ponderado original, temos a seguinte configuração.



Vamos agora verificar que a r-arborescência geradora T, obtida da construção descrita, é de custo mínimo em (D, w). Por hipótese, T' é uma r-arborescência geradora

de custo mínimo em (D', w'), isto é,

$$w'(T') \leq w'(F')$$

para toda r-arborescência geradora F' de D'. Pela construção de T, temos que T é uma r-arborescência geradora tal que $w_\lambda(T) = w'(T')$, uma vez que todo arco a de C satisfaz $w_\lambda(a) = 0$.

Suponha agora que F seja uma r-arborescência geradora de custo mínimo em (D, w_λ). Então $w_\lambda(F) \leq w_\lambda(T)$. Seja $F' := F/C \mapsto x_C$. Note que F' pode não ser uma r-arborescência de D'. No entanto, F' *contém* uma r-arborescência de D'; logo, existe $F'' \subseteq F'$ tal que F'' é uma r-arborescência de D'. Temos, então, $w'(F'') \leq w'(F')$ pois $w'(a) \geq 0$ para cada $a \in A(D')$. Por hipótese, $w'(T') \leq w'(F'')$ o que, combinado com $w_\lambda(T) = w'(T')$, permite inferir que

$$w_\lambda(T) \leq w'(F'') \leq w'(F') = w_\lambda(F).$$

Portanto, T é uma r-arborescência geradora de custo mínimo em (D, w_λ). Agora, pela Proposição 2.1, T é uma r-arborescência geradora de custo mínimo em (D, w), como queríamos. Isso completa a prova de que a r-arborescência devolvida pelo algoritmo é de custo mínimo em (D, w).

Convém agora sumarizar e exibir uma descrição em pseudocódigo do algoritmo de Chu–Liu/Edmonds. O mapeamento desse pseudocódigo para código `Python` será discutido na próxima seção.

Algoritmo 2.1: Chu–Liu/Edmonds

```
1 def chu-liu-edmonds(D, w, r):
2     λ := {(v, min{w(a) : a ∈ δ⁻(v)}) : v ∈ V \ {r}}
3     para cada v ∈ V \ {r}, seja a_v ∈ δ⁻(v) tal que w_λ(a_v) = 0
4     seja D_0 := (V, {a_v : v ∈ V \ {r}})
5     if D_0 é uma r-arborescência: return D_0
6     seja C um ciclo em D_0
7     T' := chu-liu-edmonds(D/C ↦ x_C, w_λ/C ↦ x_C, r)
8     T := expand(T')
9     return T
```

Complexidade

Não é difícil ver que as operações envolvidas nas linhas 2 a 6 e na linha 8 podem ser implementadas de forma a serem executadas em tempo $O(|A|)$. Como cada chamada

recursiva contrai ao menos um vértice, o número total de chamadas é limitado por $O(|V|)$. Portanto, o consumo de tempo do algoritmo está em $O(|V||A|)$.

Quanto ao consumo de memória, é possível realizar as operações de contração de modo que o uso total de memória adicional permaneça em $O(|V||A|)$. A implementação descrita a seguir, disponível em <https://github.com/lorenypsum/GraphVisualizer>, apresenta consumo de tempo e memória em $O(|V||A|)$.

2.3 Implementação em Python

Esta seção descreve a implementação do algoritmo de Chu–Liu/Edmonds em `Python`, estruturada para refletir com precisão as etapas discutidas anteriormente. Cada operação fundamental — redução dos custos, construção do subdigrafo gerador de arcos de custo reduzido iguais a zero, contração de ciclos e reexpansão — é implementada utilizando como suporte a biblioteca `networkx`.

Representação de digrafos e detecção de ciclos

A implementação utiliza a biblioteca NetworkX[4]. Digrafos ponderados são representados por instâncias da classe `DiGraph`. Internamente, essa classe usa dicionários do `Python` para armazenar vértices, arcos e atributos, o que garante operações eficientes na prática. Por exemplo, adicionar ou remover um arco tem consumo amortizado de tempo em $O(1)$; iterar sobre os sucessores de um vértice u tem consumo de tempo em $O(|\delta^+(u)|)$; e iterar sobre todos os arcos tem consumo de tempo em $O(m)$, em que m é o número de arcos do digrafo.

Métodos da API NetworkX

Elencamos a seguir alguns métodos da API `NetworkX` utilizados na implementação. Para uma instância `D` de `DiGraph`:

Consulta de estrutura

- `D.nodes()`: devolve um iterável sobre o conjunto dos vértices de `D`.
- `D.in_edges(v, data="w")`: devolve um iterável de triplas da forma `(u, v, w)`, em que `uv` é um arco de `D` com peso `w`.
- `D.out_edges(u, data="w")`: devolve um iterável de triplas da forma `(u, v, w)`, em que `uv` é um arco de `D` com peso `w`.

[4] Disponível em <https://networkx.org/>.

- `D[u][v]["w"]`: devolve o peso do arco `uv` de `D`, isto é, o valor armazenado no atributo `"w"` associado a esse arco.

Modificação de estrutura

- `D.add_edge(u, v, w=p)`: adiciona o arco `uv` a `D` com peso `p` armazenado no atributo `"w"`. Os vértices `u` e/ou `v` são criados automaticamente se ainda não existirem em `D`.
- `D.remove_edges_from(edges)`: recebe um iterável `edges` de arcos e remove de `D` cada arco `(u, v)` em `edges`.
- `D.remove_nodes_from(nodes)`: recebe um iterável `nodes` de vértices e remove de `D` cada vértice `v` em `nodes` (bem como todos os arcos incidentes em `v`).

Remoção de arcos que entram na raiz

Recorde que o algoritmo de Chu–Liu–Edmonds recebe um r-digrafo ponderado e que, por definição, em um r-digrafo ponderado nenhum arco entra em r. Escrevemos esta função como uma etapa de pré-processamento justamente para garantir que a raiz r não possua arcos de entrada antes de iniciar o algoritmo principal.

Em detalhes, essa função recebe como entrada uma instância `D` de `DiGraph` e um vértice `r` de `D`. A função modifica `D` removendo todos os arcos que entram em `r` e tem consumo amortizado de tempo em $O(k)$, em que k é o número de arcos que entram em `r`. Destacamos que é necessário armazenar, em uma lista, todos esses arcos usando o método `in_edges` (linha 2), pois esse método devolve um iterador que é invalidado assim que alguma operação modifica a estrutura de dados que o produziu.

Remoção de arcos que entram na raiz

Entrada: `D: Digraph` e `r`.
Pré-condição: `r` vértice de `D`.
Modifica: `D`.
Pós-condição: `D` não possui nenhum arco que entra em `r`.

```
1 def remove_edges_to(D: nx.DiGraph, r: int):
2     in_edges = list(D.in_edges(r))
3     D.remove_edges_from(in_edges)
```

2.3.1 Redução de custos

A função `reduce_weights` tem como propósito realizar a redução de custos por vértice. Em outras palavras, é nessa função que, para um vértice v, calculamos $\lambda(v)$ e obtemos os custos λ-reduzidos dos arcos que entram em v. A função recebe `D: DiGraph` e um vértice `v`. A variável `in_edges` é um iterável de triplas da forma `(u, v, w)`, em que `(u, v)` é um arco de `D` e `w` é o seu peso, obtidas por meio do método `D.in_edges(node, data="w")`. A variável `yv` é o peso mínimo entre esses arcos. Em seguida, os pesos dos arcos que entram em `v` são decrementados de `yv`. O consumo de tempo está em $O(k)$, em que k é o número de arcos que entram em `node`.

Redução de custos por vértice (normalização)

Entrada: `D: DiGraph`, `v: int`.
Pré-condição: `v` é vértice de `D` e possui ao menos um arco de entrada.
Modifica: `D`.
Pós-condição: `D[u][v]["w"]` é o custo λ-reduzido do arco `(u, v)` para cada `u` que é predecessor de `v`.

```
1 def reduce_weights(D: nx.DiGraph, v: int):
2     in_edges = D.in_edges(v, data=True)
3     yv = min((data["w"] for _, _, data in in_edges))
4     for u, _, _ in in_edges:
5         D[u][v]["w"] -= yv
```

A Figura 20 ilustra o funcionamento da redução:

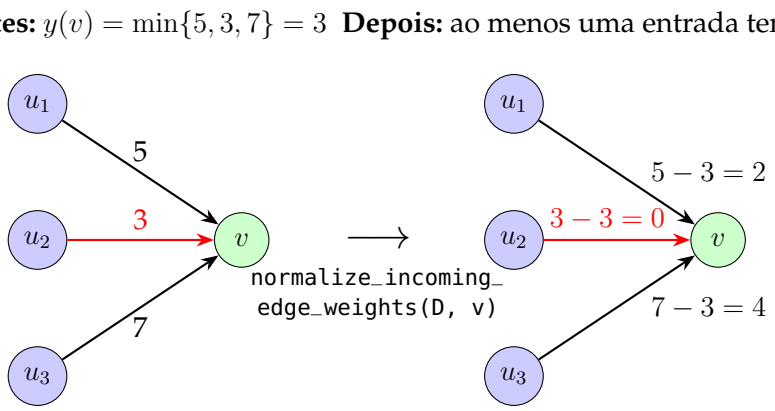


Figura 20 – À esquerda, vértice v com três arcos de entrada (pesos 5, 3 e 7). À direita, após aplicar `reduce_weights(D, v)`: o menor peso 3 é subtraído de todas as entradas, resultando em custos reduzidos 2, 0 e 4. O arco (u_2, v) (em vermelho) tem custo zero e será selecionado para D_0.

2.3.2 Construção de D_0

Vamos mostrar agora como construir o subdigrafo D_0 de D. Lembre-se de que D_0 é o subdigrafo gerador de D em que, para cada vértice $v \neq r$, selecionamos um arco que entra em v com peso zero.

A função é bastante simples. Ela recebe um `D: DiGraph` e um vértice `r` de `D` tais que, em cada vértice `v` diferente de `r`, entra ao menos um arco de peso zero. A função devolve um subdigrafo gerador `D0: DiGraph`, construído ao se iterar sobre o conjunto dos vértices `v` de `D` distintos de `r` e selecionar exatamente um arco de peso zero que entra em `v`.

Construção de D_0

Entrada: `D: DiGraph`, `r: int`.
Pré-condição: `r` é vértice de `D` e para cada vértice distinto de `r` existe um arco de peso zero (atributo `"w"`) que nele entra.
Saída: `D0: Digraph` subdigrafo gerador de `D` tal que em cada vértice diferente de `r` entra exatamente um arco de custo reduzido igual a zero.

```
1 def get_Dzero(D: nx.DiGraph, r: int):
2     D_zero = nx.DiGraph()
3     for v in D.nodes():
4         if v != r:
5             in_edges = D.in_edges(v, data=True)
6             u = next((u for u, _, data in in_edges
7                         if data[”w”] == 0))
8             D_zero.add_edge(u, v)
9     return D_zero
```

As funções de redução de custo e construção de D_0 juntas implementam os passos das linhas 2, 3, e 4 do Algoritmo de Chu-Liu/Edmonds.

2.3.3 Detecção de ciclo:

O próximo passo consiste em mostrar como decidir se o digrafo D_0 é uma r-arborescência e, caso isso não ocorra, como encontrar um ciclo para futura contração. A decisão sobre se D_0 é uma arborescência é delegada a uma função de biblioteca, chamada `is_arborescence`. Note que, em virtude da forma como D_0 é construído, se D_0 é uma arborescência, então D_0 é necessariamente uma r-arborescência. No que segue, vamos assumir que D_0 não é uma arborescência.

A função recebe `D_zero: DiGraph` e supõe que (i) `D_zero` não é uma arborescência;

(ii) existe exatamente um vértice de `D_zero` no qual não entra nenhum arco; e (iii) em cada um dos demais vértices, entra ao menos um arco. A função devolve um subdigrafo `C: DiGraph` de `D_zero` que é um ciclo.

Uma função de biblioteca, `find_cycle`, é usada para encontrar um ciclo em `D_zero`. Os arcos desse ciclo — resultado da chamada

```
find_cycle(D_zero, orientation="original")
```

— determinam o subconjunto de vértices do ciclo. A função devolve o subdigrafo de `D_zero` induzido por esse subconjunto vértices.

Detecção de ciclo dirigido em D_0

Entrada: `D_zero: DiGraph`.
Pré-condição: `D_zero` não é uma arborescência, e existe exatamente um vértice de `D_zero` no qual não entra nenhum arco e nos demais vértices entra ao menos um arco.
Saída: `C: Digraph` subdigrafo de `D_zero` que é um ciclo.

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

2.3.4 Contração de um ciclo

Vamos agora mostrar como implementar a contração de um ciclo C de D_0. A função `contract_cycle` recebe um digrafo `D`, um[5] subdigrafo `C` de `D` e um nome `label` para o supervértice do digrafo contraído. A função modifica `D` de tal forma que, após a chamada, `D` representa o digrafo $\texttt{D/C} \mapsto \texttt{label}$ e devolve dois dicionários

```
in_to_cycle, out_from_cycle : dict[int, tuple[int, float]],
```

cuja formação é explicada a seguir.

Considere um vértice `u` de `D` que está fora de `C` e que possui ao menos um sucessor em `C`. Dizemos que um arco `uv` de `D` é **essencial de `u` para `C`** se `v` é vértice de `C` e o custo de `uv` é mínimo entre os custos dos arcos que saem de `u` e entram em `C`. De forma similar, considere um vértice `v` de `D` que está fora de `C` e que possui ao menos um antecessor em `C`. Dizemos que um arco `uv` de `D` é **essencial de `C` para `v`** se `u` é vértice de `C` e o custo de `uv` é mínimo entre os custos dos arcos que saem de `C` e entram em `v`.

[5] Isso é irrelevante: `C` poderia ser qualquer subdigrafo não vazio de `D`.

A função constrói os dicionários com os arcos essenciais e seus pesos. Assim, para cada `u` que é antecessor de `C`, `in_to_cycle[u] = (v, w)` se, e somente se, `uv` é um arco essencial de `u` para `C`, de custo `w`. De forma similar, para cada `v` que é sucessor de `C`, `out_from_cycle[v] = (u, w)` se, e somente se, `uv` é um arco essencial de `C` para `v`, de custo `w`.

É fácil ver que o consumo de tempo da função está em $O(m)$, em que m é o número de arcos de `D`.

Contração de ciclo

Entrada: `D: DiGraph`, `C: DiGraph`, `label: int`.
Pré-condição: `C` subdigrafo de `D` e `label` não é um vértice de `D`.
Modifica: `D`.
Pós-condição: `D` é o digrafo $\texttt{D}/\texttt{C} \mapsto \texttt{label}$.
Saída: `in_to_cycle, out_from_cycle: dict[int, tuple[int, float]]`.
Para cada `u` que é antecessor de `C`, `in_to_cycle[u] = (v, w)` se, e somente se, `uv` é um arco essencial de `u` para `C` de custo `w`.
Para cada `u` que é sucessor de `C`, `in_to_cycle[u] = (v, w)` se, e somente se, `vu` é um arco é um arco essencial de `C` para `u` de custo `w`.

```
1  def contract_cycle(D: nx.DiGraph, C: nx.DiGraph, label: int):
2      cycle_nodes: set[int] = set(C.nodes())
3      in_to_cycle: dict[int, tuple[int, float]] = {}
4      for u in D.nodes:
5          if u not in cycle_nodes:
6              min_weight_edge_to_cycle = min(
7                  ((v, data["w"])
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[int, tuple[int, float]] = {}
17     for v in D.nodes:
18         if v not in cycle_nodes:
19             min_weight_edge_from_cycle = min(
20                 ((u, data["w"])
21                     for u, _, data in D.in_edges(v, data=True)
```

```
22                      if u 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      D.remove_nodes_from(cycle_nodes)
30      return in_to_cycle, out_from_cycle
```

2.3.5 Procedimento principal

Vamos agora apresentar a função principal, que orquestra todas as funções auxiliares descritas anteriormente e completa a implementação do algoritmo de Chu–Liu–Edmonds. A função recebe um digrafo ponderado `D`, um vértice raiz `r` e um inteiro `label` que satisfazem:

- os vértices de `D` são inteiros no conjunto $\{0, 1, \ldots, n-1\}$, para algum $n \geq 1$;
- `r` é um vértice de `D`;
- `D` possui ao menos uma r-arborescência;
- nenhum arco de `D` entra em `r`; e
- `label` é um inteiro maior ou igual a n.

A função devolve um digrafo ponderado que é uma r-arborescência geradora de custo mínimo de `D`.

A seguir, comentamos brevemente o papel de cada bloco de instruções do código.

Na linha 2, `D_copy` é uma cópia de `D`. As linhas 3 a 5 são responsáveis por calcular o custo reduzido de cada arco do digrafo `D_copy` (as modificações dos custos são feitas em `D_copy`). A linha 6 determina o digrafo `D_zero`, que contém exatamente um arco de custo reduzido igual a zero entrando em cada vértice distinto de `r`. A linha 7 determina se `D_zero` é uma arborescência. Se esse for o caso, a função restaura os pesos originais nos arcos de `D_zero` e devolve `D_zero`.

Suponha que `D_zero` não seja uma arborescência. A linha 11 determina um ciclo em `D_zero`, armazenando-o em `C`. A linha 12 contrai o ciclo `C` em `D_copy`; o digrafo `D_copy` é modificado de tal forma que corresponda ao digrafo contraído. A linha 13 determina em `F_prime` uma r-arborescência geradora de custo mínimo do digrafo

`D_copy`. As demais linhas são responsáveis pelo processo de expansão de `F_prime`: elas modificam `F_prime` de tal forma que `F_prime` se torne uma r-arborescência geradora de custo mínimo de `D`.

O código completo da função principal é apresentado a seguir:

Procedimento principal (recursivo)

Entrada: `D: DiGraph`, `r: int`, `label: int`.
Pré-condição: `D` é um digrafo ponderado; os vértices de `D` são inteiros no conjunto $\{0, 1, \ldots, n-1\}$ para algum $n \geq 0$; `r` é um vértice de `D`; `D` possui ao menos uma `r`-arborescência; nenhum arco de `D` entra em `r`; e `label` é maior ou igual a n.
Saída: `T: DiGraph` é uma `r`-arborescência de custo mínimo de `D`.

```
1  def chuliu_edmonds(D: nx.DiGraph, r: int, label: int):
2      D_copy = cast(nx.DiGraph, D.copy())
3      for v in D_copy.nodes:
4          if v != r:
5              reduce_weights(D_copy, v)
6      D_zero = get_Dzero(D_copy, r)
7      if nx.is_arborescence(D_zero):
8          for u, v in D_zero.edges:
9              D_zero[u][v]["w"] = D[u][v]["w"]
10         return D_zero
11     C = find_cycle(D_zero)
12     in_to_cycle, out_from_cycle = contract_cycle(D_copy, C, label)
13     F_prime = chuliu_edmonds(D_copy, r, label + 1)
14     in_edge = next(iter(F_prime.in_edges(label, data=True)))
15     u, _, _ = cast(tuple, in_edge)
16     v, _ = in_to_cycle[u]
17     F_prime.add_edge(u, v)
18     for u_c, v_c in C.edges:
19         if v != v_c: F_prime.add_edge(u_c, v_c)
20     for _, z, _ in list(F_prime.out_edges(label, data=True)):
21         u_cycle, _ = out_from_cycle[z]
22         F_prime.add_edge(u_cycle, z)
23     F_prime.remove_node(label)
24     for u, v in F_prime.edges:
25         F_prime[u][v]["w"] = D[u][v]["w"]
26     return F_prime
```

3 Algoritmo de András Frank

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

3.1 O algoritmo

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

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

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

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

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

fonte no digrafo atualizado também chamado de digrafo de condensação.

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

Vamos desenvolver essas ideias utilizando um exemplo: considere o dígrafo D da Figura 21 com seis vértices $\{r, a, b, c, d, e\}$ e custos nos arcos para ilustrar o comportamento completo do algoritmo.

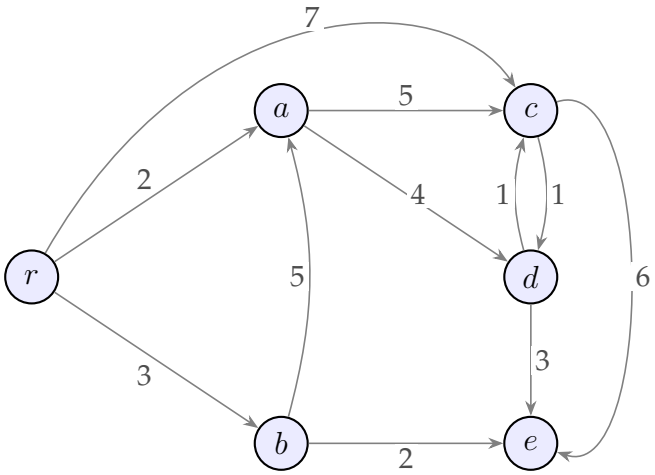


Figura 21 – Dígrafo D com custos originais. Este exemplo ilustrará todas as etapas do algoritmo de András Frank, incluindo formação de ciclos e contração.

Fase 1

Inicialmente, nenhum arco tem custo reduzido zero, cada vértice não-raiz $v \neq r$ forma seu próprio subconjunto minimal $\{v\}$. No dígrafo da Figura 21, os subconjuntos minimais iniciais são portanto $\{a\}$, $\{b\}$, $\{c\}$, $\{d\}$ e $\{e\}$. Para cada um desses conjuntos unitários, encontramos $\delta(\{v\})$, o menor custo entre todos os arcos entrando em v, e subtraímos esse valor de todos esses arcos.

Por exemplo, consideremos o vértice a. Os arcos entrando em a são (r, a) com custo 2 e (b, a) com custo 5. Calculamos $\delta(\{a\}) = \min\{2, 5\} = 2$ e subtraímos este valor de ambos os arcos. Assim, (r, a) passa a ter custo 0 e (b, a) passa a ter custo 3. A Figura 22 ilustra essa operação de redução aplicada ao vértice a.

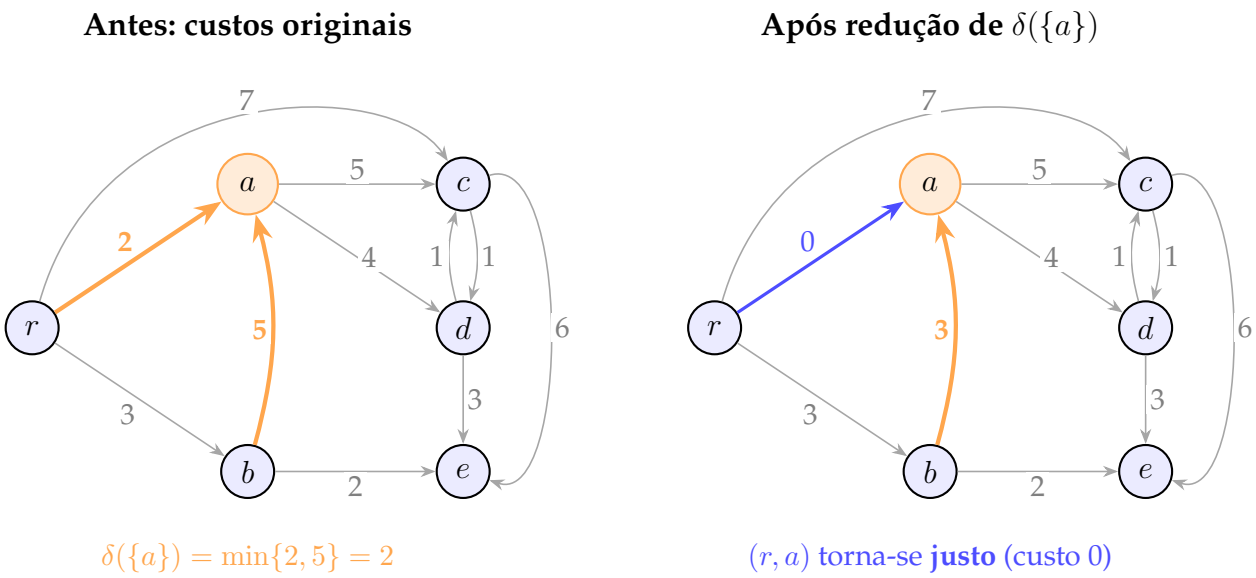


Figura 22 – Exemplo de redução de custo para o vértice a no dígrafo completo. À esquerda, os arcos entrando em a estão destacados em laranja com custos originais 2 e 5. Calculamos $\delta(\{a\}) = 2$ e subtraímos esse valor de ambos os arcos. À direita, após a redução: (r, a) tem custo zero (arco justo, em azul) e (b, a) tem custo $5 - 2 = 3$ (em laranja). Os demais arcos permanecem inalterados.

Essa operação vai sendo repetida para todos os conjuntos minimais iniciais. A Figura 23 mostra o resultado dessa primeira iteração no exemplo da Figura 21, onde aplicamos a redução apenas para os vértices isolados.

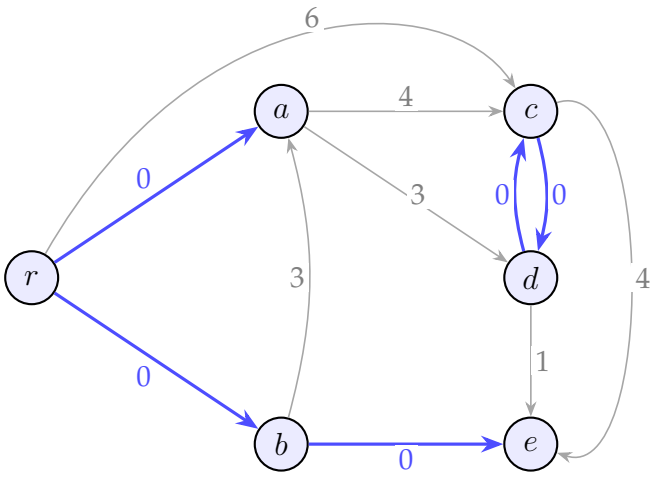


Figura 23 – Dígrafo após a primeira iteração. Os arcos justos (custo 0) são: (r, a), (r, b), (b, e), (c, d) e (d, c). Todos os vértices não-raiz possuem arcos justos entrando: a tem (r, a), b tem (r, b), e tem (b, e), e o conjunto $\{c, d\}$ tem (a, c) (além do ciclo interno). Os arcos (c, d) e (d, c) formam um ciclo justo.

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

zero entrando neles.

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

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

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

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

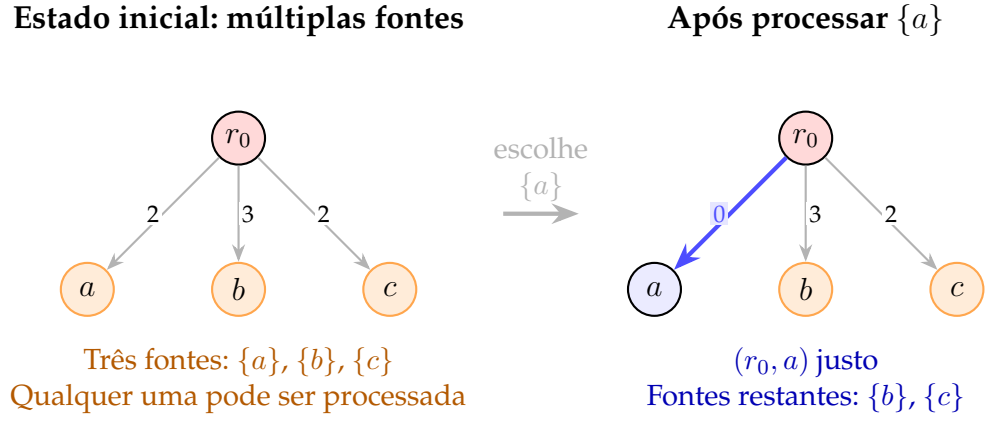


Figura 24 – Exemplo de múltiplas fontes disponíveis para processamento. À esquerda, o estado inicial possui três componentes $\{a\}$, $\{b\}$, $\{c\}$ (em laranja) que são fontes no grafo de condensação. Qualquer uma pode ser escolhida. À direita, após escolher e processar $\{a\}$ (elevando seu potencial por $\Delta(\{a\}) = 2$), o arco (r_0, a) torna-se justo (em azul). Agora $\{a\}$ deixa de ser fonte e as fontes restantes são $\{b\}$ e $\{c\}$. A ordem de processamento não afeta a arborescência ótima final.

No exemplo da Figura 23, após a primeira iteração de redução, temos os vértices $\{a\}$, $\{b\}$, $\{e\}$ que são componentes triviais (cada um forma sua própria CFC). Porém,

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

Nesse exemplo da Figura 23, os arcos justos não formam uma r-arborescência, pois nem todos os vértices são alcançáveis a partir de r pelos arcos justos e o conjunto $\{c, d\}$ forma um ciclo com dois arcos justos entre c e d. Temos então as fontes $\{r, a\}$, $\{b\}$, $\{e\}$ e o conjunto $\{c, d\}$. Como não existem arcos justos entrando em $\{c, d\}$, esse conjunto é um subconjunto minimal.

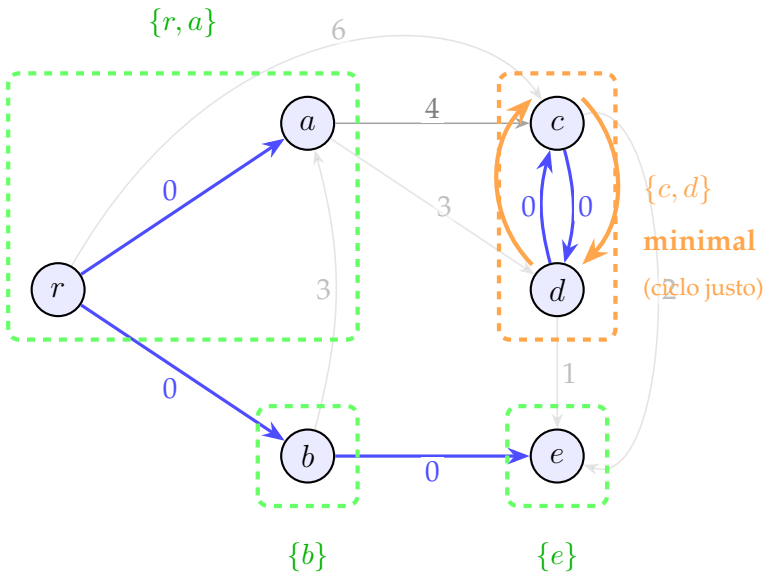


Figura 25 – Identificação de componentes fortemente conexas nos arcos justos após a primeira iteração. As componentes triviais $\{r, a\}$, $\{b\}$ e $\{e\}$ estão em verde. A componente não-trivial $\{c, d\}$ (em laranja) forma um ciclo justo com os arcos (c, d) e (d, c), e é identificada como **minimal** para a próxima iteração. Os arcos justos internos ao ciclo estão destacados, indicando que $\{c, d\}$ deve ser tratado como uma unidade no processo de contração.

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

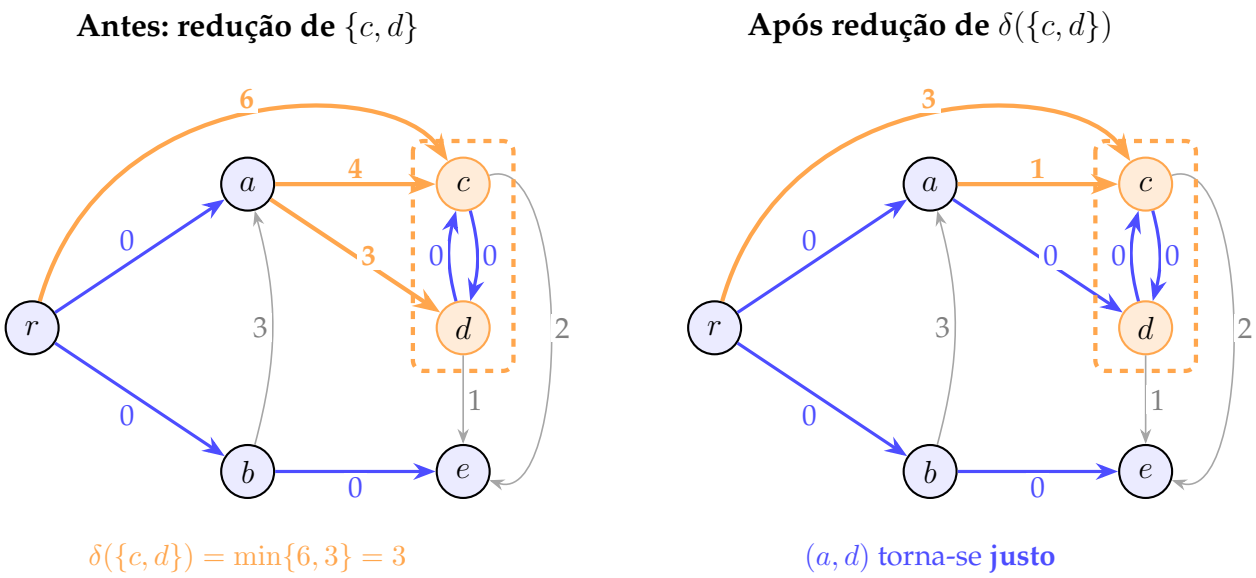


Figura 26 – Redução de custos para o subconjunto minimal $\{c, d\}$. À esquerda, antes da redução: os arcos entrando em $\{c, d\}$ vindos de fora são (r, c) com custo 6, (a, d) com custo 3 e (a, c) com custo 4, destacados em laranja. Calculamos $\delta(\{c, d\}) = 3$ e subtraímos esse valor. À direita, após a redução: (a, d) torna-se justo (custo 0), (r, c) tem custo reduzido para 3 e (a, c) tem custo reduzido para 1. O conjunto $\{c, d\}$ está destacado em laranja para enfatizar que é tratado como uma unidade.

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

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

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

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

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

da contração.

A Figura 27 ilustra essa contração.

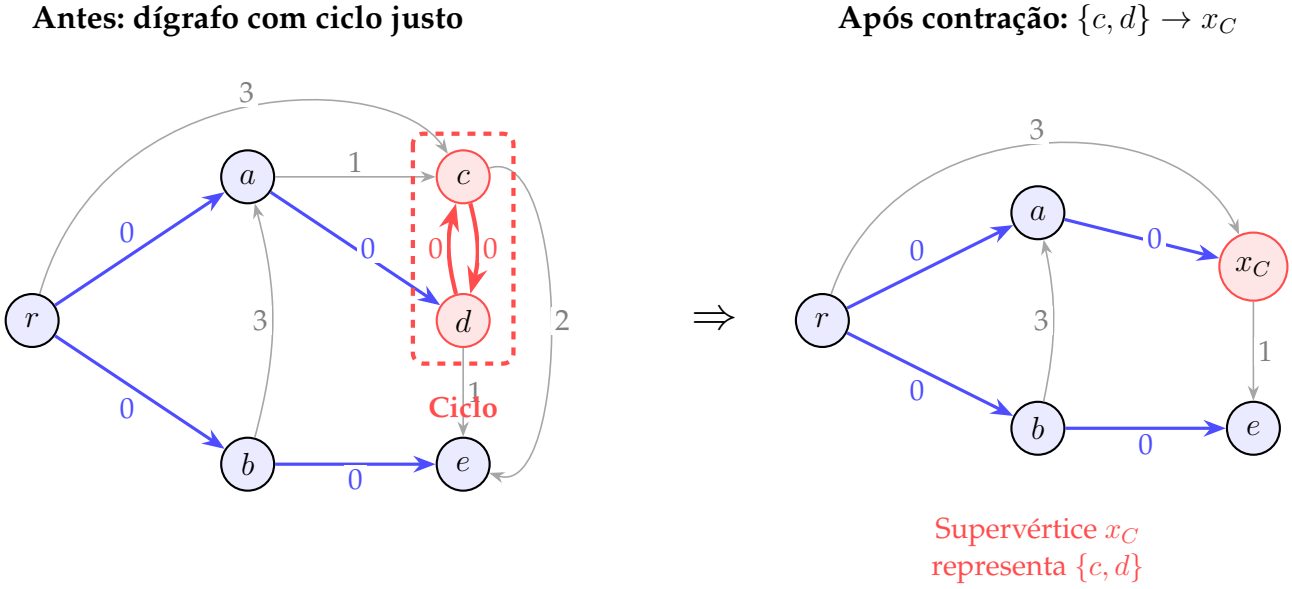


Figura 27 – Contração do ciclo justo $\{c, d\}$. À esquerda, o dígrafo após as reduções de custo mostra o ciclo justo formado pelos arcos (c, d) e (d, c) (em vermelho). À direita, o dígrafo contraído onde os vértices c e d são substituídos pelo supervértice x_C. Os arcos que entravam ou saíam do ciclo são redirecionados para x_C. Note que os arcos justos agora formam uma r-arborescência no dígrafo contraído.

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

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

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

Quando A_0 contém um ciclo, a operação de contração (ilustrada na Figura 27) é necessária para eliminar essa ambiguidade. Ao contrair o ciclo em um supervértice e resolver recursivamente, o algoritmo efetivamente escolhe quais arcos do ciclo fazer parte da solução final. Durante a expansão, apenas $|C| - 1$ arcos do ciclo contraído C são incluídos na arborescência (onde $|C|$ é o número de vértices no ciclo), descartando exatamente um arco — aquele que entra no vértice por onde a arborescência alcança o

ciclo.

Outras sujeiras podem ocorrer quando múltiplos arcos justos entram em um mesmo vértice, como no caso de múltiplas fontes sendo processadas em ordens diferentes. A Figura 28 ilustra esse fenômeno no exemplo anterior: após processar os conjuntos $\{a\}$, $\{b\}$ e $\{e\}$, tanto o arco (r, a) quanto o arco (b, a) podem se tornar justos simultaneamente se os custos forem adequados. Nesse caso, A_0 conteria ambos os arcos entrando em a, mas a arborescência final incluirá apenas um deles — o algoritmo escolhe o arco cujo vértice origem já está conectado à raiz na arborescência parcial.

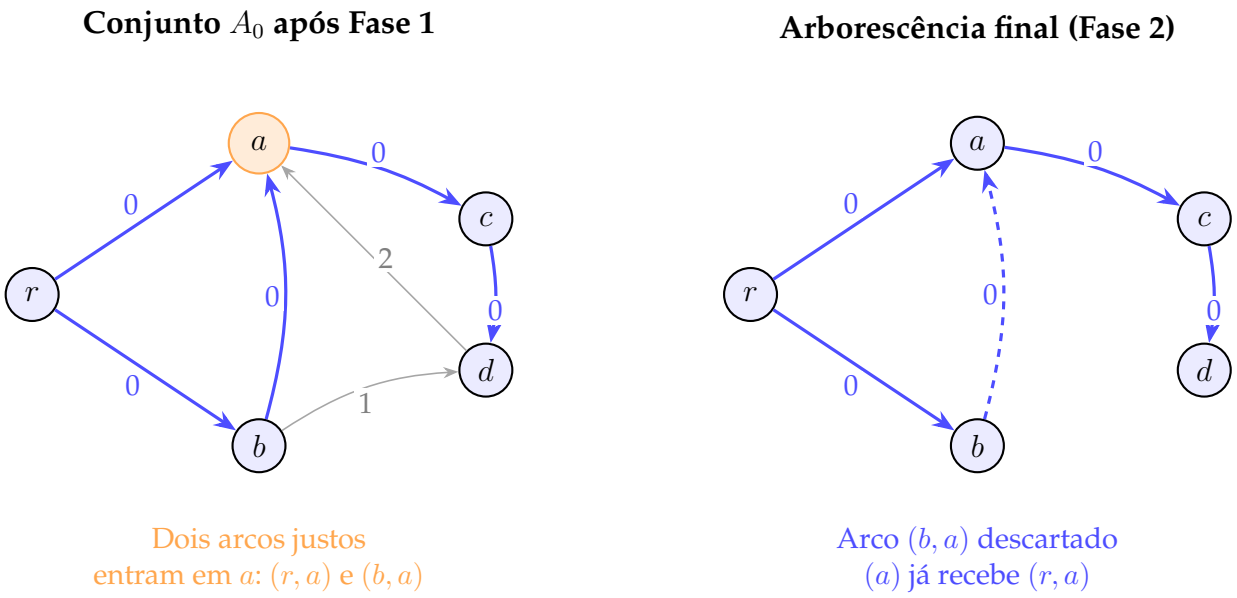


Figura 28 – Exemplo de múltiplos arcos justos entrando em um mesmo vértice. À esquerda, após a Fase 1, o conjunto A_0 contém tanto (r, a) quanto (b, a) como arcos justos (ambos destacados em azul sólido). À direita, durante a construção incremental da arborescência na Fase 2, apenas um arco é escolhido: (r, a) é incluído porque r já está na arborescência parcial, enquanto (b, a) (mostrado tracejado) é descartado, pois a já possui um arco de entrada. A Fase 2 garante que cada vértice não-raiz receba exatamente um arco entrando na arborescência final.

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

Fase 2

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

A Figura 29 ilustra esse processo no exemplo. No dígrafo contraído, a arborescência ótima inclui o arco (a, x_C) entrando no supervértice x_C. Na expansão, esse arco corresponde a (a, d) no dígrafo original. A arborescência final então inclui (a, d) e todos os arcos justos do ciclo exceto o arco que entra em d, ou seja, exclui (c, d) e mantém (d, c). Assim, o vértice c é alcançado através de d.

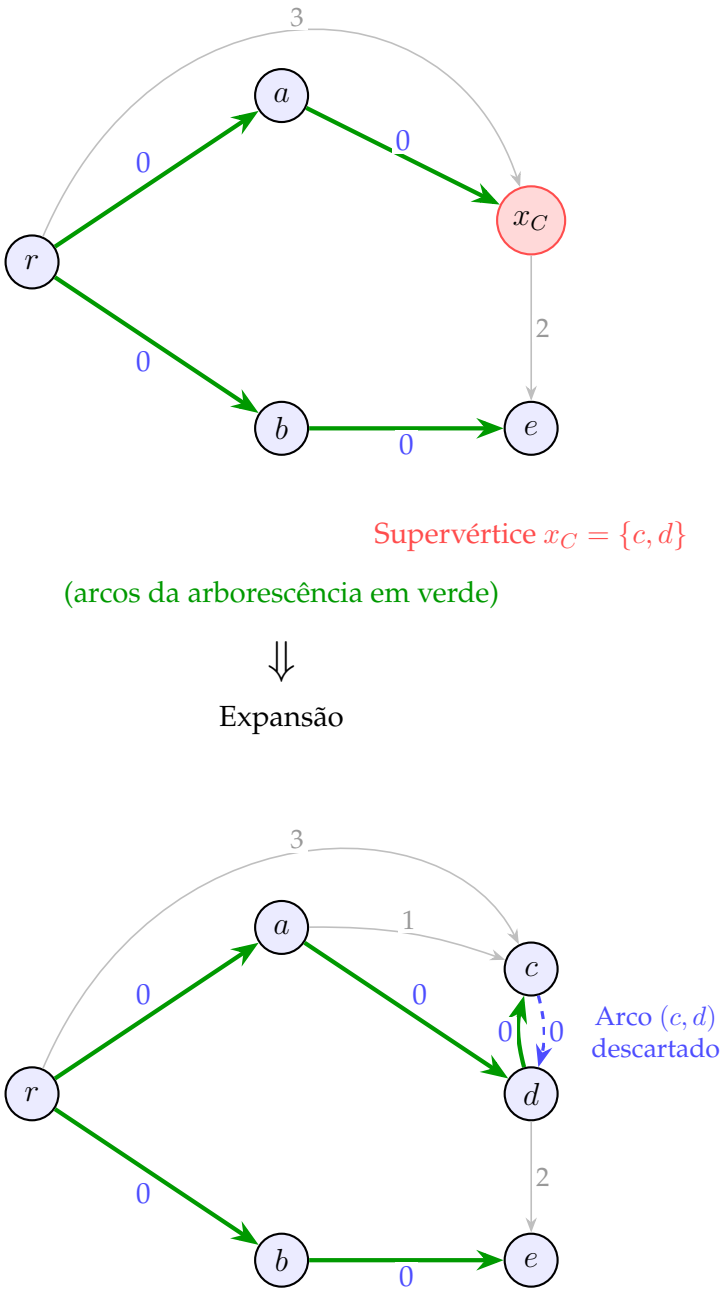


Figura 29 – Fase 2: Expansão do supervértice e construção da arborescência final. No topo, a arborescência ótima no dígrafo contraído (arcos verdes grossos) inclui o arco (a, x_C) entrando no supervértice $x_C = \{c, d\}$. Abaixo, após a expansão no dígrafo original, o arco (a, x_C) é substituído por (a, d), que corresponde ao arco justo que entra no ciclo. O ciclo é "aberto"incluindo apenas $|C| - 1 = 1$ arco interno: (d, c) é incluído na arborescência (verde), enquanto (c, d) é descartado (azul tracejado), pois d já recebe o arco (a, d). O resultado é uma r-arborescência com exatamente $n - 1 = 5$ arcos, todos de custo reduzido zero, portanto ótima.

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

3.2 Descrição do algoritmo

Apresentamos agora uma descrição formal do algoritmo de András Frank.

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

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

Algoritmo 3.1: András Frank

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

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

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

Fase 2: Construção da arborescência

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

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

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

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

3.2.1 Corretude

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

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

   Isso porque, para qualquer r-arborescência F,

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

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

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

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

3.2.2 Complexidade

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

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

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

3.3 Implementação em Python

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

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

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

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

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

3.3.1 Identificação de arcos entrando em conjunto X

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

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

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

Identificação de arcos entrando em conjunto X

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

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

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

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

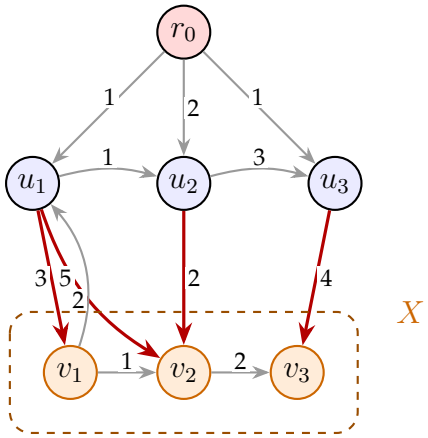


Figura 30 – Ilustração da função `get_arcs_entering_X` em D_{32}. A raiz r_0 (em vermelho claro) conecta-se aos vértices u_1, u_2, u_3. Os vértices em laranja pertencem ao conjunto $X = \{v_1, v_2, v_3\}$. A função identifica apenas os arcos em vermelho: aqueles que saem de vértices fora de X e entram em vértices dentro de X. Arcos da raiz, arcos internos a X, externos a X, ou saindo de X não são retornados.

3.3.2 Cálculo do peso mínimo de corte

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

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

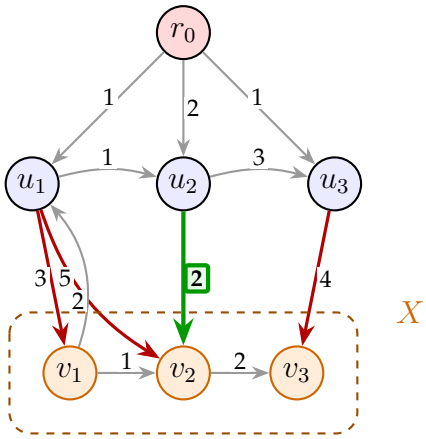
A função devolve o peso mínimo encontrado entre todos os arcos da lista. A complexidade é $O(k)$, onde k é o número de arcos na lista, pois examina cada arco uma vez para encontrar o mínimo.

Cálculo do peso mínimo de corte

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

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

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



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

Figura 31 – Ilustração da função `get_minimum_weight_cut` em D_{32}. Considerando os arcos em vermelho que entram em X (identificados pela função anterior), esta função calcula o peso mínimo entre eles. O arco em verde possui o menor peso (2), correspondendo ao valor $\Delta(X) = 2$.

3.3.3 Atualização de pesos em X

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

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

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

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

Atualização de pesos em X

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

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

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

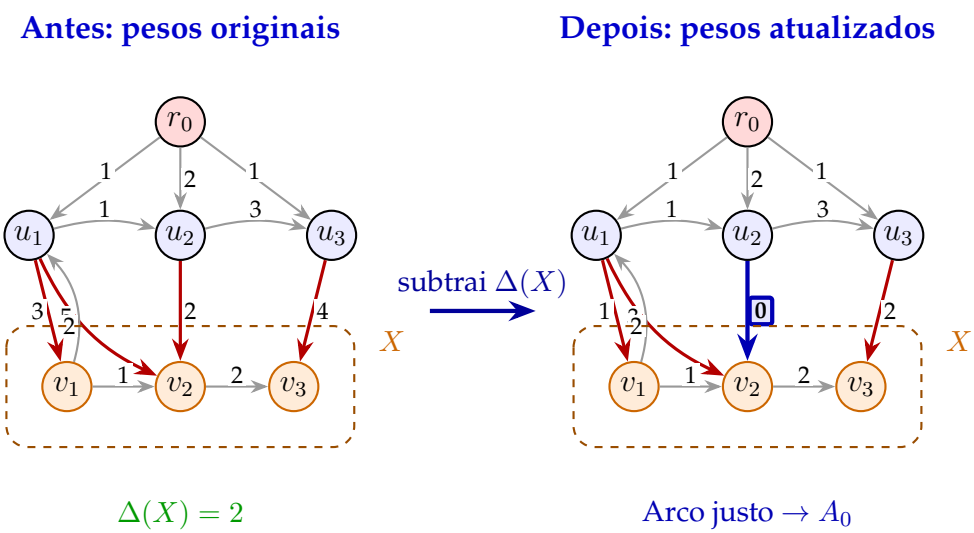


Figura 32 – Ilustração da função `update_weights_in_X` em D_{32}. À esquerda, o dígrafo antes da atualização, com os arcos em vermelho entrando em X e $\Delta(X) = 2$. À direita, após subtrair $\Delta(X)$ de cada arco entrando em X: o peso (u_1, v_1) reduz de 3 para 1, (u_2, v_2) de 2 para **0** (torna-se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

3.3.4 Verificação de arborescência

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

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

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

Verificação de arborescência

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

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

3.3.5 Fase 1: Elevação de potenciais e construção de A_0

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

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

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

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

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

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

Fase 1: Elevação de potenciais e construção de A_0

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

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

As funções auxiliares implementadas nesta seção correspondem diretamente aos

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

Correspondência entre Teoria e Implementação — Fase 1

Passo 1 — Inicialização:

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

Passo 2 — Iteração de elevação de potenciais:

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

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

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

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

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

Passo 3 — Construção de A_0:

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

3.3.6 Fase 2: Construção da arborescência

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

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

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

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

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

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

Fase 2: Construção da arborescência

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

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

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

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

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

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

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

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

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

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

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

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

Correspondência entre Teoria e Implementação — Fase 2

Passo 4 — Construção da arborescência (caso acíclico):

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

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

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

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

3.3.7 Verificação de otimalidade dual

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

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

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

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

Verificação de otimalidade dual

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

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

3.3.8 O algoritmo completo de András Frank

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

Verificação de otimalidade dual

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

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

Fase 2 — Construção da arborescência:

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

Verificação de otimalidade dual:

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

3.3.9 Correspondência entre teoria e implementação

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

Descrição Teórica	Implementação Python

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

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

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

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

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

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

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

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

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

Tabela 1 – Correspondência entre a descrição teórica do algoritmo de András Frank e sua implementação em Python. Cores: inicialização (azul), elevação de potenciais (verde/laranja/roxo/amarelo), verificação (ciano), construção (vermelho) e validação dual (rosa).

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

4 Chu-liu / Edmonds vs. Frank

Neste capítulo, apresentamos uma análise comparativa entre os algoritmos de Chu–Liu–Edmonds e András Frank para o problema da arborescência geradora de peso mínimo. Ambas metodologias são equivalentes e resolvem o mesmo problema, mas adotam estratégias distintas na redução de pesos e na construção da solução.

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

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

A seguir, apresentamos os experimentos empíricos que avaliam o comportamento prático dessas metodologias em termos de tempo de execução, consumo de memória e características estruturais dos digrafos processados. Vale salientar que este trabalho não tem como objetivo explorar otimizações específicas, mas sim o comportamento dos algoritmos em sua forma clássica.

4.1 Análise comparativa dos algoritmos

Conduzimos 2000 experimentos em digrafos aleatórios construídos a partir de uma raiz r com quantidade de vértices $|V| \in [101, 4996]$ (mediana 2464), número de arestas $|A|$ proporcional ao número de vértices com densidade média de $1{,}98|V|$ arcos e pesos inteiros associados $\in [1, 50]$. Para cada instância, executamos Chu–Liu/Edmonds e as duas fases de András Frank com duas versões v1 e v2 para a Fase II, sendo a v2 uma versão otimizada utilizando fila de prioridade - *heap*.

Os 2000 testes confirmaram que todos os métodos retornam sempre o mesmo peso, validando a corretude das implementações e a equivalência teórica entre Chu–Liu/Edmonds e Frank. Além disso, as verificações da condição de otimalidade dual foram bem-sucedidas em todos os casos para ambas as variantes de Frank.

Quanto ao desempenho temporal, a Fase I de Frank apresenta tempo mediano de $8{,}93$ s (média $12{,}40$s), significativamente superior ao tempo mediano de Chu–Liu/Edmonds ($0{,}25$s, média $0{,}58$s). A Fase I, responsável pela identificação de subconjuntos minimais via componentes fortemente conexas, domina o tempo total de execução

do método de Frank. A Fase II, em contrapartida, representa uma fração residual do processamento com mediana de $0{,}98$s para versão 1 do algoritmo e $0{,}016$s para a versão 2.

As Figuras 33–38 apresentam os resultados experimentais.

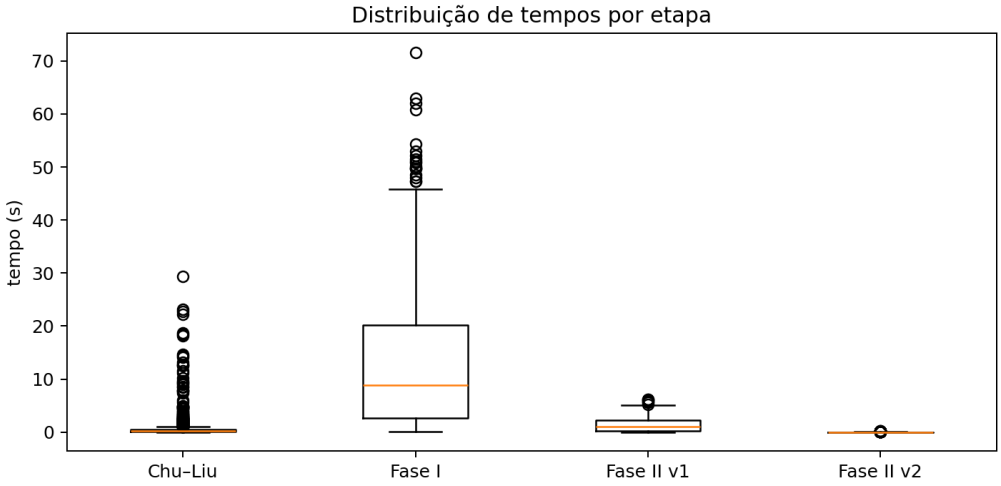


Figura 33 – Distribuição de tempos: Fase I apresenta maior mediana ($8{,}93$s) e variabilidade que Chu–Liu/Edmonds ($0{,}25$s).

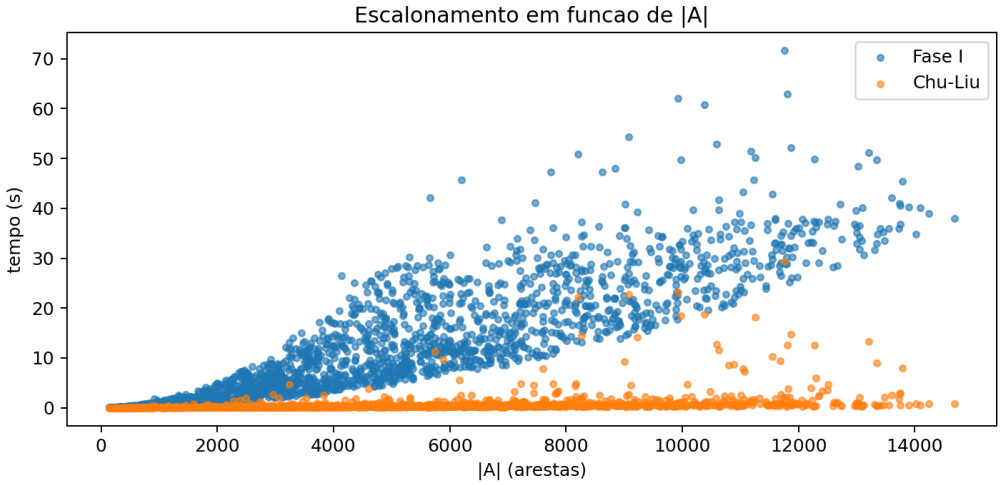


Figura 34 – Escalonamento temporal em função de $|A|$: crescimento aproximadamente linear.

A comparação entre as duas variantes da Fase II revela ganho expressivo com o uso de *heap*. A Figura 35 apresenta essa comparação em dois painéis: à esquerda, o boxplot evidencia a diferença de magnitude entre os tempos de execução — enquanto v1 (lista) apresenta mediana de $0{,}98$s, a versão v2 (heap) reduz para $0{,}016$s. À direita, o histograma do fator de aceleração (*speedup*) mostra distribuição com mediana de 58,12 vezes e média de 61,30 vezes, com metade das instâncias apresentando aceleração

entre 28 e 91 vezes (intervalo interquartil). Esses resultados confirmam empiricamente a vantagem da estrutura de dados com complexidade $O(\log n)$ versus $O(n)$ por operação de seleção de mínimo.

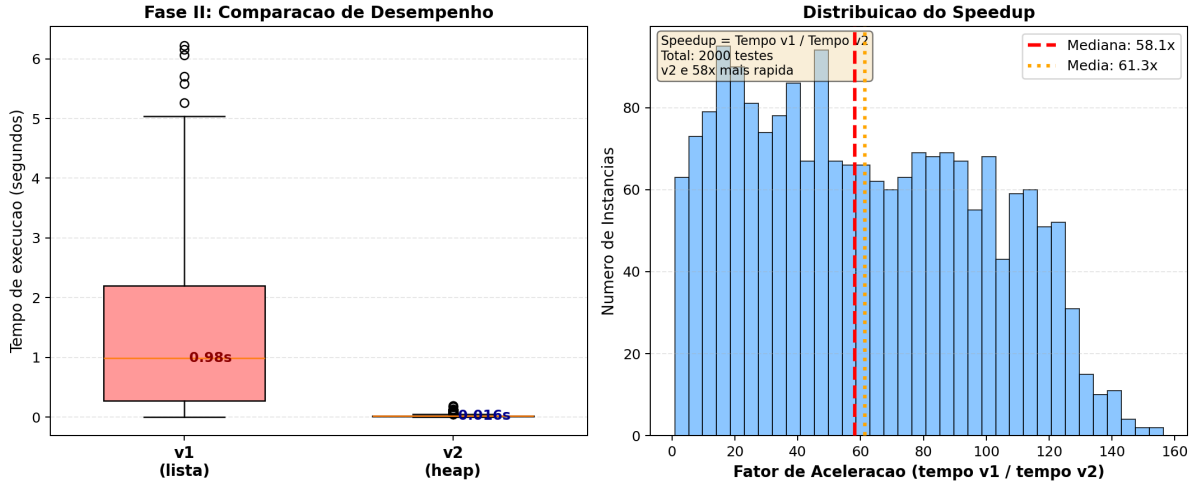


Figura 35 – Comparação de desempenho entre implementações da Fase II. À esquerda, o gráfico de boxplot mostra a distribuição de tempo da v1 do algoritmo com mediana de $0{,}98$s enquanto v2 reduz para $0{,}016$s. À direita, o histograma do fator de aceleração (*speedup*) mostra a distribuição concentrada entre 40 e 80 vezes, com mediana de 58,12 vezes (linha tracejada vermelha) e média de 61,30 vezes (linha pontilhada laranja).

As métricas estruturais do algoritmo de Chu–Liu/Edmonds são apresentadas na Figura 36 em dois histogramas. O painel esquerdo mostra que o número de contrações é pequeno (mediana 2, média 6,82, máximo 406), com a grande maioria das instâncias requerendo menos de 20 contrações. O painel direito exibe a profundidade de recursão, que acompanha o número de contrações.

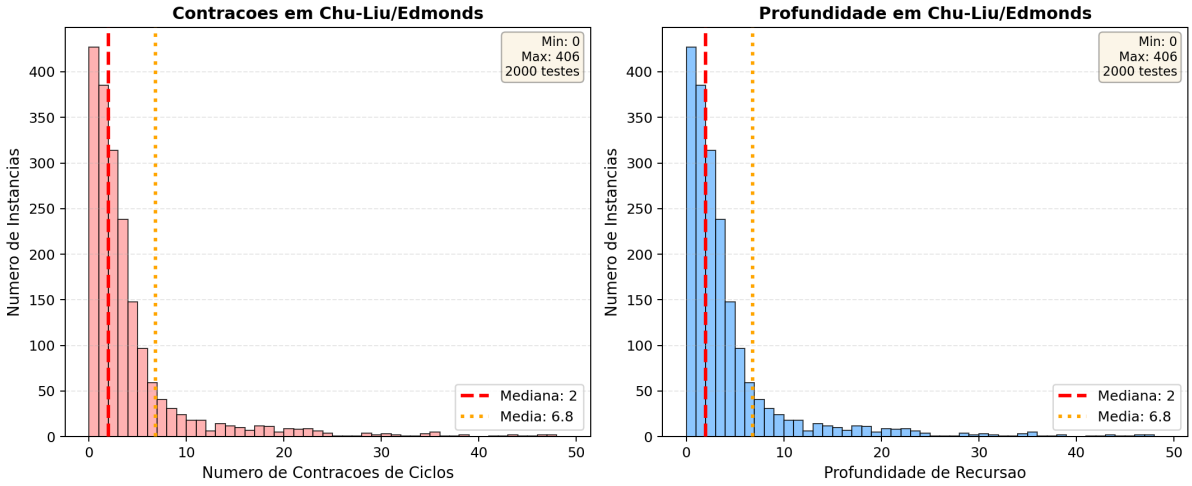


Figura 36 – Métricas estruturais de Chu–Liu/Edmonds. À esquerda, o histograma vermelho mostra que o número de contrações (eixo horizontal) é concentrado em valores baixos — a maioria das 2000 instâncias (eixo vertical) requer menos de 20 contrações, com mediana 2 (linha tracejada) e média $6{,}82$ (linha pontilhada). À direita, o histograma azul da profundidade de recursão exibe padrão similar).

O consumo de memória na Fase I mantém-se modesto (mediana 11,5 MB, média 14,8 MB), viabilizando a aplicação dos algoritmos mesmo em ambientes com recursos limitados.

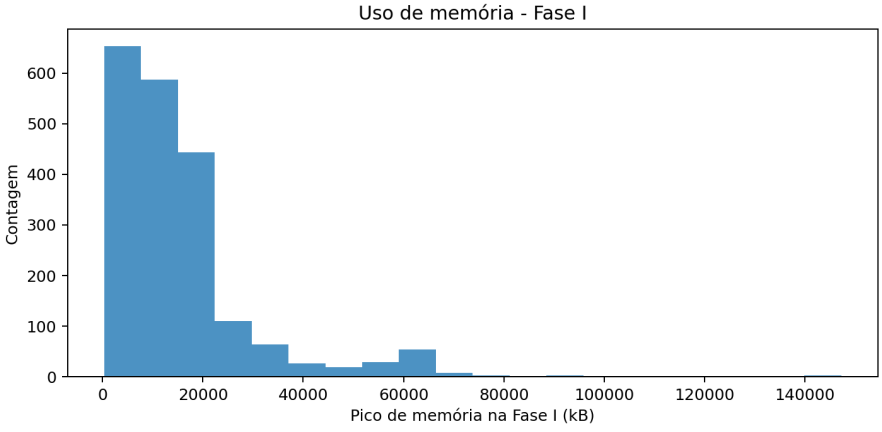


Figura 37 – Pico de memória na Fase I: mediana $11{,}5$ MB.

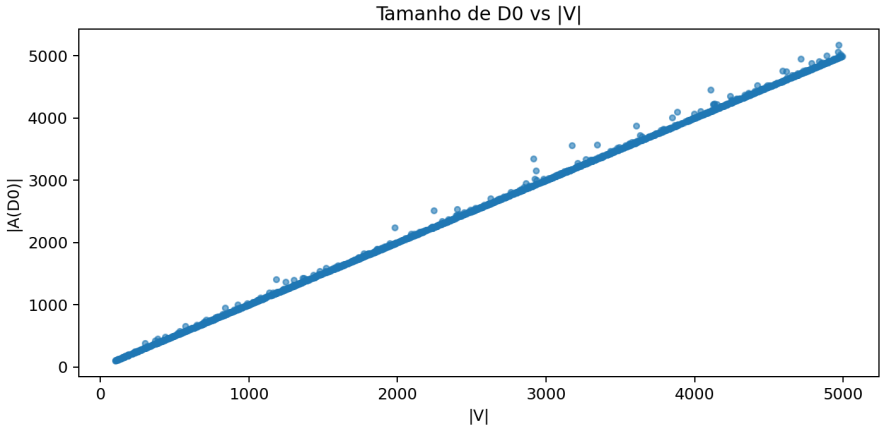


Figura 38 – Tamanho de D_0 versus $|V|$: relação linear confirma $|A_0| = O(|V|)$.

4.2 Conclusões

Os experimentos validam empiricamente as previsões teóricas e revelam características importantes do comportamento prático dos algoritmos. A equivalência de pesos é verificada em todas as 2000 instâncias, somada à verificação bem-sucedida das condições de otimalidade dual.

Chu–Liu/Edmonds demonstra-se mais eficiente para construção direta de arborescências nas instâncias aleatórias testadas (mediana $0{,}25$s, média $0{,}58$s), enquanto o método de Frank apresenta tempo adicional significativo na Fase I (mediana $8{,}93$s, média $12{,}40$s) devido ao processamento de subconjuntos minimais via componentes fortemente conexas. A versão heap da Fase II valida os ganhos assintóticos esperados, com aceleração de 58,12 vezes (mediana) sobre a versão linear, demonstrando que melhorias algorítmicas fundamentais se traduzem em benefícios práticos mensuráveis.

Observamos também que o comportamento em digrafos aleatórios mostra que o número de contrações observado tem mediana 2 e média 6,82 e o subgrafo D_0 mantém razão $|A_0|/|V_0| \approx 1{,}00$. O consumo modesto de memória (mediana 11,5 MB) e a escalabilidade observada viabilizam a aplicação prática de ambos os métodos em contextos com recursos computacionais limitados.

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

5 A Didática do Abstrato

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

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

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

5.0.1 Fundamentos cognitivos e didáticos

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

Aprender conteúdos de alta abstração envolve lidar com sobrecarga cognitiva intrínseca e extrínseca (SWELLER, 1988). Diretrizes de aprendizagem multimídia indicam que combinar representações verbais e visuais pode reduzir carga desnecessária e favorecer integração semântica (MAYER, 2009; PAIVIO, 1990). Em matemática avançada, a transição entre níveis de representação (intuitivo, formal, simbólico) exige mediação cuidadosa (TALL, 1991) e atenção a como exemplos, contraexemplos e invariantes são apresentados.

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

5.0.2 Lidando com grafos e digrafos

Na prática, o principal obstáculo para o ensino de dígrafos não está na definição de vértices e arcos, mas na articulação entre as operações realizadas nessas estruturas, os algoritmos que as exploram e as provas de correção que os fundamentam. No contexto desta dissertação, isso inclui compreender as arborescências de custo mínimo e os métodos clássicos de Chu–Liu/Edmonds e de Frank. Quando essa articulação não é explicitada, observa-se um aumento simultâneo da *carga intrínseca* (intrínseca devido às múltiplas dependências conceituais) e da *carga extrínseca* (decorrente do esforço de coordenar texto, fórmulas e representações gráficas)

A partir de uma análise reflexiva sobre o próprio processo de aprendizado das autoras, foi possível mapear onde a compreensão falha ou se torna nebulosa. Essas impressões pessoais permitiram sistematizar as dificuldades em três eixos didáticos principais, discutidos a seguir.

(1) Articulação entre decisões locais e coerência global. A articulação entre decisões locais e a integridade global da solução é um ponto crítico em ambos os métodos. Isso fica evidente na seleção da menor aresta de entrada para cada vértice: uma estratégia que parece intuitiva, mas que é enganosa. Escolhas localmente ótimas podem introduzir ciclos, violando a coerência global necessária para uma arborescência. O obstáculo principal está na crença equivocada de que regras de decisão simples e locais resultam automaticamente em uma solução global válida

(2) Acompanhamento dos efeitos de contração e expansão. Um segundo obstáculo central é o gerenciamento das mudanças estruturais e numéricas durante o algoritmo. Métodos de contração de ciclos exigem que o estudante transite mentalmente entre três estados distintos: o grafo original, a versão condensada (onde ciclos tornam-se super-vértices) e a reexpansão final. O desafio não é apenas visual; cada contração altera os custos reduzidos das arestas e redefine quais cortes permanecem ativos. A dificuldade típica ocorre quando se perde a "rastreabilidade"entre essas visões: o aprendiz esquece que uma aresta escolhida no grafo condensado corresponde a uma aresta específica no original, mas com um custo modificado. Sem explicitar o que muda e o que se mantém invariante, o processo torna-se uma "caixa preta", onde o aluno segue passos mecanicamente sem compreender a conexão entre a solução do grafo menor e a do grafo maior.

(3) Relação entre a execução do algoritmo e a formulação primal–dual. Aqui

o desafio reside na interpretação teórica das ações mecânicas. Passos operacionais, como contrair ciclos, possuem significados precisos na análise Primal-Dual, que é o motor que garante a otimalidade da solução. No entanto, essa correspondência raramente é óbvia para o aprendiz. A dificuldade emerge quando o estudante executa o algoritmo como uma "receita de bolo", sem perceber que cada ação está, na verdade, manipulando variáveis duais para tornar restrições "justas"(tight). Explicitar esses vínculos é fundamental: o aluno precisa entender que modificar um custo no desenho do grafo não é apenas um truque aritmético, mas a atualização de um certificado de otimalidade que valida a resposta final.

5.0.3 Visualização e interação: princípios em uso

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

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

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

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

Esses princípios orientaram o desenvolvimento do sistema interativo criado neste trabalho. A ferramenta permite visualizar e manipular dígrafos, acompanhar a execução passo a passo dos métodos de Chu–Liu/Edmonds e de Frank e alternar entre representações essenciais ao entendimento dos algoritmos: o grafo original, as contrações de ciclos, os custos reduzidos e as reexpansões. Ao concentrar em um único

ambiente a estrutura gráfica, as operações do algoritmo e sua justificativa conceitual, a ferramenta busca reduzir a carga extrínseca e facilitar a construção de esquemas mentais integrados.

5.1 O ecossistema de ferramentas

Materiais que articulam teoria, evidências empíricas e interatividade tendem a favorecer transferência e retenção. Com base nesses fundamentos, realizamos um levantamento de ferramentas digitais relevantes para o ensino de grafos e dígrafos, buscando mapear soluções existentes, suas finalidades e limitações. Esse mapeamento permitiu identificar tanto o espaço de possibilidades quanto lacunas específicas que motivaram o desenvolvimento da aplicação proposta neste trabalho.

De modo geral, ferramentas digitais podem reduzir carga extrínseca e facilitar a integração entre registros visual, simbólico e formal quando a interação é projetada para promover engajamento ativo (MAYER, 2009; SWELLER, 1988; HUNDHAUSEN et al., 2002; NAPS et al., 2003). No entanto, as abordagens disponíveis atualmente distribuem-se em diferentes categorias, cada qual cobrindo apenas parte das necessidades envolvidas no ensino de algoritmos para arborescências.

A seguir, descrevemos quatro categorias de ferramentas digitais que podem apoiar o ensino de grafos e dígrafos, indicando para cada uma suas forças, limitações e exemplos representativos: (i) diagramas programáveis e tipografia matemática; (ii) exploração e edição de grafos; (iii) visualização de algoritmos; e (iv) ambientes programáveis e reprodutibilidade.

As ferramentas da primeira categoria permitem criar diagramas de grafos com semântica visual consistente, integrando-os a textos matemáticos. Essas ferramentas são úteis para ilustrar conceitos, definições e provas em materiais didáticos. Existem uma série de benefícios didáticos como semântica visual estável (mesmo conceito, apresentado da mesma forma), autoria próxima ao símbolo e ao texto (co-referência) e manutenção e versionamento fáceis. Podemos levantar como limitações o fato de que as interação costumam ser offline (figuras estáticas) e a curva de aprendizado de sintaxe pode ser um obstáculo inicial. Em contextos de prova e definição, esses recursos ancoram a narrativa formal com diagramas que obedecem às diretrizes de (LARKIN; SIMON, 1987; WARE, 2012).

Já ferramentas de exploração e edição de grafos permitem que os usuários interajam com representações gráficas de dados, facilitando a manipulação e a análise de estruturas complexas. Essas ferramentas são essenciais para atividades que exigem uma compreensão profunda das relações entre os elementos de um grafo. Elas são adequadas para: reconhecer padrões estruturais (componentes, comunidades), discutir implicações

de layouts para percepção de estruturas, atividades de descoberta assistida ("*overview → filter → details*") (SHNEIDERMAN, 1996). Porém existem algumas limitações, como o fato dessas ferramentas serem focadas para análise exploratória de dados, não em algoritmos específicos, alta carga extrínseca ao alternar entre interface gráfica e conceitos teóricos, além falta de controle fino sobre estados intermediários de algoritmos.

Sobre as ferramentas de visualização de algoritmos, elas são projetadas para ilustrar o funcionamento interno de algoritmos através de animações e representações gráficas. Essas ferramentas são particularmente eficazes para demonstrar processos dinâmicos e mudanças de estado ao longo do tempo. Evidências sugerem ganhos quando o estudante prevê, manipula e explica o que vê, ao invés de consumir animações passivamente. (HALIM et al., ; HUNDHAUSEN et al., 2002; NAPS et al., 2003)

Ainda temos ferramentas em ambientes programáveia, elas são valiosas para criar exemplos reprodutíveis e explorar algoritmos de forma prática.Contudo, requerem familiaridade com programação e podem introduzir carga extrínseca se o foco se desviar para detalhes de implementação. A curadoria do conteúdo é essencial para manter o foco didático e evitar dispersão.

A Tabela 2 sumariza as categorias discutidas, com suas forças, limitações e exemplos representativos.

Categoria	**Forças**	**Limitações**	**Exemplos**
Diagramas programáveis e tipografia matemática	Semântica visual estável; co-referência entre texto e diagrama; manutenção e versionamento fáceis; layouts consistentes.	Interação offline; curva de aprendizado de sintaxe; pouco adequadas para representar dinâmica de algoritmos.	*Graphviz/dot*; *TikZ/PGF*
Exploração e edição de grafos	Layouts automáticos; suporte a filtros e métricas; adequadas para investigar padrões estruturais e relações complexas.	Foco maior em análise do que em algoritmos; alternância entre interface gráfica e conceitos teóricos aumenta carga extrínseca; pouco controle sobre estados intermediários.	*Gephi*; *yEd*; *Cytoscape*
Visualização de algoritmos	Explica transformações dinâmicas; eficaz para raciocínio procedimental; ganhos quando o estudante prevê, manipula e explica o que vê.	Pode induzir consumo passivo; limita personalização e integração com código.	*VisuAlgo*; repositórios de animações de algoritmos
Ambientes programáveis e reprodutibilidade	Alta flexibilidade; integra código, texto e visualizações; ideal para exploração prática e criação de exemplos reprodutíveis.	Exige familiaridade com programação; risco de dispersão em detalhes técnicos; requer curadoria didática.	*Jupyter Notebooks*; *NetworkX*

Tabela 2 – Síntese das categorias de ferramentas digitais para o ensino de grafos e digrafos.

Esse levantamento mostra que, embora existam soluções robustas em cada categoria, nenhuma delas integra de forma unificada a visualização de algoritmos, a manipulação direta do grafo com acesso aos estados intermediários (como contrações, custos reduzidos e reexpansões) e a coordenação entre a representação visual e a explicitação conceitual subjacente. Diante dessas lacunas, desenvolvemos uma aplicação web interativa que combina visualização algorítmica e manipulação gráfica, com foco específico nos procedimentos que compõem a construção de arborescências de custo mínimo. Na seção seguinte, apresentamos os princípios de interação humano–computador

que orientaram esse design e detalhamos como a aplicação se posiciona no ecossistema identificado.

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

Discutimos até aqui fundamentos teóricos dos algoritmos, análises de complexidade, resultados empíricos e princípios pedagógicos que justificam o uso de ferramentas interativas. Estabelecemos *o quê* ensinar (Chu–Liu/Edmonds e Frank), *por quê* usar visualizações (redução de carga cognitiva, engajamento ativo).

O passo seguinte consiste em transformar essas diretrizes em uma solução concreta. A aplicação *web* desenvolvida busca materializar os princípios apresentados, integrando escolhas didáticas e heurísticas de IHC em uma interface que favorece exploração gradual, leitura orientada e compreensão progressiva. Assim, passamos do plano conceitual para o plano operacional: *como* traduzir teoria e princípios pedagógicos em decisões de design

Mas antes de fundamentar os conceitos de design utilizados iremos apresentar o objetivo da aplicação, sua estrutura e principais funcionalidades, bem como o fluxo de interação proposto para os usuários.

6.1 Descrição da aplicação

A aplicação *web* foi concebida como uma ferramenta educacional interativa para auxiliar na compreensão dos algoritmos de Chu-Liu/Edmonds e Andras Frank para a construção de r-arborescências dirigidas de custo mínimo. Seu principal objetivo é permitir que estudantes e pesquisadores explorem visualmente o funcionamento desses algoritmos, acompanhando passo a passo suas operações em grafos dirigidos.

6.1.1 Estrutura e Funcionalidades

A aplicação organiza-se em módulos funcionais projetados para atender aos objetivos didáticos do projeto, distribuídos em três eixos principais de navegação. O primeiro eixo, dedicado à visualização algorítmica, reúne três páginas focadas na execução passo a passo dos métodos estudados. Uma delas apresenta o algoritmo de Chu-Liu/Edmonds, enquanto as outras duas exploram diferentes implementações do algoritmo de Frank. Nessas páginas, o sistema mostra as iterações de forma sequencial, evidencia as alterações visuais no grafo e exibe o resultado final da r-arborescência.

O segundo eixo corresponde à modelagem livre, disponibilizando uma interface de edição em formato de sandbox na qual o usuário pode desenhar grafos arbitrariamente.

Esse espaço oferece liberdade para experimentar diferentes topologias e estruturas, permitindo testar hipóteses além dos exemplos previamente definidos.

Por fim, o terceiro eixo concentra-se na disseminação científica, reunindo uma página informativa dedicada à divulgação da dissertação e do projeto. Seu propósito é contextualizar a ferramenta, esclarecer as motivações de seu desenvolvimento e servir como canal de difusão do conhecimento teórico acerca dos algoritmos implementados.

6.1.2 Fluxo de interação

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

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

6.1.3 Arquitetura do Sistema

A arquitetura da aplicação foi projetada para operar integralmente no lado do cliente (*client-side*), utilizando o navegador como ambiente de execução para o código Python via WebAssembly. O sistema estrutura-se em três camadas lógicas principais:

- **Camada de Apresentação:** Responsável pela interface com o usuário, estruturada em HTML5, estilizada com o *framework* utilitário Tailwind CSS e dinamizada por JavaScript. Esta camada gerencia a entrada de dados, a interatividade dos elementos e a exibição dos resultados, mantendo a responsividade em diferentes dispositivos.

- **Núcleo de Processamento (PyScript):** Atua como a ponte entre a interface *web* e as bibliotecas científicas. O PyScript permite a importação e execução de módulos Python diretamente no DOM. O processamento dos grafos é realizado pela biblioteca `NetworkX`, enquanto a geração das representações visuais estáticas (snapshots) é delegada ao `Matplotlib`.

- **Camada de Dados e Persistência:** A troca de informações entre o editor livre e os algoritmos utiliza o formato JSON (*JavaScript Object Notation*). Os grafos são serializados no padrão `node_link`, permitindo a representação leve de nós, arestas e atributos (pesos e custos) para armazenamento local ou transferência entre módulos.

Esta organização modular assegura que cada página carregue apenas os *scripts* necessários para sua função específica, otimizando o tempo de carregamento e o consumo de recursos do navegador.

6.2 Princípios de interação humano-computador

A Interação Humano-Computador (IHC) orienta o design de sistemas para que sejam, simultaneamente, eficientes, eficazes e agradáveis ao usuário. Nesse contexto, promovemos uma síntese entre heurísticas clássicas de usabilidade (NIELSEN, 1994; SHNEIDERMAN et al., 2016) e teorias de aprendizagem e carga cognitiva (ROGERS et al., 2011; MAYER, 2009; SWELLER, 1988; NAPS et al., 2003), resultando em oito princípios norteadores: (i) usabilidade, (ii) eficiência cognitiva, (iii) feedback imediato, (iv) engajamento ativo, (v) visão geral com detalhe sob demanda (o mantra overview $\rightarrow$ filter $\rightarrow$ details de (SHNEIDERMAN, 1996)), (vi) consistência semântica, (vii) múltiplos registros de representação e (viii) prevenção e recuperação de erros. A seguir, detalhamos cada princípio e sua operacionalização na ferramenta.

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

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

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

Esses elementos favorecem o engajamento ativo, que refere-se à participação dos usuários no processo de aprendizagem, incentivando-os a explorar, experimentar e

interagir com o sistema. Nossa aplicação promove o engajamento ativo ao permitir que os usuários manipulem o grafo, testem diferentes abordagens e visualizem os resultados de suas ações em tempo real, além de acessar descrições passo a passo da execução dos algoritmos.

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

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

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

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

Esses princípios de interação humano-computador foram fundamentais para o desenvolvimento da nossa aplicação *web* interativa, garantindo que ela seja não apenas funcional, mas também acessível e eficaz como ferramenta de aprendizagem. A seguir, detalhamos a implementação técnica da aplicação e como ela se posiciona no ecossistema de ferramentas didáticas para o ensino de grafos.

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

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

6.3 Detalhes de Implementação

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

6.3.1 Estrutura de arquivos

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

6.3.2 Páginas da Aplicação *web*

A seguir, apresentamos as páginas que compoem a ferramente desenvolvida.

`Home`:

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

Figura 39 – Captura de tela de `home.html`: visão geral com resumo e integrantes.

`Draw graph`:

Editor de grafos livre com funcionalidades de criação, edição, importação e exportação. Ele constitui o segundo módulo que definimos na seção de 6.1.1 Utiliza `Cytoscape.js` para visualização interativa e `PyScript` para lógica algorítmica. Abaixo, apresentamos uma captura de tela da página.

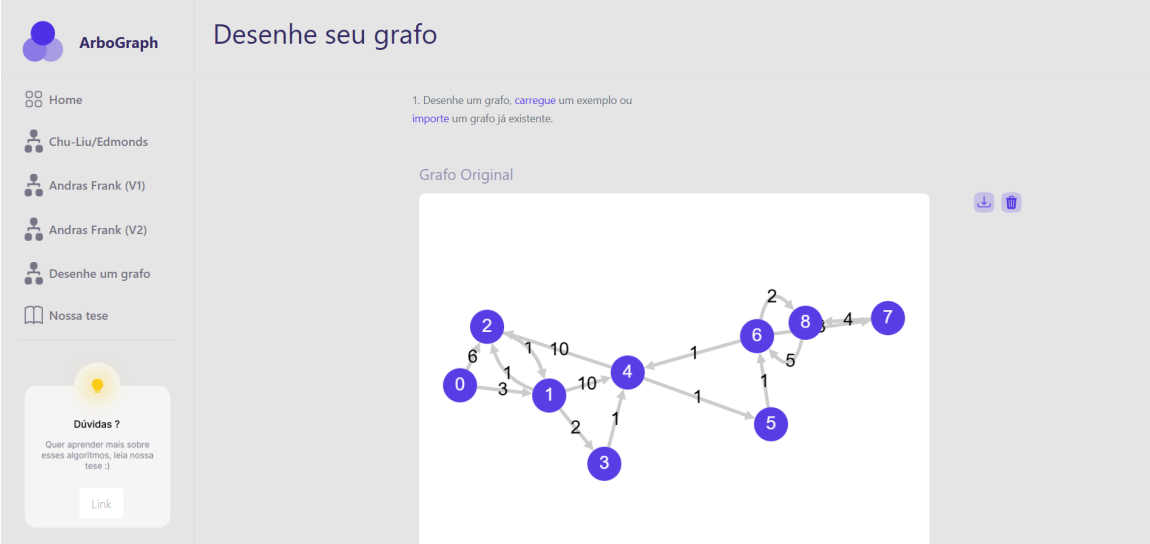


Figura 40 – Captura de tela de `draw_graph.html`: editor livre de grafos.

`Tese`:

Essa página compoem o módulo 3 apresentado na seção 6.1.1, nela apresentamos nossa tese bem como permitimos o download da mesmo, o intuito é permitir com que o usuário possa ter um entendimento maior tanto dos algoritmos quanto do nosso sistema.

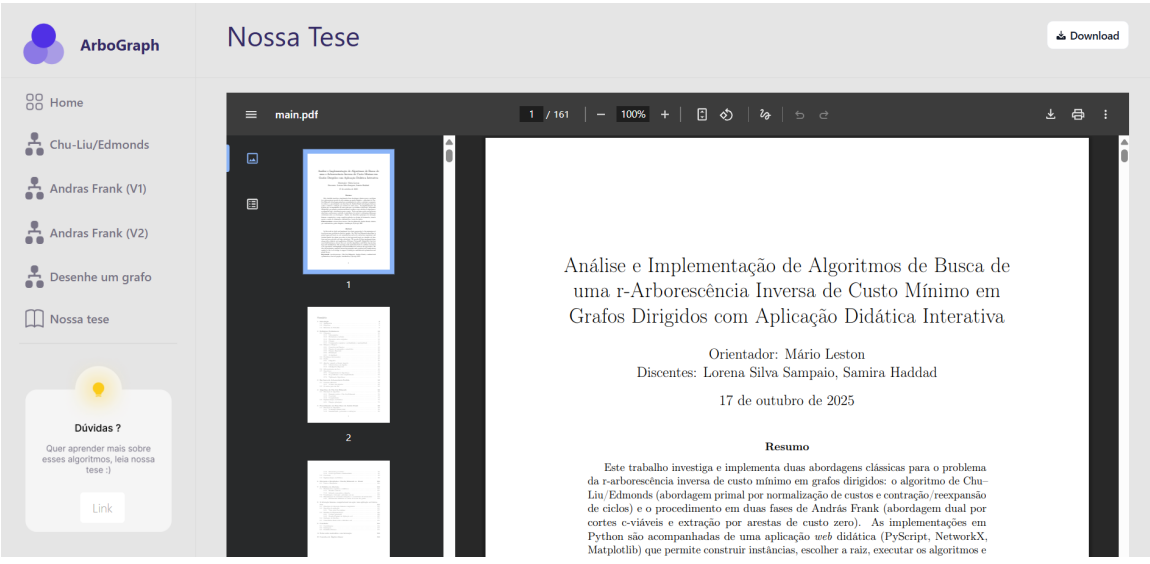


Figura 41 – Captura de tela de `tese.html`: visão geral com resumo e integrantes.

`Chuliu/Edmonds`:

página dedicada ao visualizador do algoritmo de Chu-Liu/Edmonds. Inclui um passo a passo guiado para criar um grafo, selecionar o nó raiz e executar o algoritmo, com feedback visual e textual. ELe faz parte do primeiro módulo destacado na seção 6.1.1. Abaixo, apresentamos uma captura de tela da página.

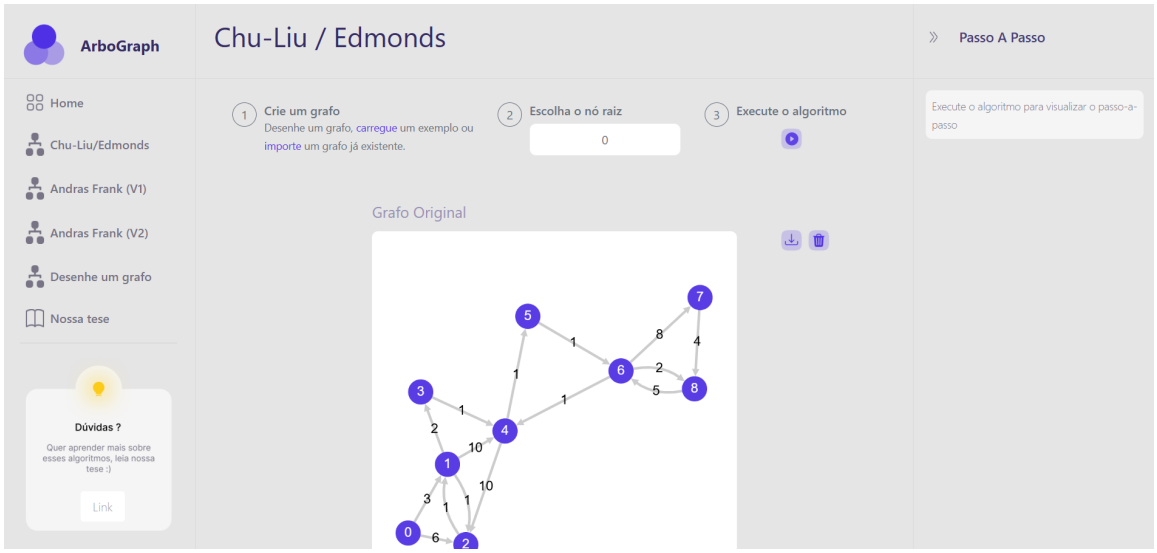


Figura 42 – Captura de tela de `chuliu.html`: criação de grafo, seleção de raiz e execução do algoritmo.

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

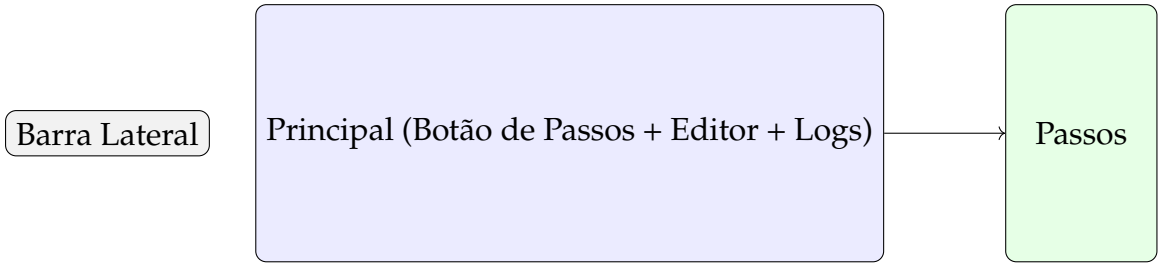


Figura 43 – Tripartição funcional (navegação, conteúdo interativo, guia de passos). A presença do passo a passo auxilia na compreensão sequencial do algoritmo.

6.3.3 Página do Andrasfrank (v1) e Andrasfrank (v2):

Ambas as páginas são dedicadas ao visualizador do algoritmo de Andras Frank (em suas diferentes implementações elucidadas em capítulos anteriores). Assim como na página dedicada ao algoritmo de Chu–Liu/Edmonds, essa página também faz parte do módulo 1 definido na seção 6.1.1 Além disso, ele inclui um passo a passo guiado para criar um grafo, selecionar o vértice raiz e executar o algoritmo, com feedback visual e textual. Abaixo, apresentamos uma captura de tela da página.

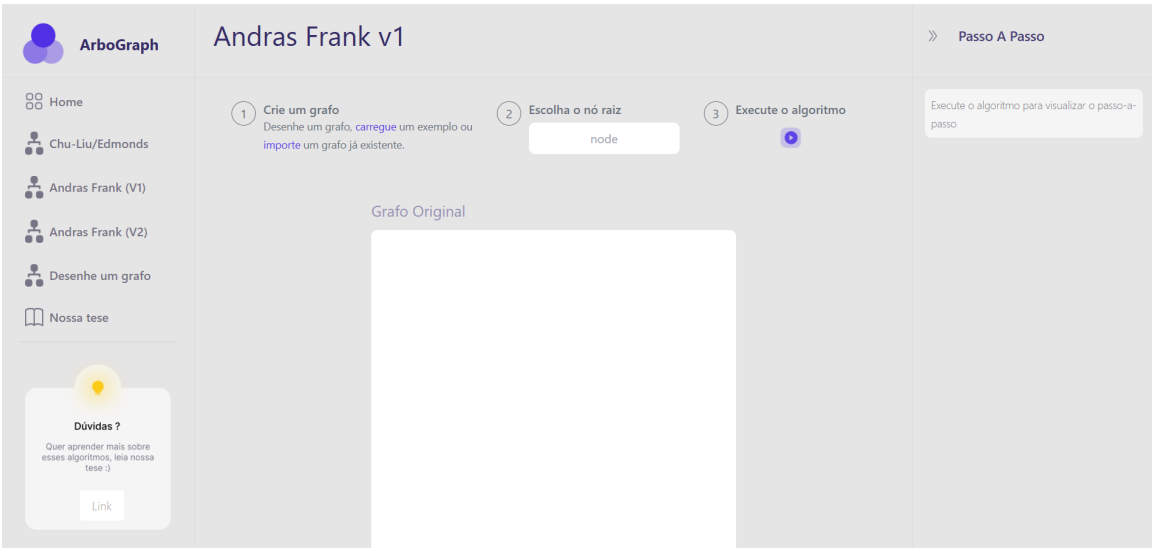


Figura 44 – Captura de tela de `andrasfrank_v1.html`: interface para o procedimento em duas fases, a tela da página `andrasfrank_v2.html` tem aparência similar.

Assim como na página do algoritmo de Chu–Liu/Edmonds (fig. **??**) a página dedicadas às implementações do algoritmo do Andras Frank também utilização o padrão de tripartição funcional para manter consistência cognitiva entre páginas.

6.3.4 Estrutura das páginas

A estrutura das páginas da aplicação segue um padrão modular e reutilizável, facilitando a manutenção e extensão futura. Cada página é composta por três seções principais: uma barra lateral de navegação, um conteúdo interativo central e um painel de passos à direita. Essa organização promove consistência cognitiva e facilita a navegação do usuário entre diferentes funcionalidades.

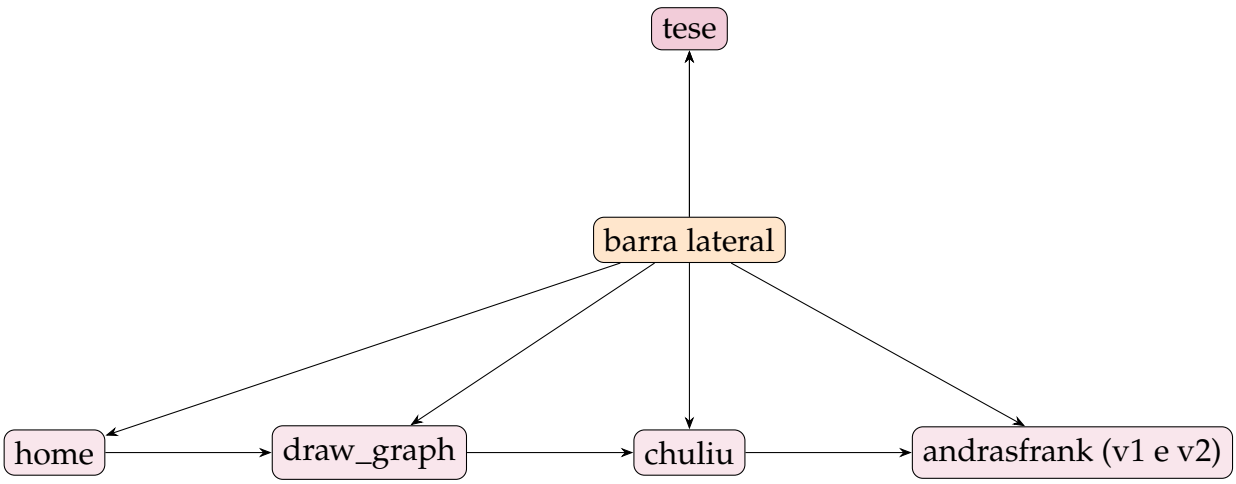


Figura 45 – A `barra lateral` injeta navegação consistente; páginas de algoritmo formam trilha exploratória.

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

6.4 Considerações Finais e Trabalhos Futuros

Este capítulo detalhou a implementação técnica do visualizador interativo de arborescências dirigidas, cobrindo desde a arquitetura *web* até a integração de algoritmos complexos. A escolha de tecnologias modernas como HTML5, CSS3, JavaScript e PyScript permitiu criar uma interface intuitiva e responsiva, facilitando a experimentação e a compreensão dos algoritmos de Chu–Liu/Edmonds e Andras Frank. A reutilização de padrões arquiteturais promoveu consistência cognitiva, enquanto a estrutura modular assegura a manutenção e a extensibilidade do projeto. Não obstante os resultados alcançados, a aplicação apresenta oportunidades de evolução para aprofundar a experiência didática. Atualmente, a ausência de uma visualização explícita da contração de ciclos e a falta de uma comparação lado a lado entre as abordagens de Chu–Liu e Frank limitam a clareza de certos processos. Para endereçar essas questões, planeja-se incorporar animações de contração com agrupamentos colapsáveis e desenvolver um módulo comparativo dedicado, incluindo uma extensão paralela para a abordagem primal–dual de Frank.No aspecto visual, o layout planar simples impõe restrições em instâncias densas. A mitigação desse problema prevê a adoção de layouts adaptativos (como *spring* ou *dagre-like*) e a implementação de uma camada de sinalização cromática para custos reduzidos e arcos críticos ($c' = 0$). Por fim, a funcionalidade de exportação será expandida para gerar relatórios automáticos (PDF/ZIP) que contemplem estados intermediários e métricas de desempenho, superando a limitação atual de exportar apenas os grafos inicial e final. No próximo capítulo, discutiremos as conclusões gerais da dissertação.

7 Conclusão

Ao final desta tese, refletimos sobre o ponto de chegada: o visualizador interativo de arborescências dirigidas. Este projeto foi concebido com o propósito de apresentar de forma didática os algoritmos envolvidos no problema da r-arborescência de custo mínimo.

Nessa trajetória, os fins não justificam os meios; pelo contrário, os meios pelos quais buscamos a solução, os algoritmos de Chu–Liu/Edmonds e o de Andras Frank, constituem o ponto central de discussão e a essência deste trabalho.

O algoritmo de Chu-Liu/Edmonds é um clássico da teoria dos grafos, com diversas aplicações práticas. Já o algoritmo de Andras Frank, embora menos conhecido, oferece uma abordagem elegante e eficiente para o mesmo problema, utilizando conceitos avançados de otimização combinatória. Ambos os algoritmos possuem complexidades e sutilezas que podem ser desafiadoras para estudantes e profissionais que buscam compreendê-los profundamente. Nesse contexto ressaltamos invariantes, cortes e custos reduzidos, e mostrando como escolhas locais se conectam a garantias globais de otimalidade. Procuramos menos descrever "o que o algoritmo faz" e mais explicitar "por que" cada passo se justifica, aproximando a mecânica operacional da linguagem primal–dual e de suas condições de complementaridade.

Os algoritmos foram implementados em Python, aproveitando bibliotecas como NetworkX para manipulação de grafos e Matplotlib para visualização. A escolha do Python se deve à sua sintaxe clara e à vasta gama de bibliotecas científicas disponíveis, facilitando tanto a implementação quanto a compreensão dos algoritmos. A integração com PyScript permitiu que esses algoritmos fossem executados diretamente no navegador, eliminando a necessidade de instalações complexas e tornando a ferramenta acessível a um público mais amplo. Os resultados foram validados através de testes com grafos de diferentes tamanhos e estruturas, garantindo a correção e eficiência das implementações.

A interface *web* foi projetada com foco na usabilidade e na experiência do usuário, utilizando HTML5, CSS3 e JavaScript para criar uma plataforma interativa e intuitiva obedecendo a princípios de design centrados no usuário orientados por princípios de interação humano-computacional. A estrutura modular da página permite fácil navegação entre diferentes seções, como a criação de grafos, a execução dos algoritmos e a visualização dos resultados. Elementos interativos, como botões, menus suspensos e áreas de desenho, foram incorporados para facilitar a interação do usuário com a ferramenta. A reutilização de padrões arquiteturais entre as páginas promoveu

consistência cognitiva, enquanto a sequência lógica de páginas guia o usuário do contexto à experimentação e formalização, alinhando-se a princípios pedagógicos orientados pela teoria da aprendizagem.

A partir desse desenvolvimento, o visualizador interativo de arborescências dirigidas se apresenta como uma ferramenta valiosa para estudantes, educadores e profissionais interessados em teoria dos grafos e algoritmos de otimização. A seguir destacamos algumas contribuições.

7.1 Contribuições

Este trabalho contribui para a interseção entre teoria dos grafos e design pedagógico, culminando no desenvolvimento de um visualizador interativo de arborescências dirigidas que promove a compreensão de algoritmos cruciais para o problema da r-arborescência de custo-mínimo.

A primeira e principal contribuição reside na integração detalhada de algoritmos complexos, como os de Chu–Liu/Edmonds e Andras Frank. A ferramenta não apenas implementa esses métodos, mas também destaca suas bases teóricas e operacionais, demonstrando como decisões locais se traduzem em garantias globais de otimalidade.

Este rigor teórico é complementado pelo design pedagógico da aplicação. Mediante a aplicação de princípios de Interação Humano-Computador (IHC) e de aprendizagem multimídia, o sistema foi desenhado para criar uma interface que facilita a compreensão de conceitos complexos, atuando ativamente na redução da carga cognitiva e promovendo o engajamento do usuário.

Tecnicamente, a contribuição se manifesta em uma arquitetura modular e reutilizável. O desenvolvimento utilizou tecnologias modernas como PyScript, NetworkX e Matplotlib, resultando em uma estrutura web que assegura fácil manutenção e consistência cognitiva entre as diferentes páginas. Essa escolha técnica culmina na criação de uma ferramenta acessível e reprodutível, que pode ser acessada diretamente no navegador, eliminando barreiras de adoção e permitindo que usuários experimentem e aprendam de forma autônoma.

Em suma, estas contribuições avançam o estado da arte na visualização e ensino de algoritmos de grafos, oferecendo uma plataforma que combina rigor teórico com práticas de design centradas no usuário.

7.2 Limitações

Apesar dos avanços alcançados, a implementação apresenta algumas limitações que devem ser consideradas. A complexidade dos algoritmos pode levar a tempos de execução elevados para grafos muito grandes, o que pode impactar a experiência do usuário. Além disso, a interface, embora intuitiva, pode beneficiar-se de melhorias adicionais em termos de acessibilidade e usabilidade, especialmente para usuários com menos experiência em manipulação de grafos (descrevemos essas melhorias na seção 6.4 do capítulo anterior). Outro ponto de atenção está relacionado a dependência de bibliotecas externas, como Cytoscape.js e PyScript, que podem introduzir desafios de compatibilidade e manutenção a longo prazo.

7.3 Trabalhos Futuros

Para aprimorar a ferramenta, futuras iterações podem focar em otimizações de desempenho para lidar com grafos maiores de forma mais eficiente. A interface pode ser refinada com base em testes de usabilidade, incorporando feedback de uma base de usuários diversificada. A adição de funcionalidades avançadas, como suporte a diferentes tipos de grafos e algoritmos adicionais, pode expandir o escopo da ferramenta. A integração de análises de aprendizado, como rastreamento do progresso do usuário e sugestões personalizadas, também pode enriquecer a experiência educacional. Finalmente, a realização de estudos formais para avaliar o impacto pedagógico da ferramenta em ambientes educacionais contribuirá para validar sua eficácia e orientar futuras melhorias.

Referências

CHU, Y. J.; LIU, T. H. On the shortest arborescence of a directed graph. *Scientia Sinica*, v. 14, p. 1396–1400, 1965. Citado na página 67.

EDMONDS, J. Optimum branchings. *Journal of Research of the National Bureau of Standards*, v. 71B, p. 233–240, 1967. Citado na página 67.

FRANK, A. A weighted matroid intersection approach to R-arborescences and related problems. In: FRANK, A. et al. (Ed.). *Paths, Flows, and VLSI-Layout*. [S.l.]: Springer, 1981. Two-phase primal–dual method for minimum-cost arborescences; placeholder citation. Citado na página 67.

FRANK, A.; HAJDU, G. A simple algorithm and min–max formula for the inverse arborescence problem. *Algorithms*, v. 7, n. 4, p. 637–647, 2014. Citado na página 67.

HALIM, S. et al. *VisuAlgo*. <https://visualgo.net/>. Acesso didático a visualizações interativas de algoritmos, acessado em 2025. Citado na página 76.

HUNDHAUSEN, C. D.; DOUGLAS, S. A.; STASKO, J. T. A meta-study of algorithm visualization effectiveness. *Journal of Visual Languages & Computing*, v. 13, n. 3, p. 259–290, 2002. Citado 3 vezes nas páginas 74, 75 e 76.

LARKIN, J. H.; SIMON, H. A. Why a diagram is (sometimes) worth ten thousand words. *Cognitive Science*, v. 11, n. 1, p. 65–100, 1987. Citado 2 vezes nas páginas 74 e 75.

MAYER, R. E. *Multimedia Learning*. 2nd. ed. [S.l.]: Cambridge University Press, 2009. Citado 4 vezes nas páginas 72, 74, 75 e 81.

NAPS, T. L. et al. Exploring the role of visualization and engagement in computer science education. In: *Proceedings of the 34th SIGCSE Technical Symposium on Computer Science Education*. [S.l.]: ACM, 2003. p. 131–152. Citado 4 vezes nas páginas 74, 75, 76 e 81.

NIELSEN, J. *Usability Engineering*. [S.l.]: Morgan Kaufmann, 1994. Fonte das heurísticas clássicas de usabilidade. ISBN 978-0125184069. Citado na página 81.

PAIVIO, A. *Mental Representations: A Dual Coding Approach*. [S.l.]: Oxford University Press, 1990. Citado 2 vezes nas páginas 72 e 74.

ROGERS, Y.; SHARP, H.; PREECE, J. *Interaction Design: Beyond Human-Computer Interaction*. 3rd. ed. [S.l.]: Wiley, 2011. ISBN 978-0470665763. Citado na página 81.

SHNEIDERMAN, B. The eyes have it: A task by data type taxonomy for information visualizations. In: *Proceedings 1996 IEEE Symposium on Visual Languages*. [S.l.]: IEEE, 1996. p. 336–343. Citado 2 vezes nas páginas 76 e 81.

SHNEIDERMAN, B. et al. *Designing the User Interface: Strategies for Effective Human-Computer Interaction*. 6th. ed. [S.l.]: Pearson, 2016. ISBN 978-0134380384. Citado na página 81.

SWELLER, J. Cognitive load during problem solving: Effects on learning. *Cognitive Science*, v. 12, n. 2, p. 257–285, 1988. Citado 3 vezes nas páginas 72, 75 e 81.

TALL, D. *Advanced Mathematical Thinking*. [S.l.]: Kluwer Academic Publishers, 1991. Citado 2 vezes nas páginas 72 e 74.

WARE, C. *Information Visualization: Perception for Design*. 3rd. ed. [S.l.]: Morgan Kaufmann, 2012. Citado 2 vezes nas páginas 74 e 75.

Anexos

ANEXO A – Anexo A

Conteúdo do anexo A.