Lorena Silva Sampaio, Samira Haddad

Algoritmos para o problema da arborescência geradora mínima: uma aplicação didática interativa

Brasil

2025

Lorena Silva Sampaio, Samira Haddad

Algoritmos para o problema da arborescência geradora mínima: 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 Rey

Brasil

2025

Dedicatória (opcional).

Agradecimentos

Agradecimentos (opcional).

“Epígrafe (opcional).”

Lorena Silva Sampaio, Samira Haddad

Algoritmos para o problema da arborescência geradora mínima: 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.

Trabalho aprovado. Brasil, 25 de novembro de 2025:

Prof. Dr. Mário Leston Rey
Orientador

Profa. Dra. Cristiane Maria Sato
Convidada

Prof. Dr. Aritanan Borges Garcia Gruber
Convidado

Brasil

2025

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 Introdução

Encontrar uma *r-arborescência geradora mínima* em digrafos é um problema clássico em Ciência da Computação estudado desde os anos 1960, com formulações fundamentais apresentadas por Jack Edmonds em 1967 (EDMONDS, 1967).

Neste trabalho implementamos dois métodos para esse problema: o algoritmo de Chu–Liu–Edmonds (CHU; LIU, 1965; EDMONDS, 1967) e o procedimento em duas fases de András Frank (FRANK, 2011; FRANK, 1979).

Além disso, desenvolvemos uma aplicação *web* interativa que permite a visualização passo a passo dessas metodologias, facilitando a compreensão de suas operações e estruturas subjacentes.

1.1 Estrutura do Trabalho

Resumidamente, esta dissertação abrange as seguintes frentes:

O capítulo 2 introduz definições e conceitos básicos sobre digrafos, arborescências etc, estabelecendo a notação empregada ao longo do texto.

O capítulo 3 detalha o algoritmo de Chu–Liu–Edmonds e sua implementação em `Python`.

O capítulo 4 apresenta o procedimento em duas fases de András Frank e sua respectiva implementação computacional.

O capítulo 5 discute os resultados experimentais comparativos entre as duas abordagens, avaliando desempenho temporal, consumo de memória e características estruturais dos digrafos processados.

O capítulo 6 explora fundamentos teóricos da didática de assuntos abstratos, destacando a importância de visualizações interativas para o aprendizado de algoritmos complexos.

O capítulo 7 descreve a construção da aplicação *web*, incluindo arquitetura, tecnologias empregadas e aspectos de design centrado no usuário com base em princípios de interação humano-computacional.

Finalmente, o capítulo 8 sintetiza as conclusões da pesquisa, suas contribuições, limitações e direções para trabalhos futuros.

2 Preliminares

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

2.1 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.

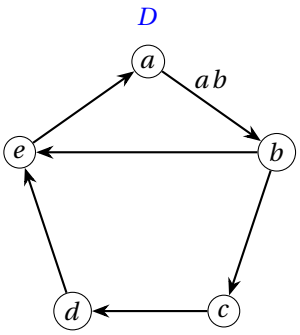


Figura 1 – A figura ilustra um digrafo D cujo conjunto dos vértices é $\{a, b, c, d, e\}$ e cujo conjunto dos arcos é $\{ab, bc, cd, de, ea, be\}$.

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 $w(a)$.

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

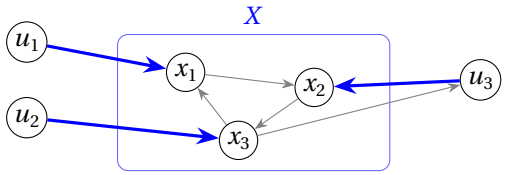


Figura 2 – Os arcos em azul entram em um subconjunto $X \subseteq V(D)$.

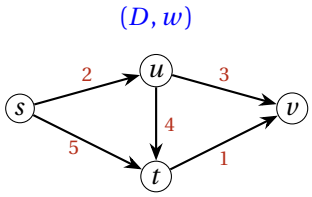


Figura 3 – Exemplo de digrafo ponderado (D, w): cada arco recebe um custo real $w(a)$.

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,

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

2.2 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.

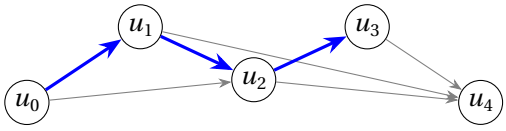


Figura 4 – A figura ilustra um caminho simples (u_0, u_1, u_2, u_3) em um digrafo.

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 $r \notin V(D)$.

Proposição 2.1. *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.* □

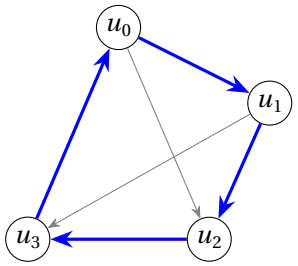


Figura 5 – A figura ilustra um ciclo $(u_0, u_1, u_2, u_3, u_0)$ em um digrafo.

2.3 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**.

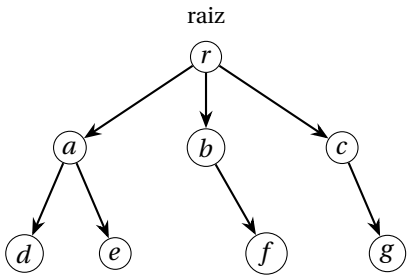


Figura 6 – A figura ilustra uma r-arborescência.

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

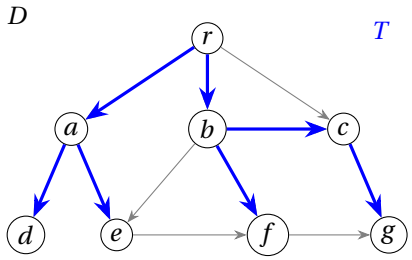


Figura 7 – O subdigrafo T (em azul) é uma arborescência geradora do digrafo D, que inclui tanto os arcos azuis quanto os arcos cinza.

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** se $D[B]$ é uma arborescência de D.

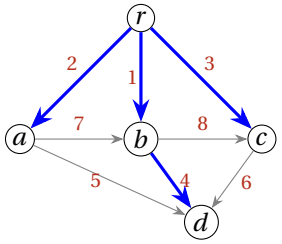


Figura 8 – A figura ilustra um subconjunto B (de arcos azuis) de custo $w(B) = 2+1+3+4 = 10$ visto como uma arborescência geradora de D.

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 2.2. *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.* □

2.4 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;
- $\delta^-(r) = \emptyset$; 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.

2.5 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 $\emptyset \neq X \subseteq V(D)$. A **contração** de X em D, denotada por D/X, é o digrafo $D/\mathscr{P}$, em que

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

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

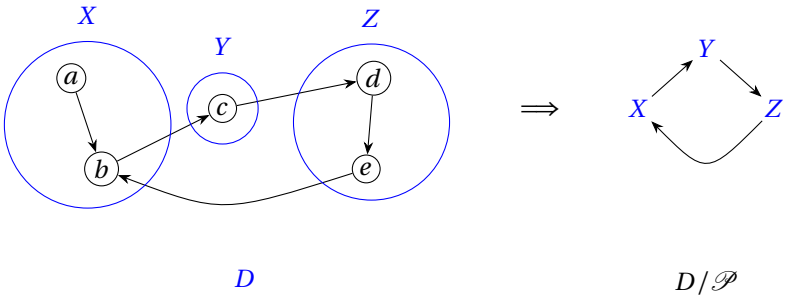


Figura 9 – Contração de um digrafo D pela partição $\mathscr{P} = \{X, Y, Z\}$, em que $X := \{a, b\}$, $Y := \{c\}$ e $Z := \{d, e\}$.

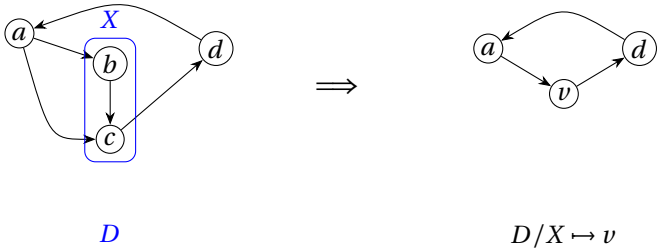


Figura 10 – Contração de um subconjunto $X \subseteq V(D)$ em um digrafo D. À esquerda, o subconjunto $X = \{b, c\}$ é destacado; à direita, os vértices de X foram identificados em um único vértice v, obtendo-se o digrafo $D/X \mapsto v$.

com um vértice $v \notin V(D)$. Para explicitar essa identificação, escreveremos $D/X \mapsto v$ em vez de simplesmente D/X.

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 $y \in Y$.

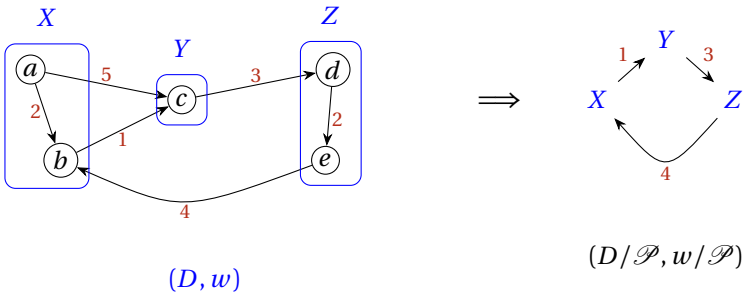


Figura 11 – Contração de um digrafo ponderado (D, w) pela partição $\mathscr{P} = \{X, Y, Z\}$, em que $X := \{a, b\}$, $Y := \{c\}$ e $Z := \{d, e\}$. O peso de cada arco XY em $D/\mathscr{P}$ é o mínimo dos pesos dos arcos de D que vão de um vértice de X para um vértice de Y.

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 $v \notin V(D)$.

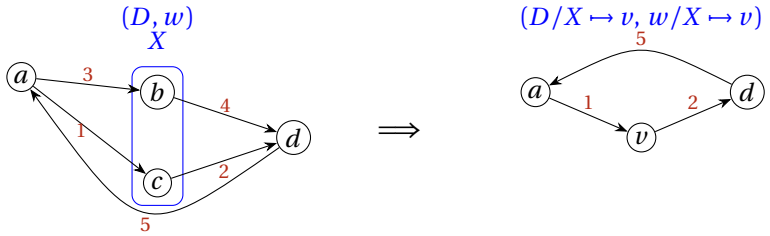


Figura 12 – Contração ponderada de um subconjunto $X \subseteq V(D)$. À esquerda, o digrafo (D, w) com $X = \{b, c\}$ destacado. À direita, o digrafo $(D/X \mapsto v, w/X)$, em que os vértices de X foram identificados em um único vértice v e cada peso w/X é o mínimo dos pesos dos arcos de D entre X e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

2.6 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).

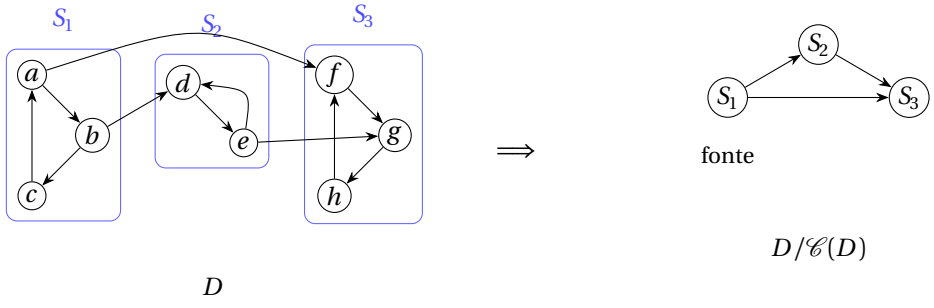


Figura 13 – Componentes fortes S_1, S_2, S_3 de um digrafo D e sua condensação $D/\mathscr{C}(D)$, um digrafo acíclico em que S_1 é fonte.

3 Algoritmo de Chu–Liu–Edmonds

Neste capítulo apresentaremos o algoritmo de Chu–Liu–Edmonds (CHU; LIU, 1965) (EDMONDS, 1967), 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.

3.1 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

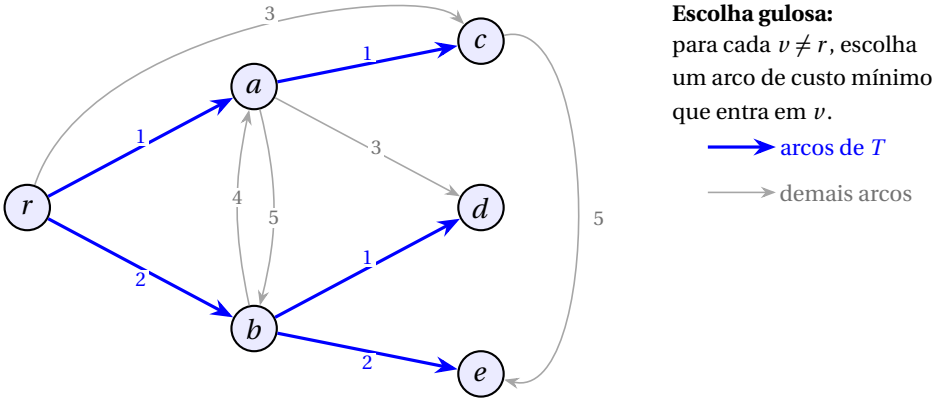


Figura 14 – A figura ilustra a escolha gulosa quando esta produz uma r-arborescência. Os arcos em azul são os escolhidos; os cinza são os demais arcos do digrafo.

para o *único* arco de F que entra em v. Pela escolha gulosa,

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

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.

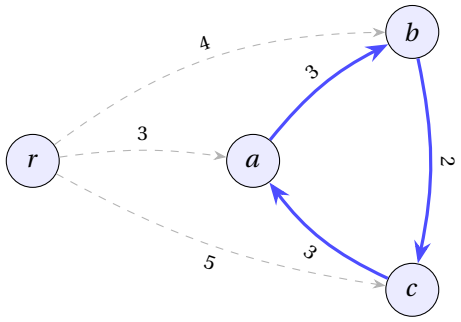


Figura 15 – Os arcos azuis são os da escolha gulosa.

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

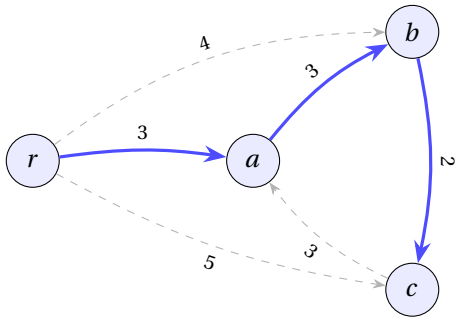


Figura 16 – Os arcos azuis são os da escolha gulosa.

3.2 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 3.1. *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. □

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

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 $v \in V \setminus \{r\}$,

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

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,

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

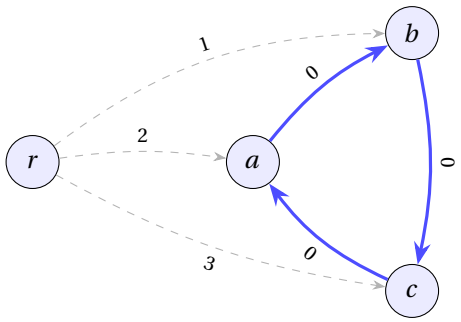


Figura 17 – Os arcos azuis são os arcos de D_0.

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 $t\,u$, 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.

[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.

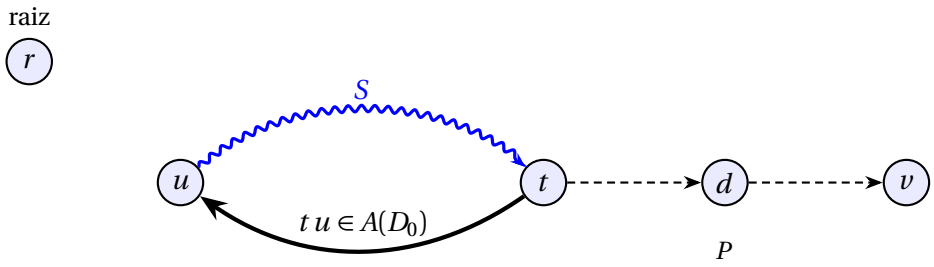


Figura 18 – O caminho simples maximal P inicia em u e termina em v. A porção S de P entre u e t é indicada pelo arco ondulado azul; o caminho $S \cdot u$ é um ciclo.

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 2.5). 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 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.

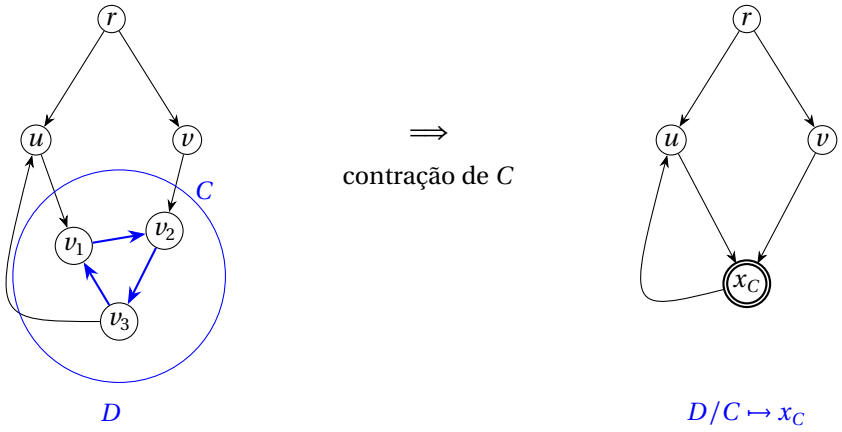
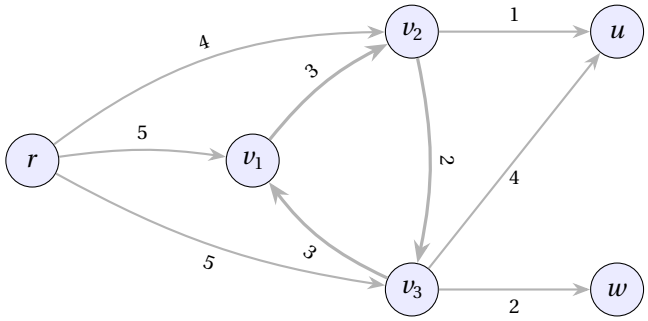


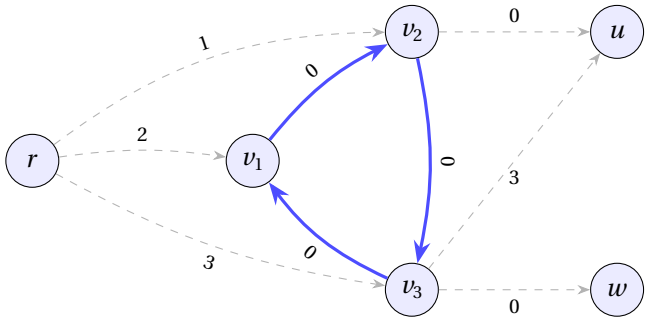
Figura 19 – Contração de um ciclo C em um vértice x_C.

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.

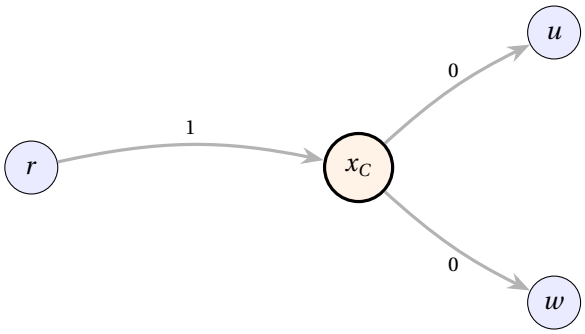
Agora vamos ilustrar um exemplo de como essa contração é feita e os custos são ajustados. Considere o digrafo a seguir.



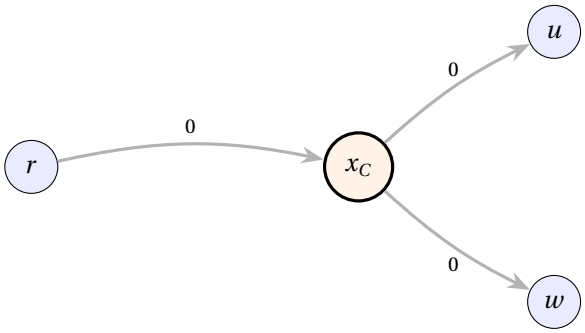
Após a redução dos custos, obtemos um ciclo $C := (v_1, v_2, v_3, v_1)$ cujos arcos têm custo reduzido igual a zero.



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.

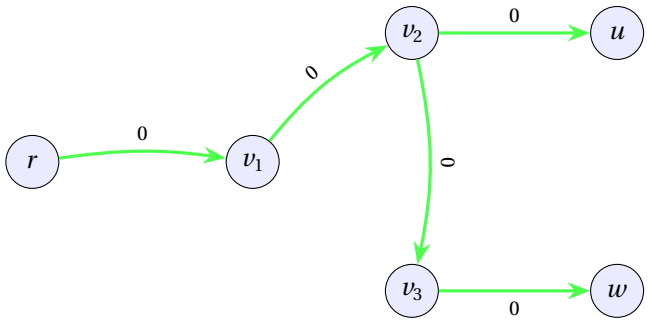


O subdigrafo gerador obtido a partir desse digrafo, usando apenas os arcos de custo reduzido igual a zero, possui uma r-arborescência. O algoritmo devolve essa r-arborescência, que agora deverá passar por um processo de expansão.

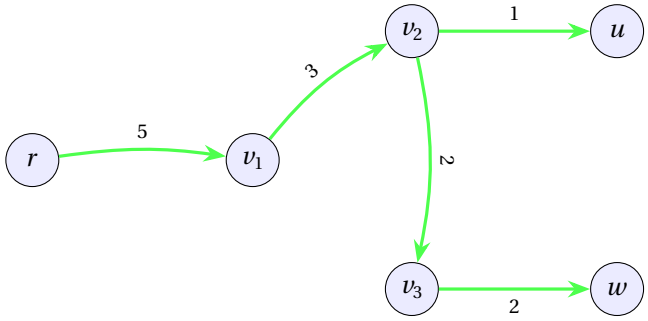
Reexpansão de arborescências

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 super-vértice x_C pelo ciclo C e adicionamos os arcos do ciclo que formam a arborescência dentro de C. Especificamente, o arco $u x_C$ 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 3.1, 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 3.1: 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 $O(|V||A|)$.

3.3 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`:

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

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`.
- `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)
```

3.3.1 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:

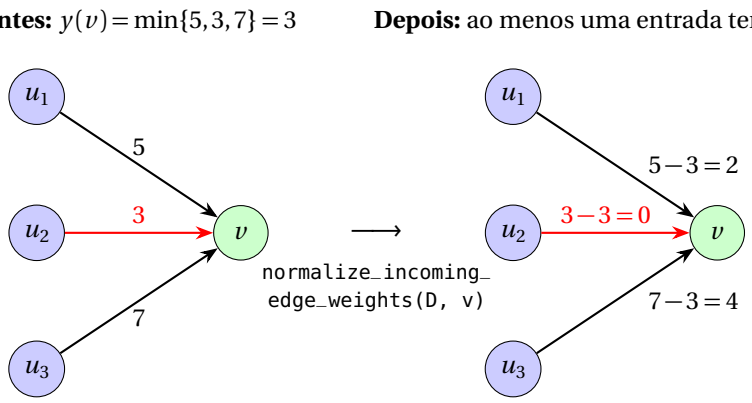


Figura 20 – À esquerda, vértice v com três arcos de entrada (pesos 5, 3 e 7). À direita, após aplicar `reduce_weights(D, v)`: o menor peso 3 é subtraído de todas as entradas, resultando em custos reduzidos 2, 0 e 4. O arco (u_2, v) (em vermelho) tem custo zero e será selecionado para D_0.

3.3.2 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.

3.3.3 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)
```

3.3.4 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`

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

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`.

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 $\texttt{D/C} \mapsto \texttt{label}$.
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
```

3.3.5 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
```

4 Algoritmo de András Frank

Neste capítulo apresentaremos o algoritmo de András Frank, que determina uma r-arborescência geradora de custo mínimo em um r-digrafo ponderado. O algoritmo é composto por duas fases. Na primeira, devida a Fulkerson (FULKERSON, 1974), constrói-se de maneira gulosa uma cobertura de r-conjuntos. Na segunda, devida a Frank (FRANK, 1979; FRANK, 2011), extrai-se dessa cobertura, também por meio de um procedimento guloso, uma r-arborescência geradora de custo mínimo. A apresentação é baseada em (LESTON-REY, 2025).

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

4.1 Preliminares

Começamos por definir a noção de uma coleção[1] laminar de conjuntos. Seja U um conjunto e denotemos por 2^U a coleção cujos elementos são os subconjuntos de U. Uma subcoleção $\mathscr{L} \subseteq 2^U$ é dita **laminar** se, para quaisquer $X, Y \in \mathscr{L}$, vale uma das alternativas:

$$X \subseteq Y \quad \text{ou} \quad Y \subseteq X \quad \text{ou} \quad X \cap Y = \emptyset.$$

Não é difícil verificar que, se U é um conjunto finito e $\emptyset \notin \mathscr{L} \subseteq 2^U$ é uma coleção laminar de subconjuntos de U, então

$$|\mathscr{L}| \leq 2|U| - 1.$$

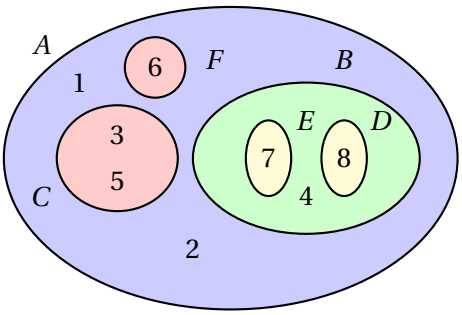


Figura 21 – A figura ilustra uma coleção laminar $\{A, B, C, D, E, F\}$ de conjuntos, onde $A := \{1,2,3,4,5,6,7,8\}$, $B := \{4,7,8\}$, $C := \{3,5\}$, $D := \{8\}$, $E := \{7\}$, $F := \{6\}$.

Seja $(D, w \geq 0, r)$ um r-digrafo ponderado. Para cada subconjunto $X \subseteq V$ e cada $F \subseteq A(D)$, seja

$$\varrho_F(X) := |F \cap \delta^-(X)|,$$

[1] Ao longo deste texto, usamos "coleção" como sinônimo de "conjunto".

isto é, $\varrho_F(X)$ é o número de arcos de F que *entram* em X. Para cada $k \in \mathbb{N}$, seja

$$[k] := \{\, i \in \mathbb{N} : i < k \,\}.$$

Assim, por exemplo, $[4] = \{0,1,2,3\}$. Uma sequência

$$\big((R_i, \lambda_i)\big)_{i\in[k)},$$

em que, para cada $i \in [k)$, o conjunto R_i é um r-conjunto[2] e λ_i é um real não negativo, é dita w-**disjunta** se

$$\sum_{i\in[k)} \lambda_i \,[a \in \delta^-(R_i)] \;\leq\; w(a)$$

para cada $a \in A(D)$, em que $[\mathsf{P}]$ denota a função indicadora da proposição P, isto é, $[\mathsf{P}] = 1$ se P é verdadeira e $[\mathsf{P}] = 0$ caso contrário. O número $\sum_{i\in[k)} \lambda_i$ é chamado de **valor** da sequência w-disjunta $\big((R_i, \lambda_i)\big)_{i\in[k)}$.

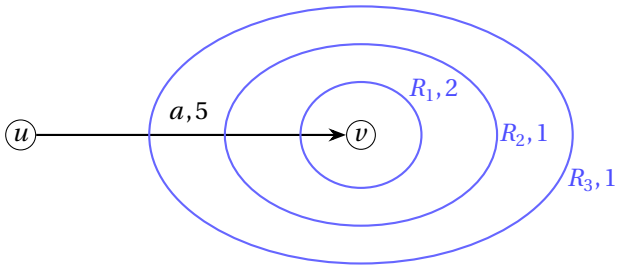


Figura 22 – Um arco a de peso 5 que entra nos r-conjuntos R_1, R_2 e R_3 com multiplidades 2, 1, e 1, respectivamente.

Podemos pensar nos valores λ_i como uma multiplicidade, no seguinte sentido. Para cada r-conjunto R_i, o valor λ_i representa a multiplicidade de R_i, isto é, o "número de cópias" de R_i que queremos incluir. Além disso, fixado um arco a, o número total de vezes que a é coberto pelos λ_i, isto é, o valor

$$\sum_{i\in[k)} \lambda_i \,[a \in \delta^-(R_i)],$$

não excede $w(a)$.

Suponha que F é uma cobertura de r-conjuntos – assim, para cada r-conjunto

[2] Recorde que um r-conjunto é um subconjunto não vazio de V que *não* contém r.

R, vale $\varrho_F(R) \geq 1$ – de D e que a sequência $\big((R_i, \lambda_i)\big)_{i\in[k)}$ é w-disjunta. Então

$$\begin{aligned} w(F) &= \sum_{a\in F} w(a) \\ &\geq \sum_{a\in F} \sum_{i\in[k)} \lambda_i [a \in \delta^-(R_i)] \\ &= \sum_{i\in[k)} \lambda_i \sum_{a\in F} [a \in \delta^-(R_i)] \\ &= \sum_{i\in[k)} \lambda_i \, \varrho_F(R_i) \\ &\geq \sum_{i\in[k)} \lambda_i . \end{aligned}$$

Na última desigualdade usamos que F é uma cobertura de r-conjuntos, de modo que, para cada $i \in [k)$, vale $\varrho_F(R_i) \geq 1$.

Note que $w(F) = \sum_{i\in[k)} \lambda_i$ se, e somente se, há igualdade em todas as desigualdades na cadeia acima. Logo,

$$w(F) = \sum_{i\in[k)} \lambda_i$$

se, e somente se, as seguintes **condições de otimalidade** valem:

$$\forall a \in F: \quad w(a) = \sum_{i\in[k)} \lambda_i [a \in \delta^-(R_i)], \tag{4.1a}$$

$$\forall i \in [k): \quad \varrho_F(R_i) = 1. \tag{4.1b}$$

Assim, se F é uma r-arborescência geradora e $\big((R_i, \lambda_i)\big)_{i\in[k)}$ é uma sequência w-disjunta que satisfaz as condições de otimalidade (4.1), então F é uma r-arborescência de peso mínimo e $\big((R_i, \lambda_i)\big)_{i\in[k)}$ é uma sequência w - disjunta de valor máximo.[3]

4.2 Fase 1 do algoritmo de Frank

Antes de iniciarmos a descrição da fase 1, vale destacar que, embora ela seja referida como a fase 1 do algoritmo de Frank, ela é, na verdade, essencialmente devida a Fulkerson (FULKERSON, 1974).

É conveniente introduzir a seguinte definição para simplificar a notação. Seja U um conjunto e seja $X \subseteq U$. Definimos $\mathbf{1}_X^U : U \to \{0,1\}$ pondo

$$\mathbf{1}_X^U(u) := \begin{cases} 1, & \text{se } u \in X, \\ 0, & \text{caso contrário,} \end{cases}$$

para cada $u \in U$. Quando o contexto permitir, escrevemos $\mathbf{1}_X$ no lugar de $\mathbf{1}_X^U$. Além disso, se $\lambda \in \mathbb{R}$ e $f : U \to \mathbb{R}$, então $\lambda f : U \to \mathbb{R}$ é definida pondo

$$(\lambda f)(u) := \lambda f(u)$$

[3] Isto é, $\sum_{i\in[k)} \lambda_i$ é máximo entre todas as sequências w-disjuntas.

para cada $u \in U$. Finalmente, como de costume, se $g : U \to \mathbb{R}$, então $f - g : U \to \mathbb{R}$ é definida pondo

$$(f-g)(u) := f(u) - g(u)$$

para cada $u \in U$.

Voltemos à descrição da primeira fase. A apresentação dessa fase, bem como dos demais algoritmos deste capítulo, segue de perto a exposição informal, porém rigorosa, de (FEOFILOFF, 2005).

Seja $(D, w \geq 0, r)$ um r-digrafo ponderado. A primeira fase do algoritmo constrói uma sequência

$$\big((f_i, R_i, \lambda_i)\big)_{i \in [k]}$$

que satisfaz as seguintes condições:

$$\{f_i : i \in [k]\} \text{ é uma cobertura de } r\text{-conjuntos de } D, \tag{4.2a}$$

$$\big((R_i, \lambda_i)\big)_{i \in [k]} \text{ é uma sequência } w\text{-disjunta}, \tag{4.2b}$$

$$\forall\, j \in [k]: \quad \sum_{i \in [k]} \lambda_i [f_j \in \delta^-(R_i)] = w(f_j). \tag{4.2c}$$

Note que (4.2c) é a condição (4.1a).

Cada iteração da primeira fase começa com uma função $c : A \to \mathbb{R}_+$ e uma sequência $\big((f_i, R_i, \lambda_i)\big)_{i \in [k]}$ tais que

$$c = w - \sum_{i \in [k]} \lambda_i \mathbf{1}_{\delta^-(R_i)} \tag{4.3a}$$

$$\forall\, i \in [k]: \quad f_i \text{ entra em } R_i, \tag{4.3b}$$

$$\forall\, i \in [k],\ \forall\, j \in [i]: \quad f_j \textit{ não} \text{ entra em } R_i, \tag{4.3c}$$

$$\forall\, i \in [k],\ \forall \varnothing \subset R \subset R_i,\ \exists\, h \in [i]: \quad f_h \text{ entra em } R, \tag{4.3d}$$

$$\big((R_i, \lambda_i)\big)_{i \in [k]} \text{ é uma sequência } w\text{-disjunta}, \tag{4.3e}$$

$$\{R_i : i \in [k]\} \text{ é uma coleção laminar de } r\text{-conjuntos}, \tag{4.3f}$$

$$\forall\, i \in [k]: \quad c(f_i) = 0 \tag{4.3g}$$

Note que a condição (4.3g) é equivalente a

$$\forall\, i \in [k]: \quad \sum_{j \in [k]} \lambda_j [f_i \in \delta^-(R_j)] = w(f_i).$$

A primeira iteração começa com $c = w$ e com a sequência vazia. Cada iteração consiste no seguinte. Suponha que $\sigma := \big((f_i, R_i, \lambda_i)\big)_{i \in [k]}$ satisfaz (4.3). Se $F := \{f_i : i \in [k]\}$ é uma cobertura de r-conjuntos de D, então o algoritmo pára e devolve σ. Suponha que esse não seja o caso. O algoritmo então seleciona um r-conjunto *minimal* R_k que não é

coberto por F. Como D possui ao menos uma r-arborescência, então existe ao menos um arco que entra em R_k. A próxima iteração começa com $\sigma \cdot (f_k, R_k, \lambda_k)$ no lugar de σ e $c - \lambda_k \mathbf{1}_{\delta^-(R_k)}$ no lugar de c, em que

- $\lambda_k := \min\Big\{ c(a) a \in \delta^-(R_k) \Big\}$, e
- f_k é um arco em $\delta^-(R_k)$ tal que $c(f_k) = \lambda_k$.

Precisamos verificar que $\sigma \cdot (f_k, R_k, \lambda_k)$ satisfaz (4.3). A única propriedade em (4.3) que não segue diretamente das escolhas de f_k, R_k e λ_k é a laminaridade da coleção $\{R_i : i \in [k+1)\}$. Suponha que esse não seja o caso. Então existe $i \in [k)$ tal que

$$R_i \cap R_k \neq \emptyset, \quad R_i \setminus R_k \neq \emptyset, \quad R_k \setminus R_i \neq \emptyset.$$

Em virtude de (4.3d), existe $h \in [i)$ tal que $f_h =: uv$ entra em $R_i \cap R_k$. Assim, $u \notin R_i \cap R_k$ e $v \in R_i \cap R_k$. Ora, $u \notin R_i \cap R_k$ implica que $u \in V \setminus R_i$ ou $u \in V \setminus R_k$. Mas $u \in V \setminus R_i$ implica que uv entra em R_i, o que contraria (4.3c). Por outro lado, $u \in V \setminus R_k$ implica que uv entra em R_k, o que novamente é uma contradição, pois F não entra em R_k. Logo, a coleção $\{R_i : i \in [k+1)\}$ é laminar.

Para completar a descrição da primeira fase, basta mostrar como encontrar um r-conjunto minimal que não é coberto por F. Para isso, considere o digrafo $D_0 := (V, F)$. Como F não é uma cobertura de r-conjuntos, D_0 não contém uma r-arborescência. Logo, pela Proposição 2.2, existe um r-conjunto $X \subseteq V$ tal que nenhum arco de F entra em X. Logo, existe pelo menos uma fonte, digamos S, em $\mathscr{C}(D_0)$. Proposição 4.1, S é um r-conjunto minimal não coberto por F.

Proposição 4.1. *Seja H um digrafo e seja r um vértice de H. Para toda fonte S de $\mathscr{C}(H)$, se $r \notin S$, então S é um r-conjunto minimal não coberto por $A(H)$.*

Prova. Suponha que S seja uma fonte de $\mathscr{C}(H)$ que satisfaz $r \notin S$. Como S é uma fonte de $\mathscr{C}(H)$, nenhum arco de H entra em S. Além disso, $r \notin S$, donde S é um r-conjunto não coberto por $A(H)$. Resta mostrar que S é minimal.

Ora, S é uma componente fortemente conexa de H, de modo que $H[S]$ é fortemente conexo. Seja agora $\emptyset \neq R \subset S$ um subconjunto próprio e não vazio de S. Como $H[S]$ é fortemente conexo, existe ao menos um arco de $H[S]$ que entra em R. Como $A(H[S]) \subseteq A(H)$, esse arco pertence a $A(H)$ e entra em R. Logo, $A(H)$ entra em todo subconjunto próprio e não vazio de S.

Portanto, S não é coberto por $A(H)$, mas todo subconjunto próprio não vazio de S é coberto por $A(H)$, isto é, S é um r-conjunto minimal não coberto por $A(H)$. □

Para sumarizar o processo algoritmo recém descrito, eis um pseudo-código da fase 1 do algoritmo de Frank.

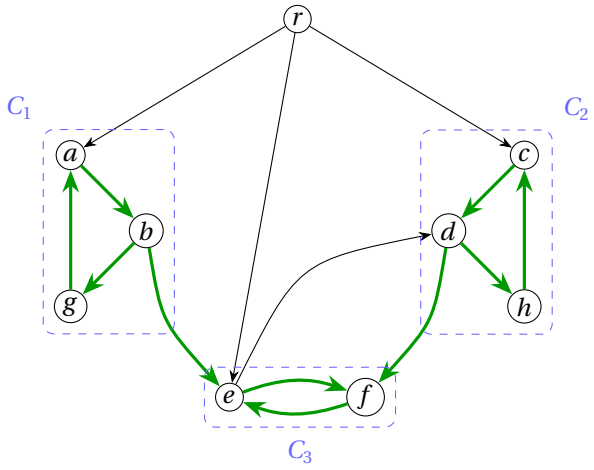


Figura 23 – Digrafo D com raiz r: arcos pretos em $A(D)\setminus F$ e arcos verdes em F. As caixas tracejadas destacam os componentes fortemente conexos de $D[F]$; C_1 e C_2 são fontes na condensação.

Fase 1 do algoritmo de Frank (algoritmo de Fulkerson)

Recebe um r-digrafo ponderado (D,w,r) e devolve uma sequência σ que satisfaz (4.2).

```
1  def phase1(D, w, r):
2      c,σ := w,ϵ
3      loop:
4          (f_i,R_i,λ_i)_{i∈[k)} := σ
5          F := {f_i | i ∈ [k)}
6          D_0 := (V,F)
7          if 𝒞(D_0) não possui fontes: return σ
8          seja R_k uma fonte em 𝒞(D_0)
9          λ_k := min{c(a) : a ∈ δ⁻(R_k)}
10         seja f_k ∈ δ⁻(R_k) tal que c(f_k) = λ_k
11         σ := σ·(f_k,R_k,λ_k)
12         c := c − λ_k 1_{δ⁻(R_k)}
```

Complexidade

A cada iteração, o algoritmo determina um componente fonte de $\mathscr{C}(D_0)$, o que pode ser realizado em tempo $O(|A|)$, usando, por exemplo, o algortimo de Kosaraju (CORMEN et al., 2009). O conjunto obtido da sequência de r-conjunto devolvida pelo algoritmo é laminar (e a sequência não possui repetições). Logo, seu tamanho é limitado por $2|V|-1$. Consequentemente, é possível implementar a fase 1 em tempo $O(|V||A|)$.

4.3 Implementação da fase 1

A função `phase1` recebe `D: DiGraph`, ponderado pelo atributo `"w"` associado a cada arco, e `r: int`, que é um vértice de `D`. Além disso, `D` deve possuir uma `r`-arborescência. A função devolve uma lista `sigma` de triplas que corresponde à sequência λ satisfazendo (4.2).

Na linha 3, é definido `D_zero`, que, durante a execução da função, armazena os arcos da lista `sigma` — mais precisamente, o primeiro componente de cada tripla em `sigma`. Assim, em cada iteração, temos `D_zero` $= (V, F)$, onde $F = \{f_i\}$ é a família de arcos já selecionados.

A função usa `condensation`, da biblioteca `NetworkX`, que recebe um `DiGraph` e devolve um `DiGraph` que é a sua condensação. Assim, após a execução da linha 5, `C` é a condensação de `D_zero`. Na linha 6, as fontes de `C`, isto é, os vértices nos quais nenhum arco entra, são coletadas na lista `sources`.

Pela hipótese de que (D, w, r) admite uma r-arborescência, sabemos que toda fonte de $\mathscr{C}(D_0)$ que não contém r corresponde a um r-conjunto minimal não coberto. Logo, quando `sources` contém apenas uma fonte, todos os r-conjuntos já estão cobertos e a função termina.

Caso contrário, para cada fonte de `C`, os vértices do componente forte correspondente em `D_zero` são coletados em `X` (linha 9); um tal `X` é de interesse desde que `r` não pertença a `X`. Nesse caso, os arcos de `D_copy` que entram em `X` são armazenados em `arcs` (linha 12). O peso de um arco de custo mínimo em `arcs` é determinado na linha 15 e armazenado em `min_weight`.

A função `update_weights` decrementa de `min_weight` o peso de cada arco em `arcs` e devolve um arco de menor peso em `arcs`, que é armazenado em `a` (linha 16). O arco `a` é adicionado ao digrafo `D_zero` (linha 17). Finalmente, na linha 18, a tripla `(a, X, min_weight)` é adicionada à lista `sigma`.

Fase 1 do algoritmo de Frank

Entrada: `D: DiGraph`, `r: int`
Pré-condição: `D` é ponderado, `r` é um vértice de `D`, e `D` possui uma `r`-arborescência.
Saída: uma lista `sigma` que corresponde à uma sequência que satisfaz 4.2.

```
1 def phase1(D: nx.DiGraph, r: int):
2     D_copy = D.copy()
3     sigma = []
4     D_zero = nx.DiGraph(); D_zero.add_nodes_from(D_copy.nodes())
5     while True:
6         C = nx.condensation(D_zero)
7         sources = [x for x in C.nodes() if C.in_degree(x) == 0]
8         if len(sources) == 1: break
```

```
        for s in sources:
            X = C.nodes[s]["members"]
            if r in X:
                continue
            arcs = [(u, v, data)
                    for u, v, data in D_copy.edges(data=True)
                    if u not in X and v in X]
            min_weight = min(data["w"] for _, _, data in arcs)
            a = update_weights(D_copy, arcs, min_weight)
            D_zero.add_edge(a[0], a[1])
            sigma.append((a, X, min_weight))
    return sigma
```

A função `update_weights` é muito simples e segue abaixo.

Fase 1 do algoritmo de Frank

Entrada: `D: DiGraph, arcs: list[tuple[int, int, dict]], min_weight: float`.

Pré-condição: `D` é ponderado, `arcs` é uma lista de arcos, `min_weight` é o peso de um arco de menor peso em `arcs`.

Pós-condição: Decrementa de `min_weight` o peso de cada arco em `arcs`, modificando esse atributo em `D`.

```
def update_weights(D: nx.DiGraph,
                   arcs: list[tuple[int, int, dict]],
                   min_weight: float):
    for u, v, _ in arcs:
        D[u][v]["w"] -= min_weight
        if D[u][v]["w"] == 0:
            a = (u, v)
    return a
```

4.4 Fase 2 do Algoritmo de Frank

A fase 2 do algoritmo de Frank recebe um r-digrafo ponderado $(D, w \geq 0, r)$ e uma sequência $(f_i)_{i\in[k)}$ extraída da sequência $\big((f_i, R_i, \lambda_i)\big)_{i\in[k)}$, obtida na fase 1 do algoritmo de Frank, a qual satisfaz (4.2). A fase 2 devolve um subconjunto $J \subseteq \{f_i : i \in [k)\}$ que é uma r-arborescência geradora de D, satisfazendo (4.1a), ou seja,[4]

$$\forall i \in [k): \quad \varrho_J(R_i) = 1.$$

A combinação da sequência $\big((R_i, \lambda_i)\big)_{i\in[k)}$ com esse conjunto J prova que J é uma r-arborescência geradora de peso mínimo de (D, w).

Cada iteração da fase 2 começa com um conjunto J de arcos tal que J é uma r-arborescência.[5] A primeira iteração começa com $J := \varnothing$. Cada iteração consiste no

[4] Se preferir em palavras, isto quer dizer que, em cada r-conjunto R_i, entra exatamente um arco de J.

[5] Assuma, por simplicidade, que quando $J = \varnothing$ o digrafo J é uma r-arborescência.

seguinte. Se J é uma r-arborescência geradora, então o algoritmo pára e devolve J. Suponha que esse não seja o caso. O algoritmo então considera a sequência

$$(f_i)_{i\in[k)}$$

e seleciona o menor $i \in [k)$ tal que f_i é um arco que sai de $V(J)$. Note que tal arco deve existir, pois $\{f_i : i \in [k)\}$ é uma cobertura de r-conjuntos. A próxima iteração começa com $J \cup \{f_i\}$ no lugar de J.

Afirmamos que no início de cada iteração, o conjunto J satisfaz

$$\forall i \in [k) : \varrho_J(R_i) \leq 1.$$

Isso é óbvio no início da primeira iteração. Considere uma iteração arbitrária e suponha que J não é uma r-arborescência. Vamos mostrar que $J \cup \{f_i\}$ satisfaz

$$\forall i \in [k) : \varrho_{J\cup\{f_i\}}(R_i) \leq 1.$$

Por construção, dentre os arcos de $(f_i)_{i\in[k)}$ que saem de $V(J)$, o arco f_i é o de menor índice. Além disso, f_i entra em R_i. Suponha, por um momento, que algum arco de J entre em R_i; então $V(J) \cap R_i \neq \emptyset$, donde $R_i \setminus V(J) \subset R_i$. Além disso, como f_i sai de $V(J)$ e entra em R_i, concluímos que $R_i \setminus V(J) \neq \emptyset$. Logo, $\emptyset \subset R_i \setminus V(J) \subset R_i$.

Sabemos que, para cada $\emptyset \subset R \subset R_i$, existe um arco de $F := \{f_j : j \in [k)\}$ que entra em R e cujo índice é menor que i. Em particular, tomando $R := R_i \setminus V(J)$, existe $f_k =: uv \in F$ tal que f_k entra em $R_i \setminus V(J)$ e, portanto, $k < i$ e $v \in R_i \setminus V(J)$. Como f_k não entra em R_i e a ponta final v pertence a R_i, segue que $u \in R_i$.

Ora, $R_i = (R_i \cap V(J)) \cup (R_i \setminus V(J))$, e, como $u \in R_i$ e $u \notin R_i \setminus V(J)$, obtemos $u \in V(J)$. Logo, f_k é um arco de F que sai de $V(J)$, o que contraria a escolha de i, pois $k < i$. Essa contradição mostra que $J \cup \{f_i\}$ satisfaz

$$\forall i \in [k) : \varrho_{J\cup\{f_i\}}(R_i) \leq 1,$$

como queríamos.

O pseudo-código do algoritmo da fase 2, em uma versão ingênua, é muito simples.

Fase 2: Construção da r-arborescência geradora

Recebe um digrafo D, um vértice r de D e uma sequência $(f_i)_{i\in[k)}$ de arcos tal que $\{f_i : i \in [k)\}$ é uma cobertura dos r-conjuntos de D, e devolve um subdigrafo que é uma r-arborescência geradora de D.

```
1 def phase2(D, r, (f_i)_{i∈[k)}):
2     U, J := {r}, ∅
```

```
    for t := 1 to |V(D)| − 1:
        for i ∈ [k]:
            (u_i, v_i) := f_i
            if u_i ∈ U e v_i ∉ U:
                U, J := U ∪ {v_i}, J ∪ {f_i}
                break
    return (U, J)
```

Complexidade

Observe que o algoritmo realiza $O(|V(D)|)$ iterações e que cada iteração pode ser implementada de modo a consumir tempo em $O(|F|)$. Logo, o consumo de tempo está em $O(|V||F|)$.

4.5 Implementação da fase 2

A implemantação da fase 2 é muito simples e segue bem de perto a versão do pseudo-código.

Fase 2: Construção da arborescência

Entrada: `D: nx.DiGraph`, `r: int`, `F: list[tuple[int, int]]`
Pré-condição: `F` é uma cobertura de `r`-conjuntos de `D`.
Saída: Um `DiGraph` que é uma `r`-arborescência geradora de `D` cujos arcos estão contidos em `F`.

```python
def phase2(D: nx.DiGraph, r: int, F: list[tuple[int, int]]):
    Arb = nx.DiGraph()
    Arb.add_node(r)
    n = len(D.nodes())
    for _ in range(n - 1):
        for u, v in F:
            if u in Arb.nodes() and v not in Arb.nodes():
                edge_data = D.get_edge_data(u, v)
                Arb.add_edge(u, v, **edge_data)
                break
    return Arb
```

4.6 Versão alternativa da fase 2

Vamos agora exibir uma versão alternativa e mais eficiente da fase 2, como sugerido em (REY, 2025), que fornece um algoritmo que pode ser implementado em

tempo $O(|A|+|V|\log|V|)$. No contexto do nosso problema isso se reduz a $O(|V|\lg|V|)$. A ideia é simples e nada mais é do que uma variante do algoritmo de Dijkstra (CORMEN et al., 2009).

A versão alternativa recebe um r-digrafo ponderado $(D, w \geq 0, r)$ e uma sequência $(f_i)_{i\in[k)}$ extraída da sequência $\big((f_i, R_i, \lambda_i)\big)_{i\in[k)}$, obtida na fase 1 do algoritmo de Frank, a qual satisfaz (4.2). Ela devolve um subconjunto $J \subseteq \{f_i : i \in [k)\}$ que é uma r-arborescência geradora de D e satisfaz (4.1a).

Forme o subdigrafo gerador

$$H := \big(V, \{f_i : i \in [k)\}\big)$$

de D e a função *prioridade*

$$p : \{f_i : i \in [k)\} \to \mathbb{N} \quad \text{tal que} \quad p(f_i) := i \text{ para cada } i \in [k).$$

Cada iteração do algoritmo começa com um subconjunto J de arcos de H e com um subconjunto Q também de arcos de H. A primeira iteração começa com

$$J := \varnothing \quad \text{e} \quad Q := \delta^+_H(r).$$

Cada iteração consiste no seguinte.

- Se $Q = \varnothing$, então o algoritmo pára e devolve J.
- Caso contrário, o algoritmo seleciona um arco $(u, v) \in Q$ tal que $p(u, v)$ é mínimo.
  - Se $v \in V(J)$, então o algoritmo inicia uma nova iteração com J e $Q \setminus \{(u, v)\}$ nos papéis de J e Q, respectivamente.
  - Se $v \notin V(J)$, então o algoritmo inicia uma nova iteração com

    $$J' := J \cup \{(u, v)\} \quad \text{e} \quad Q' := \big(Q \setminus \{(u, v)\}\big) \cup \delta^+_H(v)$$

    nos papéis de J e Q, respectivamente.

A implementação dessa versão segue abaixo.

Versão alternativa da fase 2

Entrada: `D: nx.DiGraph, r: int, F: list[tuple[int, int]]`
Pré-condição: `F` é uma cobertura de `r`-conjuntos de `D`.
Saída: Um `DiGraph` que é uma `r`-arborescência geradora de `D`

```
1 def phase2_v2(D, r, F):
2     Arb = nx.DiGraph()
```

```
3     for i, (u, v) in enumerate(F):
4         Arb.add_edge(u, v, w=i)
5     V = {r}
6     q = []
7     for u, v, data in Arb.out_edges(r, data=True):
8         heapq.heappush(q, (data["w"], u, v))
9     J = nx.DiGraph()
10    while q:
11        _, u, v = heapq.heappop(q)
12        if v in V:
13            continue
14        J.add_edge(u, v, w=D[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 J
```

4.7 O algoritmo de Frank

O algoritmo de Frank é obtido compondo-se as duas fases. Há duas versões à disposição. Para obter a segunda, basta trocar a chamada da função `phase2_v2` pela chamada da função `phase2`.

Fase 2: Construção da r-arborescência geradora

Recebe um r-digrafo ponderado (D, w, r), e devolve um par $\left(J, (R_i, \lambda_i)_{i\in[k)}\right)$ satisfazendo as condições de otimalidade (4.1).

```
1 def frankv1(D, w, r):
2     (f_i, R_i, λ_i)_{i∈[k)} := phase1(D, w, r)
3     J := phase2_v2(D, r, {f_i : i ∈ [k)})
4     return (J, (R_i, λ_i)_{i∈[k)})
```

Complexidade

O algoritmo `phase1` tem consumo de tempo em $O(|V||A|)$. A sequência devolvida tem comprimento $k \leq 2|V|-1$. Assim, o consumo de tempo de `phase2_v2` está em $O(|V|\lg|V|)$. Logo, o consumo de tempo de `frankv1` está em $O(|V|(\lg|V|+|A|))$.

5 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.

5.1 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 12,40s), significativamente superior ao tempo mediano de Chu–Liu/Edmonds (0,25s, média 0,58s). 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 0,98s para versão 1 do algoritmo e 0,016s para a versão 2.

As Figuras 24–29 apresentam os resultados experimentais.

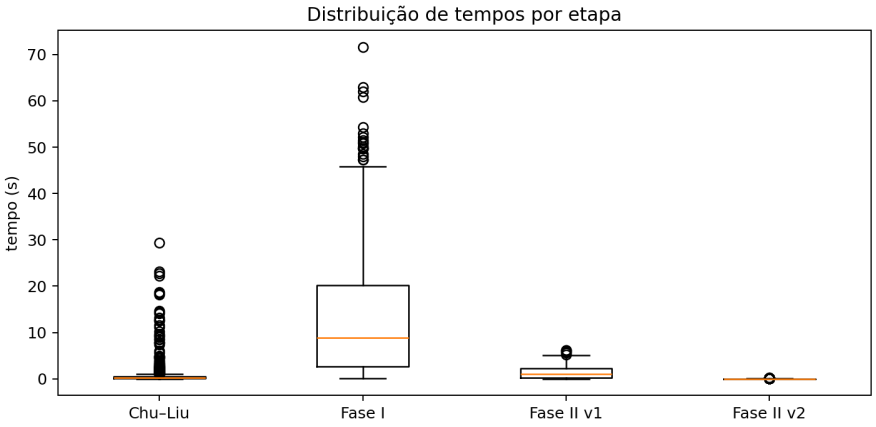


Figura 24 – Distribuição de tempos: Fase I apresenta maior mediana (8,93s) e variabilidade que Chu–Liu/Edmonds (0,25s).

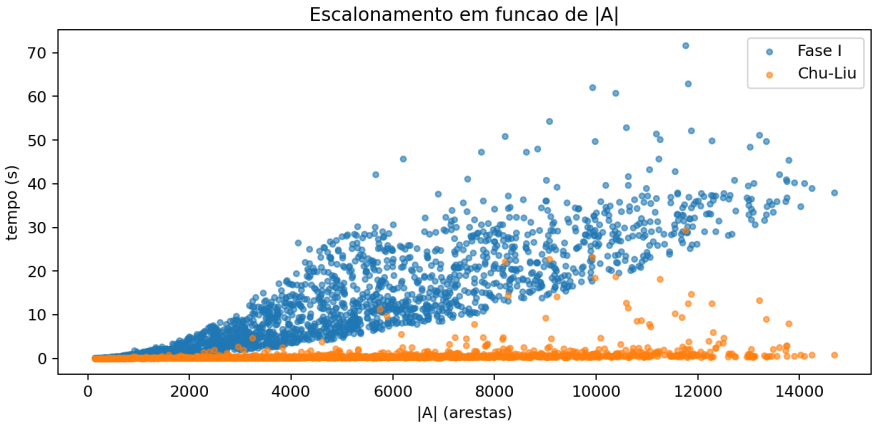


Figura 25 – Escalonamento temporal em função de $|A|$: crescimento aproximadamente linear.

A comparação entre as duas variantes da Fase II revela ganho expressivo com o uso de *heap*. A Figura 26 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 0,98s, a versão v2 (heap) reduz para 0,016s. 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.

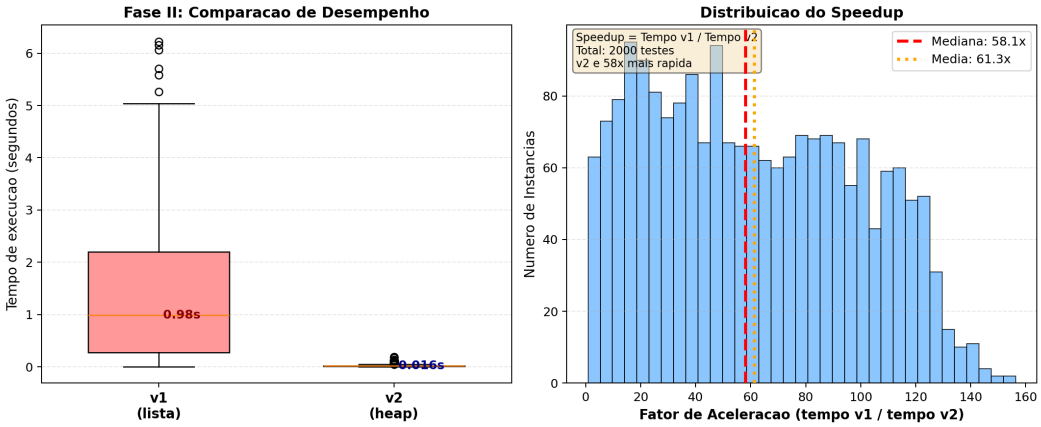


Figura 26 – Comparação de desempenho entre implementações da Fase II. À esquerda, o gráfico de boxplot mostra a distribuição de tempo da v1 do algoritmo com mediana de 0,98s enquanto v2 reduz para 0,016s. À direita, o histograma do fator de aceleração (*speedup*) mostra a distribuição concentrada entre 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 27 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.

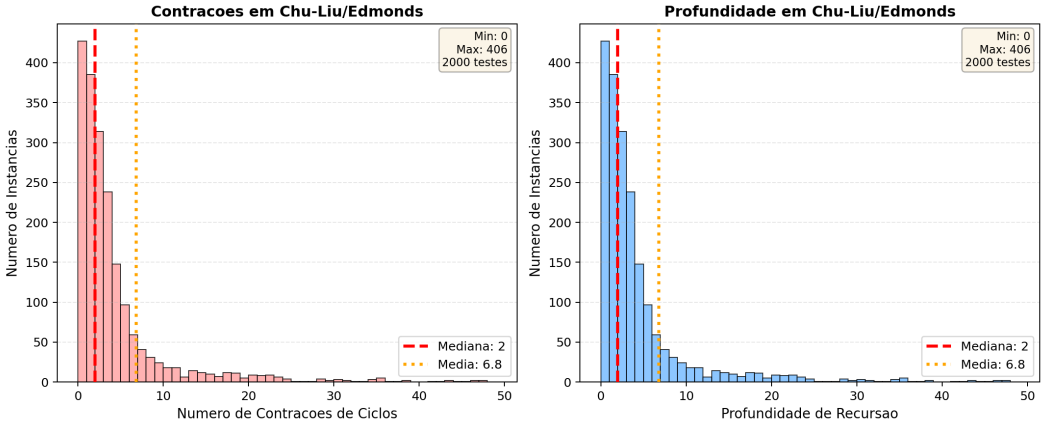


Figura 27 – Métricas estruturais de Chu–Liu/Edmonds. À esquerda, o histograma vermelho mostra que o número de contrações (eixo horizontal) é concentrado em valores baixos — a maioria das 2000 instâncias (eixo vertical) requer menos de 20 contrações, com mediana 2 (linha tracejada) e média 6,82 (linha 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.

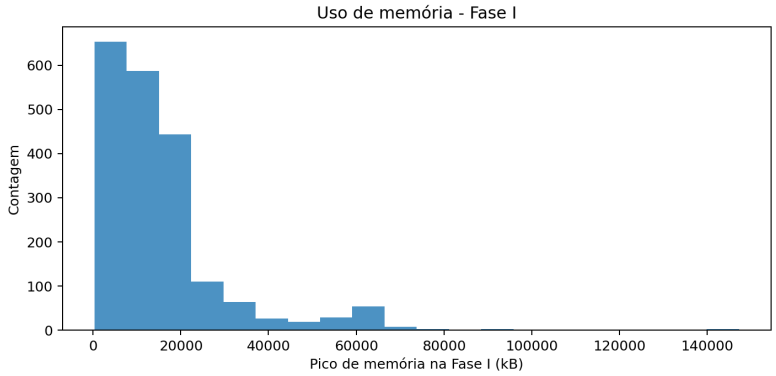


Figura 28 – Pico de memória na Fase I: mediana 11,5 MB.

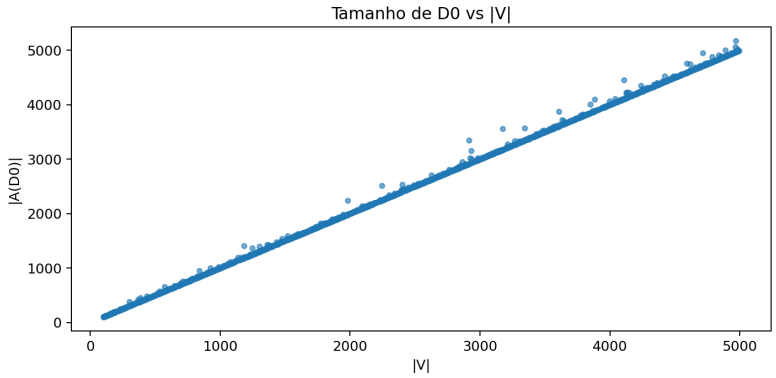


Figura 29 – Tamanho de D_0 versus $|V|$: relação linear confirma $|A_0| = O(|V|)$.

5.2 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 0,25s, média 0,58s), enquanto o método de Frank apresenta tempo adicional significativo na Fase I (mediana 8,93s, média 12,40s) 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.

6 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.

6.0.1 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 comple-

xas, é frequente que estudantes compreendam os passos operacionais sem internalizar a estrutura teórica que garante correção e otimalidade.

6.0.2 Lidando com grafos e digrafos

Na prática, o principal obstáculo para o ensino de digrafos 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.

6.0.3 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 digrafos, 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.

6.1 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 digrafos, 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 digrafos, 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 1 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 1 – 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.

7 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.

7.1 Descrição da aplicação

A aplicação *web* foi concebida como uma ferramenta educacional interativa para auxiliar na compreensão dos algoritmos de Chu–Liu–Edmonds e András 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.

7.1.1 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.

7.1.2 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.

7.1.3 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.

7.2 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 → filter → 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 2 – Síntese dos princípios de interação humano-computador aplicados e sua realização concreta na ferramenta interativa.

7.3 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.

7.3.1 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.

7.3.2 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 30 – 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 7.1.1 Utiliza `Cytoscape.js` para visualização interativa e `PyScript` para lógica algorítmica. Abaixo, apresentamos uma captura de tela da página.

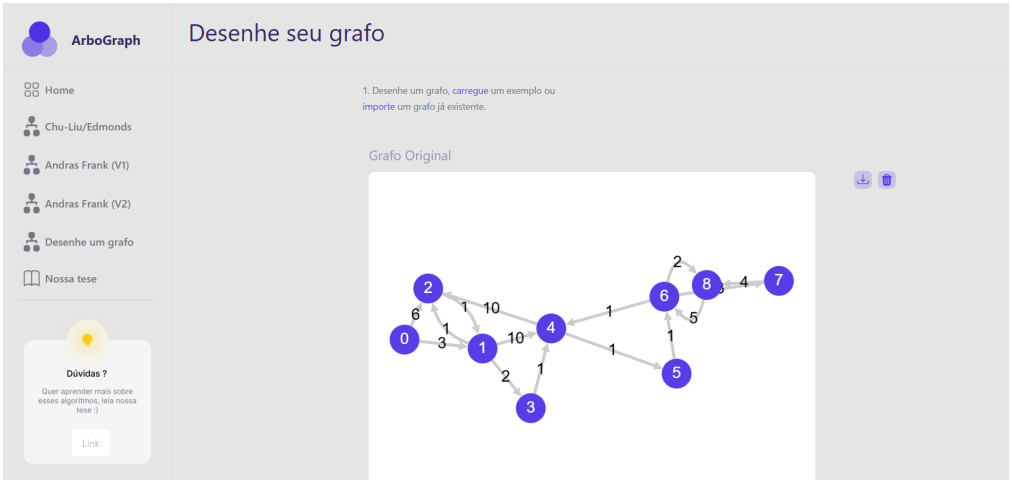


Figura 31 – Captura de tela de `draw_graph.html`: editor livre de grafos.

`Tese`:

Essa página compoem o módulo 3 apresentado na seção 7.1.1, 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.

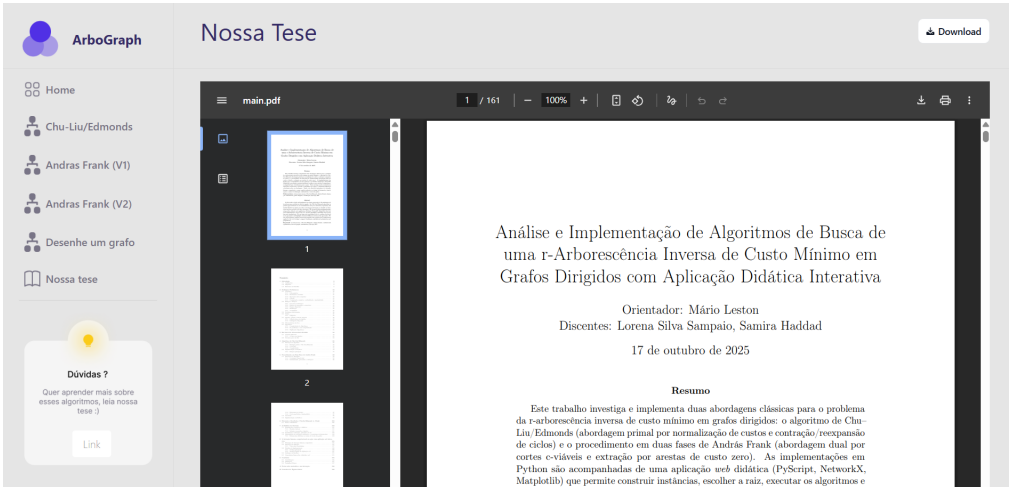


Figura 32 – Captura de tela de `tese.html`: visão geral com resumo e integrantes.

`Chu-liu-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 7.1.1. Abaixo, apresentamos uma captura de tela da página.

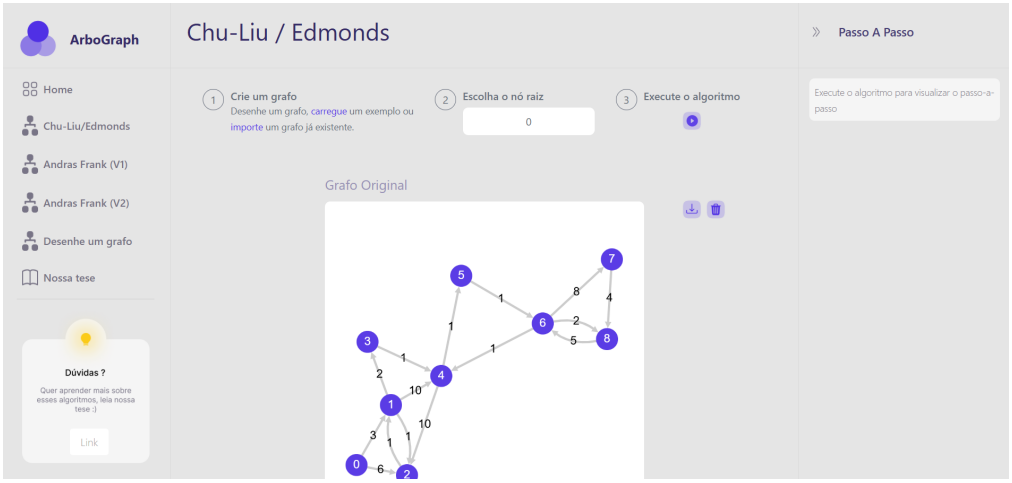


Figura 33 – Captura de tela de `chuliu.html`: criação de grafo, seleção de raiz e execução do algoritmo.

A figura a seguir destaca a tripartição funcional da página: navegação lateral,

conteúdo interativo central e guia de passos à direita.

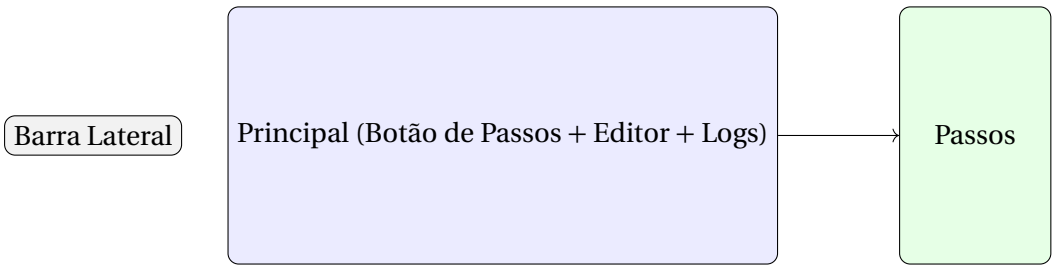


Figura 34 – Tripartição funcional (navegação, conteúdo interativo, guia de passos). A presença do passo a passo auxilia na compreensão sequencial do algoritmo.

7.3.3 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 7.1.1 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.

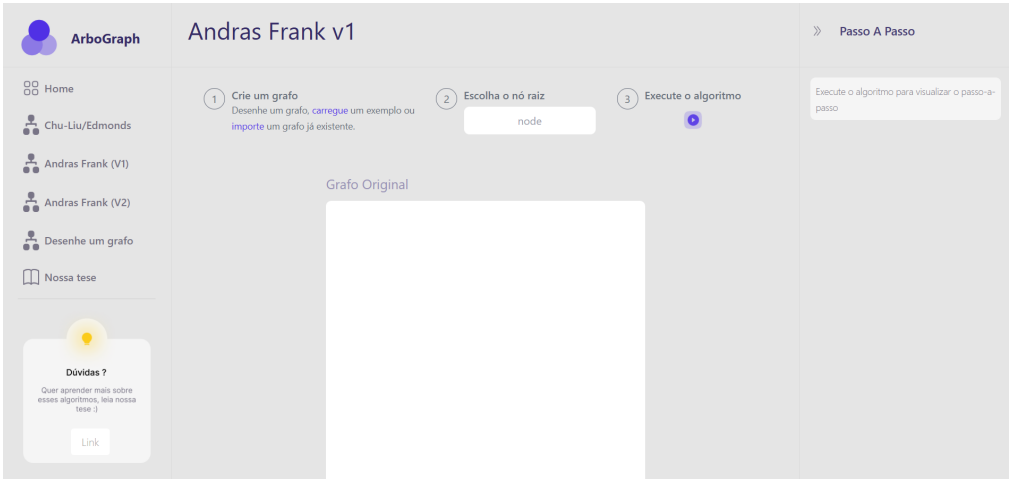


Figura 35 – Captura de tela de `andrasfrank_v1.html`: interface para o procedimento em duas fases, a tela da página `andrasfrank_v2.html` tem aparência similar.

Assim como na página do algoritmo de Chu–Liu/Edmonds (fig. 34) a página dedicadas às implementações do algoritmo do Andras Frank também utilização o padrão de tripartição funcional para manter consistência cognitiva entre páginas.

7.3.4 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.

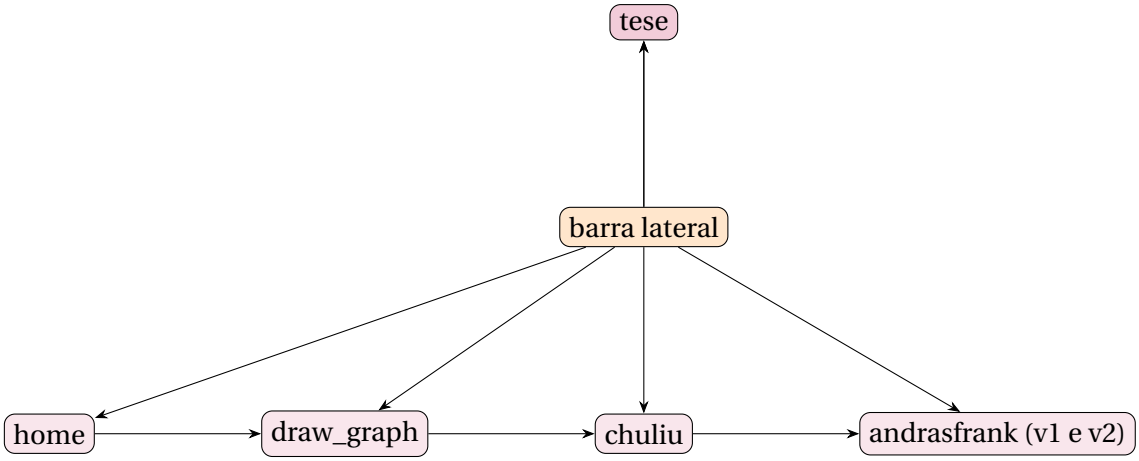


Figura 36 – A `barra lateral` injeta navegação consistente; páginas de algoritmo formam trilha exploratória.

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.

7.4 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 den-

sas. 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.

8 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.

8.1 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.

8.2 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.

8.3 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 3 vezes nas páginas 13, 21 e 50.

CORMEN, T. H. et al. *Introduction to Algorithms*. 3rd. ed. [S.l.]: MIT Press, 2009. Citado 2 vezes nas páginas 43 e 48.

EDMONDS, J. Optimum branchings. *Journal of Research of the National Bureau of Standards*, v. 71B, p. 233–240, 1967. Citado 3 vezes nas páginas 13, 21 e 50.

FEOFILOFF, P. *Algoritmos de Programação Linear: Programação Linear Concreta*. [s.n.], 2005. Notas de curso, Instituto de Matemática e Estatística, Universidade de São Paulo. Disponível em: <http://www.ime.usp.br/~pf/prog-lin>. Citado na página 41.

FRANK, A. Kernel systems of directed graphs. *Acta Scientiarum Mathematicarum (Szeged)*, v. 41, n. 1-2, p. 63–76, 1979. Citado 2 vezes nas páginas 13 e 38.

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 50.

FRANK, A. *Connections in Combinatorial Optimization*. Oxford: Oxford University Press, 2011. (Oxford Lecture Series in Mathematics and its Applications). ISBN 978-0-19-920527-1. Citado 2 vezes nas páginas 13 e 38.

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 50.

FULKERSON, D. R. Packing rooted directed cuts in a weighted directed graph. *Mathematical Programming*, v. 6, p. 1–13, 1974. Citado 2 vezes nas páginas 38 e 40.

HALIM, S. et al. *VisuAlgo*. <https://visualgo.net/>. Acesso didático a visualizações interativas de algoritmos, acessado em 2025. Citado na página 59.

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 57, 58 e 59.

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 57 e 58.

LESTON-REY, M. Notas de aula de métodos de otimização. 2025. Citado na página 38.

MAYER, R. E. *Multimedia Learning*. 2nd. ed. [S.l.]: Cambridge University Press, 2009. Citado 4 vezes nas páginas 55, 57, 58 e 64.

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 57, 58, 59 e 64.

NIELSEN, J. *Usability Engineering*. [S.l.]: Morgan Kaufmann, 1994. Fonte das heurísticas clássicas de usabilidade. ISBN 978-0125184069. Citado na página 64.

PAIVIO, A. *Mental Representations: A Dual Coding Approach*. [S.l.]: Oxford University Press, 1990. Citado 2 vezes nas páginas 55 e 57.

REY, P. S. L. Comunicação oral. 2025. Citado na página 47.

ROGERS, Y.; SHARP, H.; PREECE, J. *Interaction Design: Beyond Human-Computer Interaction*. 3rd. ed. [S.l.]: Wiley, 2011. ISBN 978-0470665763. Citado na página 64.

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 58 e 64.

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 64.

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 55, 58 e 64.

TALL, D. *Advanced Mathematical Thinking*. [S.l.]: Kluwer Academic Publishers, 1991. Citado 2 vezes nas páginas 55 e 57.

WARE, C. *Information Visualization: Perception for Design*. 3rd. ed. [S.l.]: Morgan Kaufmann, 2012. Citado 2 vezes nas páginas 57 e 58.

Anexos

ANEXO A – Anexo A

Conteúdo do anexo A.