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

Orientador: Mário Leston
Discentes: Lorena Silva Sampaio, Samira Haddad

17 de outubro de 2025

Resumo

Este trabalho investiga e implementa duas abordagens clássicas para o problema da r-arborescência inversa de custo mínimo em grafos dirigidos: o algoritmo de Chu–Liu/Edmonds (abordagem primal por normalização de custos e contração/reexpansão de ciclos) e o procedimento em duas fases de András Frank (abordagem dual por cortes c-viáveis e extração por arestas de custo zero). As implementações em Python são acompanhadas de uma aplicação *web* didática (PyScript, NetworkX, Matplotlib) que permite construir instâncias, escolher a raiz, executar os algoritmos e acompanhar logs e visualizações passo a passo. Testes em larga escala com instâncias aleatórias corroboram a corretude (coincidência de custos) e evidenciam diferenças estruturais entre as abordagens. Ainda, são discutidos princípios de interação humano–computador e carga cognitiva aplicados ao design da ferramenta, visando apoiar o ensino de otimização combinatória e teoria dos grafos.

Palavras-chave: arborescência inversa; Chu–Liu/Edmonds; András Frank; otimização combinatória; grafos dirigidos; visualização; PyScript; IHC.

Abstract

In this work we study and implement two classic approaches to the minimum-cost in-arborescence problem in directed graphs: the Chu–Liu/Edmonds algorithm (a primal approach based on cost normalization and cycle contraction/expansion) and András Frank's two-phase procedure (a dual approach based on c-feasible cut functions and zero-reduced-cost edge extraction). We provide Python implementations, along with a didactic web application (PyScript, NetworkX, Matplotlib) that lets users build instances, choose the root, run the algorithms, and follow step-by-step logs and visualizations. We ran large-scale randomized tests to confirm correctness (cost agreement) and highlight structural differences between the approaches. We also discuss human–computer interaction principles and cognitive load considerations applied to the tool's design to support teaching in combinatorial optimization and graph theory.

Keywords: in-arborescence; Chu–Liu/Edmonds; András Frank; combinatorial optimization; directed graphs; visualization; PyScript; HCI.

Sumário

Lista de Figuras

1 Introdução

Encontrar uma *r-arborescência inversa de custo mínimo* em grafos dirigidos é um problema estudado em ciência da computação desde os anos 1960, com formulações fundamentais apresentadas por Jack Edmonds em 1967 [6].

Essa busca dialoga com um princípio filosófico formulado na Idade Média por Guilherme de Ockham: a navalha de Occam (princípio da parcimônia), uma heurística filosófica segundo a qual, entre explicações concorrentes para um fenômeno, devemos preferir a mais simples ou a que faz menos suposições. Podemos pensar na navalha de Occam como critério de escolha entre explicações por meio de uma *teia explicativa mínima*: uma estrutura que conecta fatos ou observações com o mínimo de relações explicativas necessárias. Quando tais relações envolvem dependência ou causalidade, podemos representá-las pictograficamente como setas direcionadas entre os fatos (as hipóteses aparecem como rótulos dessas setas). Para refinar o modelo, associamos um custo a cada relação (por exemplo, o esforço para validar a relação ou a complexidade da explicação).

Encontrar a *teia explicativa mínima* equivale, nessa metáfora, a encontrar uma *r-arborescência de custo mínimo*: fixamos um vértice raiz r (a explicação inicial) e escolhemos um conjunto mínimo de relações explicativas de modo que todos os fatos tenham um caminho dirigido que leve a r, minimizando o custo total das arestas.

A *r-arborescência* (também chamada *out-arborescência*) orienta as arestas para fora de r: cada vértice $v \neq r$ tem exatamente uma aresta de entrada, e há um caminho dirigido único de r até v. Já a *r-arborescência inversa* (*in-arborescência*) orienta as arestas em direção a r: cada $v \neq r$ tem exatamente uma aresta de saída, e de cada vértice parte um caminho dirigido único até r [6, 7].

Com essa distinção em mente, este trabalho concentra-se na variante inversa. Formalmente, dado um grafo dirigido $G = (V, E)$ com custos $c : E \to \mathbb{R}^+$ nas arestas e um vértice raiz $r \in V$, procura-se uma *r-arborescência inversa* — isto é, uma árvore direcionada que atinja todos os vértices por caminhos dirigidos até r — que minimize o custo total das arestas selecionadas (cf. [6, 7]).

Nosso interesse, porém, não é apenas encontrar a arborescência mínima: o percurso até ela também importa, pois revela propriedades estruturais dos digrafos e ilumina técnicas distintas de otimização. Por isso, investigamos duas rotas clássicas e complementares: (i) o algoritmo de Chu–Liu/Edmonds, que opera por normalização dos custos das arestas de entrada, seleção sistemática de arestas de custo zero e contração de ciclos até obter um grafo reduzido, seguida pela reexpansão para reconstrução da solução [3, 6]; e (ii) a abordagem dual, em duas fases, de András Frank, fundamentada em cortes dirigidos, na qual se maximiza uma função de cortes c-viável para induzir arestas de custo zero e, em seguida, extrai-se a arborescência apenas a partir dessas arestas [7]. Embora assentados em princípios distintos — contração de ciclos no plano primal versus empacotamento/dualidade por cortes —, ambos os paradigmas produzem soluções ótimas e tornam explícitas a variedade de abordagens matemáticas que podem ser empregadas para resolver o mesmo problema.

1.1 Justificativa

A busca por uma *r-arborescência inversa de custo mínimo* em grafos dirigidos é um problema clássico com aplicações em diversas áreas, como redes de comunicação, planejamento de rotas, análise de dependências e modelagem de processos. Mas, não precisamos dessa justificação prática para nos interessarmos pelo problema: a riqueza estrutural dos digrafos e a variedade de técnicas algorítmicas disponíveis o tornam um excelente caso de estudo em otimização combinatória.

Do ponto de vista didático, a metáfora da "teia explicativa mínima" torna concreto o porquê de estudarmos arborescências enraizadas: ela mapeia perguntas sobre explicação, alcance e economia de recursos para estruturas dirigidas, servindo de fio condutor nas implementações e nos experimentos que apresentamos.

1.2 Objetivos

O objetivo principal deste trabalho é analisar, implementar e comparar duas abordagens clássicas para o problema de *r-arborescência de custo mínimo* em grafos dirigidos oferecendo uma aplicação *web* interativa que facilite o entendimento e a experimentação com o algoritmo de Chu–Liu/Edmonds e o método de András Frank, tornando-o acessível para estudantes e educadores.

1.3 Estrutura do Trabalho

Resumidamente, o trabalho abrange as seguintes frentes:

1. **Definições Preliminares**: fixação de notação e conceitos fundamentais (digrafos, arborescências, custos e dualidade).
2. **Em busca da Arborescência Perdida**: motivação do problema e intuições que guiam os algoritmos.
3. **Algoritmo de Chu–Liu/Edmonds**: detalhamento da abordagem primal por normalização/contração de ciclos, com ênfase em invariantes e correção.
4. **Procedimento em Duas Fases de András Frank**: detalhamento da abordagem dual por cortes c-viáveis, com ênfase em invariantes e correção.
5. **Discussão e Resultados: Chu-liu/Edmonds vs Frank**: comparação de desempenho e estrutura das soluções, com testes em larga escala.
6. **Didática do Abstrato**: discussão de desafios do ensino de matemática no ensino superior especialmente quando há o envolvimento de conceitos abstratos e como representações visuais atuam em favor da compreensão.
7. **A interação humano–computacional em ação - uma aplicação *web* interativa**: apresentação de conceitos em interação humano-computacional, a influência desses conceitos nas decisões tomadas para o desenvolvimento da aplicação *web* e detalhamento da implementação.
8. **Conclusão**: síntese das contribuições, limitações e trabalhos futuros.

Deste modo, o trabalho entrega implementações verificadas de Chu–Liu/Edmonds e András Frank, um visualizador *web* interativo e testes de volume que confirmam a equivalência de custos, úteis ao estudo e ao ensino de arborescências.

2 Definições Preliminares

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

Fixaremos notações e conceitos (conjuntos, relações, funções, digrafos, propriedade em digrafos, digrafos ponderados, ramificações geradoras, arborescências, funções de custo, dualidade, problemas duais e algoritmos), até chegar à formulação do problema da r-arborescência inversa de custo mínimo e adiamos descrições algorítmicas para capítulos posteriores.

2.1 Conjuntos

Este trabalho depende profundamente da teoria dos conjuntos, podemos dizer que todos os objetos matemáticos que iremos utilizar nessa dissertação se reduzem a conjuntos e operações entre eles.

Um **conjunto** é uma agregação de objetos distintos com características bem definidas, chamados elementos ou membros do conjunto. Os conjuntos são geralmente representados por letras maiúsculas (por exemplo, A, B, C) e seus elementos são listados entre chaves (por exemplo, $A = \{1, 2, 3\}$). Dois conjuntos são iguais se contêm exatamente os mesmos elementos.

Podemos ter conjuntos de qualquer tipo de objeto, incluindo números, letras e elementos da natureza. Para motivar as definições ao longo do texto, usaremos dois exemplos complementares que vamos chamar de exemplos-mestres:

- (i) Considere um universo U composto por três conjuntos: árvores T, plantas P e fungos F — para praticar pertinência, inclusão e operações; e

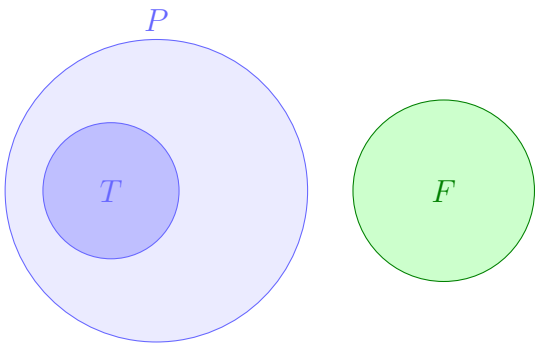


Figura 1: Relações entre os conjuntos de organismos: $T \subseteq P$ (toda árvore é planta) e F é disjunto de P.

- (ii) exemplo inspirado na navalha de Occam, com três famílias: evidências E, hipóteses H e explicações $\mathcal{M} \subseteq 2^H$. Nesse segundo exemplo, privilegiaremos

explicações parcimoniosas: entre as que cobrem E, preferimos as minimais por inclusão.

No segundo exemplo, consideremos $E = \{\text{queda de temperatura, céu nublado}\}$ e $H = \{H_A, H_B\}$, em que H_A significa "frente fria" e H_B, "ilha de calor". Tanto $\{H_A\}$ quanto $\{H_A, H_B\}$ explicam E (cobrem ambas as evidências), mas, por parcimônia, preferimos $\{H_A\}$, por ser estritamente menor do que $\{H_A, H_B\}$ ($\{H_A\} \subset \{H_A, H_B\}$). Ao longo do texto, recorreremos às noções de pertinência (por exemplo, $H_A \in H$), de inclusão e às operações usuais sobre conjuntos (união, interseção etc.) para comparar explicações.

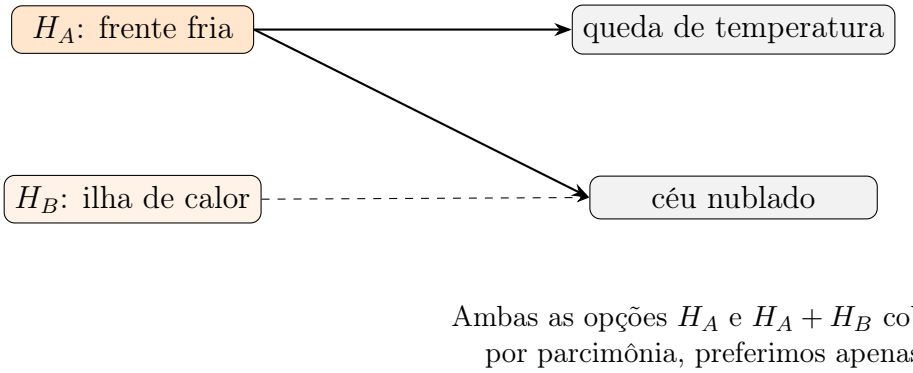


Figura 2: Exemplo inspirado na navalha de Occam: H_A cobre ambas as evidências (E), enquanto H_B seria redundante; prefere-se a explicação menor.

2.1.1 Subconjuntos

Dizemos que A é um **subconjunto** de B, denotado $A \subseteq B$, quando todo elemento de A também pertence a B. Se, além disso, $A \neq B$, escrevemos $A \subset B$ e chamamos A de **subconjunto próprio** de B. Ex.: $\{1, 2\} \subseteq \{1, 2, 3\}$ e $\{1, 2\} \subset \{1, 2, 3\}$.Por convenção, o conjunto vazio $\varnothing$ é subconjunto de qualquer conjunto X (isto é, $\varnothing \subseteq X$), e todo conjunto é subconjunto de si mesmo ($X \subseteq X$).

No primeiro exemplo-mestre, sejam P o conjunto de plantas, T o de árvores e F o de fungos; então $T \subseteq P$ (toda árvore é planta), ao passo que $F \not\subseteq P$.

No segundo, seja $H = \{H_A, H_B\}$ e $E = \{\text{queda de temperatura, céu nublado}\}$, vale $\{H_A\} \subset \{H_A, H_B\} \subseteq H$; ambas as opções explicam E, mas, por parcimônia, preferimos $\{H_A\}$.

2.1.2 Pertinência e inclusão

Pertinência e inclusão são os conceitos mais fundamentais da teoria dos conjuntos.

Começando pela **noção de pertinência** denotado por $\in$: dizemos que um elemento x pertence a um conjunto X quando $x \in X$ e não pertence quando $x \notin X$.

Seja o nosso universo U de organismos: $P = \{\text{todas as plantas}\}$, $T = \{\text{todas as árvores}\}$ e $F = \{\text{todos os fungos}\}$. Se x é um carvalho, então $x \in T$ e, como toda árvore é uma planta, $x \in P$. Já se y é um cogumelo, então $y \in F$ e, na taxonomia moderna, $y \notin T$ e $y \notin P$. Agora, considere $A = \{\text{árvores com folhas verdes}\}$; a pertinência fica clara: $x \in A$ se, e somente se, x é árvore e tem folhas verdes.

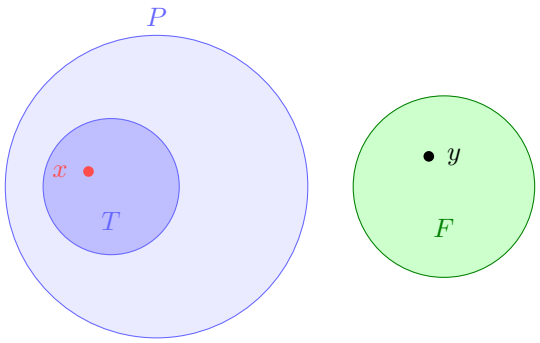


Figura 3: Pertinência: $x \in T \subseteq P$ (ponto vermelho dentro de T) e $y \in F$ (ponto preto); logo $y \notin P$ e $y \notin T$.

Continuando, vem a **relação de inclusão** entre conjuntos denotada por $\subseteq$: escrevemos $X \subseteq Y$ quando todo elemento de X também pertence a Y (e $X \subset Y$ quando, além disso, $X \neq Y$). No nosso exemplo, $A \subseteq T \subset P$ e $T \cap F = \varnothing$ (árvores e fungos não se sobrepõem).

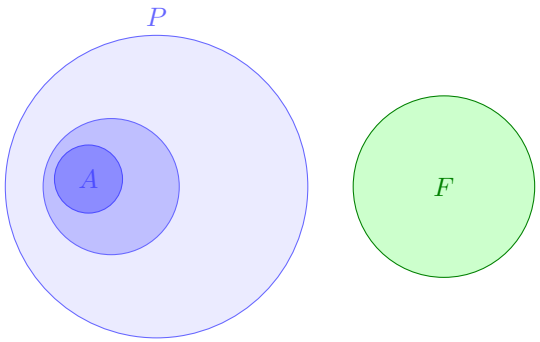


Figura 4: Inclusão: $A \subseteq T \subset P$ (círculos aninhados) e $T \cap F = \varnothing$ (círculos disjuntos).

2.1.3 Operações entre conjuntos

Com essas definições de pertinência, inclusão e subconjuntos, apresentamos as operações básicas entre conjuntos, que usaremos ao longo do texto (mantendo o exemplo com P, T, F, A).

Outras operações comuns entre conjuntos incluem:

- **União** ($A \cup B$): o conjunto de todos os elementos que pertencem a A, a B, ou ambos.

  Exemplo: $T \cup F$ é o conjunto de todos os organismos que são árvores ou fungos (ou ambos, se existissem tais organismos).

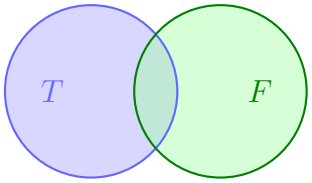


Figura 5: União: a área colorida representa $T \cup F$; a sobreposição evidencia a interseção $T \cap F$ (hipotética).

- **União disjunta** ($A \uplus B$): o conjunto de todos os elementos que pertencem a A ou a B, mas não a ambos; é igual a $A \cup B$ quando A e B são disjuntos.

  Exemplo: $T \uplus F$ é o conjunto de todos os organismos que são árvores ou fungos, mas não ambos (o que é trivialmente igual a $T \cup F$ pois T e F são disjuntos).

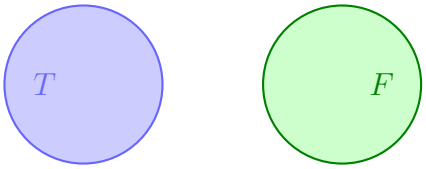


Figura 6: União disjunta: como $T \cap F = \varnothing$, tem-se $T \uplus F = T \cup F$.

- **Interseção** ($A \cap B$): o conjunto de todos os elementos que pertencem tanto a A quanto a B.

  Exemplo: $T \cap P = T$, pois todas as árvores são plantas.

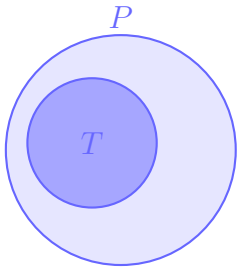


Figura 7: Interseção: como $T \subseteq P$, $T \cap P = T$ (a região escura é T).

- **Diferença** ($A \setminus B$): o conjunto de todos os elementos que pertencem a A mas não a B.

  Exemplo: $T \setminus A$ é o conjunto de todas as árvores que não têm folhas verdes.

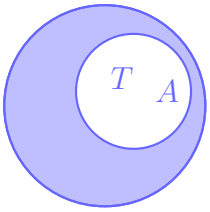


Figura 8: Diferença: região azul representa $T \setminus A$ (árvores que não têm folhas verdes).

- **Complemento** de X em um universo fixo U: $X^c := U \setminus X$ (também chamado de *complemento absoluto*); o *complemento relativo* de X em Y é $Y \setminus X$.

  Exemplo: $T^c = U \setminus T$ é o conjunto de todos os organismos que não são árvores. Ou seja, T^c inclui plantas que não são árvores, fungos e quaisquer outros organismos no universo U.

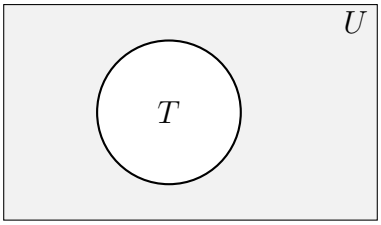


Figura 9: Complemento: a área cinza representa $T^c = U \setminus T$.

- **Diferença simétrica** $(A \,\Delta\, B)$: $(A \setminus B) \cup (B \setminus A)$; é igual a $A \cup B$ quando A e B são disjuntos.

  Exemplo: $P \,\Delta\, F$ é o conjunto de todos os organismos que são plantas ou fungos, mas não ambos (o que é trivialmente igual a $P \cup F$ pois P e F são disjuntos).

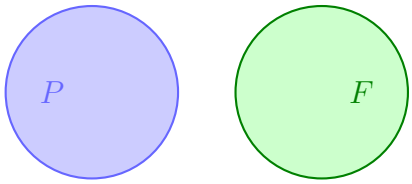


Figura 10: Diferença simétrica: como $P \cap F = \varnothing$, temos $P \,\Delta\, F = P \cup F$.

- **Produto cartesiano** $(A \times B)$: o conjunto de pares ordenados (a, b) com $a \in A$ e $b \in B$.

  Exemplo: $T = \{t_1, t_2\}$ e $F = \{f_1, f_2\}$. Então $T \times F = \{(t_1, f_1), (t_1, f_2), (t_2, f_1), (t_2, f_2)\}$.

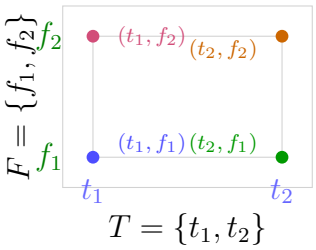


Figura 11: Produto cartesiano: pontos representam os pares de $T \times F$ para $T = \{t_1, t_2\}$ e $F = \{f_1, f_2\}$.

- **Conjunto das partes** (2^U): a família de todos os subconjuntos de U (inclui $\varnothing$ e o próprio U).

  Exemplo: se $U = \{x, y\}$, então $2^U = \{\varnothing, \{x\}, \{y\}, \{x, y\}\}$. Logo, $|2^U| = 4 = 2^{|U|}$.

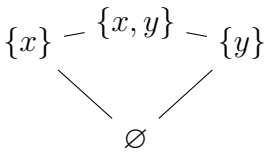


Figura 12: Conjunto das partes: diagrama de Hasse de 2^U para $U = \{x, y\}$.

Identidades úteis. Usaremos livremente as propriedades clássicas de conjuntos — comutatividade e associatividade de $\cup$ e $\cap$, distributividade e as **leis de De Morgan** — sem prova. Quando for relevante, explicitaremos a identidade no ponto de uso. Por exemplo, no nosso universo U, $(P \cup F)^c = P^c \cap F^c$.

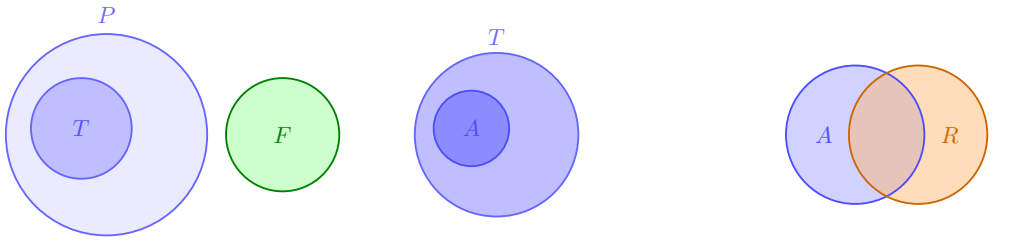
2.1.4 Coleção

Entre os objetos que podem pertencer a um conjunto, estão também eles mesmos, outros conjuntos. Chamaremos tais conjuntos de **coleções** (ou **famílias**) de conjuntos. Por exemplo, $\mathcal{C} = \{P, T, F\}$ é uma coleção formada pelos conjuntos de organismos já definidos: plantas P, árvores T e fungos F. Note que $\mathcal{C}$ é um conjunto como outro qualquer; seus elementos são, cada um, um conjunto.

Coleções são úteis para agrupar subconjuntos relacionados de um mesmo universo. Por exemplo, considere $\mathcal{D} = \{A, B\}$, onde $A = \{\text{árvores com folhas verdes}\}$ e $B = \{\text{árvores com folhas vermelhas}\}$. Assim, $\mathcal{D} \subseteq 2^T$ é uma coleção de subconjuntos de T.

Uma coleção $\mathcal{F}$ é dita **laminar** quando, para quaisquer $X, Y \in \mathcal{F}$, vale que $X \subseteq Y$, $Y \subseteq X$ ou $X \cap Y = \varnothing$; isto é, quaisquer dois conjuntos são aninhados (um está contido no outro) ou são disjuntos.

Por exemplo, na coleção $\mathcal{C} = \{P, T, F\}$: P é o conjunto de todas as plantas, T o de todas as árvores (portanto $T \subseteq P$) e F o de todos os fungos (disjunto de plantas e, logo, de árvores). Assim, quaisquer dois conjuntos em $\mathcal{C}$ são aninhados ou disjuntos, e $\mathcal{C}$ é laminar. Na coleção $\mathcal{D} = \{A, T\}$: A é o conjunto de árvores com folhas verdes e T o de todas as árvores; como toda árvore de A é árvore de T, temos $A \subseteq T$ e a coleção é laminar. Já em $\mathcal{E} = \{A, R\}$: R é o conjunto de árvores frutíferas; há árvores que são ao mesmo tempo frutíferas e de folhas verdes (a interseção é não vazia), mas nenhuma das classes contém a outra, então $\mathcal{E}$ não é laminar.



(a) $\mathcal{C} = \{P, T, F\}$ — laminar (b) $\mathcal{D} = \{A, T\}$ — laminar (c) $\mathcal{E} = \{A, R\}$ — não laminar

Figura 13: Laminaridade em coleções: em (a) e (b), quaisquer dois conjuntos são aninhados ou disjuntos; em (c), A e R se interceptam sem inclusão, violando a laminaridade.

Este é um importante conceito que aparecerá no restante do trabalho. A ideia de laminaridade retornará quando tratarmos de cortes dirigidos.

2.1.5 Comparando conjuntos: cardinalidade e maximalidade

Podemos comparar conjuntos através de relações de tamanho (cardinalidade) ou por relações de inclusão. Essas duas formas de comparação são distintas e importantes, especialmente quando lidamos com coleções de conjuntos.

A **cardinalidade** de um conjunto A, denotada por $|A|$, é o número de elementos de A. Para conjuntos finitos, é simplesmente a contagem dos elementos (por exemplo, se $A = \{1, 2, 3\}$, então $|A| = 3$). Para conjuntos infinitos, a cardinalidade pode ser mais complexa, envolvendo conceitos como infinito enumerável e não enumerável. Por exemplo, o conjunto dos números naturais $\mathbb{N}$ é infinito enumerável, enquanto o conjunto dos números reais $\mathbb{R}$ é infinito não enumerável.

Dizemos que $A \in \mathcal{C}$ tem **maior cardinalidade** se $|A| \geq |B|$ para todo $B \in \mathcal{C}$ (podendo haver empates). Esse critério não coincide, em geral, com a comparação por relação de inclusão. Em grafos, por exemplo, distinguem-se conjuntos independentes *maximais* (não ampliáveis) de conjuntos independentes *máximos* (de cardinalidade máxima).

Ao compararmos uma coleção $\mathcal{C}$ de conjuntos utilizando sua relações de inclusão $(\mathcal{C}, \subseteq)$, é imprescindível distinguir **maximal** de **máximo**.

Um conjunto $A \in \mathcal{C}$ é **maximal** se não existe $B \in \mathcal{C}$ tal que $A \subset B$. Em palavras: não dá para ampliar A estritamente dentro da coleção. Podem haver vários elementos

maximais, e eles podem ser incomparáveis entre si. Ex.: em $\mathcal{C} = \big\{\{1\}, \{2\}\big\}$, ambos $\{1\}$ e $\{2\}$ são maximais, mas não existe máximo.

Um conjunto $A \in \mathcal{C}$ é **máximo** se $B \subseteq A$ para todo $B \in \mathcal{C}$. Se existe, é único. Ex.: em $\mathcal{C} = \big\{\{1\}, \{2\}, \{1, 2\}\big\}$, o conjunto $\{1, 2\}$ é o máximo.

Um bom exemplo para ilustrar a distinção entre conjuntos maximais e máximos é a coleção $\mathcal{C} = \big\{\{1\}, \{2\}, \{1, 2\}, \{3\}\big\}$. Aqui, $\{1, 2\}$ é o único conjunto máximo (contém todos os outros), enquanto $\{1\}$, $\{2\}$ e $\{3\}$ são todos maximais (não podem ser ampliados dentro da coleção).

Esses conceitos reaparecerão ao longo do texto, especialmente na diferença entre estruturas **maximais** (saturadas por inclusão) e **máximas/ótimas** (de maior cardinalidade ou menor custo). Para fixar ideias:

- Em muitos problemas, "**maximal**" quer dizer: não dá para ampliar uma escolha sem violar as regras; já "**máximo/ótimo**" quer dizer: entre todas as escolhas válidas, essa é a melhor segundo o critério (por exemplo, menor custo).
- No algoritmo de **Chu–Liu/Edmonds**, começamos com escolhas locais que já não podem ser ampliadas dentro das regras do problema e, a partir delas, chegamos a uma solução de menor custo.
- No método de **András Frank**, primeiro construímos uma estrutura organizada que garante escolhas suficientes; depois, usando apenas relações já ativadas por essa estrutura, extraímos a solução ótima.
- Moral: partimos da ideia de "não dá para aumentar" (maximal) e chegamos a "melhor possível" (máximo/ótimo). Os detalhes técnicos de cada método aparecerão nas seções próprias.

2.2 Relações e Funções

Desde a introdução, vimos a ideia filosófica de explicar como "ligar" fatos a hipóteses da forma mais parcimoniosa possível. Para tornar essa intuição precisa, precisamos de uma linguagem que descreva objetos (conjuntos) e como eles se conectam. É aqui que entram as **relações** e, de modo ainda mais disciplinado, as **funções**: regras que associam a cada elemento de um conjunto exatamente um elemento de outro. Com elas, passamos do discurso qualitativo sobre explicações para uma estrutura matemática que permite medir, comparar e, adiante, otimizar.

Para tornar essa discussão prática, desenvolvemos uma aplicação *web* interativa que permite visualizar passo a passo o funcionamento dos dois algoritmos, destacando suas diferenças e semelhanças. A seguir, discutimos a dimensão didática que motivou essas escolhas e como fundamentos de aprendizagem e visualização embasam o projeto.

Uma **relação** R entre dois conjuntos A e B é um subconjunto do produto cartesiano $A \times B$. Ou seja, $R \subseteq A \times B$. Se $(a, b) \in R$, dizemos que a está relacionado a b pela relação R, denotado aRb.

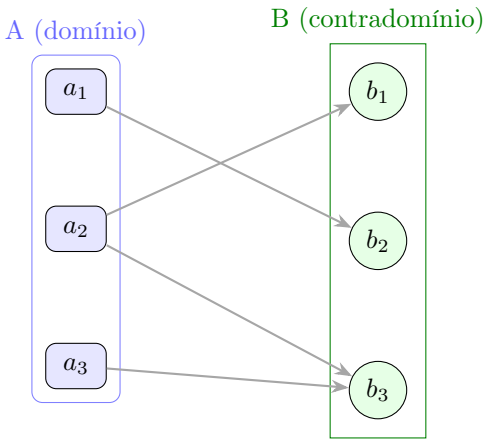


Figura 14: Relação $R \subseteq A \times B$. Cada seta representa um par $(a, b) \in R$ (isto é, $a\,R\,b$). As formas/cores distinguem domínio (A, retângulos azuis) de contradomínio (B, círculos verdes). Note que a_2 se relaciona com b_1 e b_3; logo, este R *não é função*.

No nosso primeiro exemplo-mestre, considere $P = \{\text{todas as plantas}\}$ e $F = \{\text{todos os fungos}\}$. Definimos a relação R como "é um organismo que compete com". Assim, se uma planta $p \in P$ compete com um fungo $f \in F$, então $(p, f) \in R$.

No nosso segundo exemplo, considere $H = \{\text{hipóteses}\}$ e $E = \{\text{evidências}\}$. Definimos a relação R como "explica". Se uma hipótese $h \in H$ explica uma evidência $e \in E$, então $(h, e) \in R$.

Em teoria dos grafos, uma relação pode representar conexões entre vértices. Por exemplo, em um grafo dirigido, a relação "existe uma aresta de u para v" pode ser representada como um conjunto de pares ordenados (u, v).

Uma **função** f de um conjunto A em um conjunto B é uma relação especial que associa cada elemento de A a exatamente um elemento de B. Denotamos isso como $f : A \to B$. Se $f(a) = b$, dizemos que b é a imagem de a sob f.

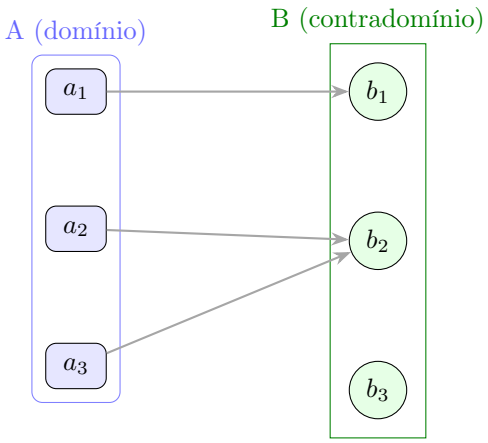


Figura 15: Função $f : A \to B$. Cada elemento de A tem *exatamente uma* imagem em B.

No nosso exemplo-mestre, considere $P = \{\text{todas as plantas}\}$ e $\mathbb{N} = \{0, 1, 2, \ldots\}$ (números naturais). Definimos a função $f : P \to \mathbb{N}$ que associa cada planta ao seu número de folhas. Se $p \in P$ é uma árvore com 100 folhas, então $f(p) = 100$.

Em teoria dos grafos, funções podem ser usadas para atribuir valores numéricos — também chamados de pesos ou custos — às arestas. Por exemplo, se temos um grafo G com arestas $e_1, e_2, \ldots, e_n$, podemos definir uma função $c : E \to \mathbb{R}^+$ que atribui um peso $c(e_i)$ a cada aresta e_i.

Na ciência da computação, relações e funções são usadas para modelar conexões entre dados, estruturas de dados e operações. Por exemplo, em bancos de dados relacionais, tabelas representam relações entre diferentes entidades. Em programação funcional, funções são tratadas como cidadãos de primeira classe, permitindo a criação de funções de ordem superior que podem receber outras funções como argumentos ou retorná-las como resultados.

2.2.1 Conceitos em Funções

Alguns conceitos importantes relacionados a funções incluem:

- **Domínio**: o conjunto A de entrada da função $f : A \to B$.
- **Contradomínio**: o conjunto B de possíveis saídas da função.
- **Imagem**: o conjunto de valores efetivamente atingidos pela função, $f(A) = \{f(a) \mid a \in A\}$.

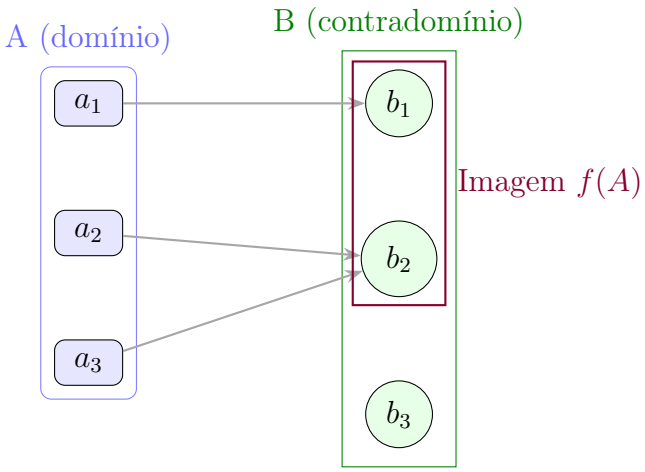


Figura 16: Domínio, contradomínio e imagem: A (retângulos azuis) mapeia via f para B (círculos verdes). A imagem $f(A)$ é o subconjunto de B efetivamente atingido (aqui, $\{b_1, b_2\}$).

- **Injetora**: uma função f é injetora se $f(a_1) = f(a_2)$ implica $a_1 = a_2$; ou seja, elementos distintos do domínio têm imagens distintas.
- **Sobrejetora**: uma função f é sobrejetora se para todo $b \in B$, existe $a \in A$ tal que $f(a) = b$; ou seja, a imagem é igual ao contradomínio.
- **Bijetora**: uma função que é tanto injetora quanto sobrejetora; estabelece uma correspondência um-para-um entre os elementos de A e B.

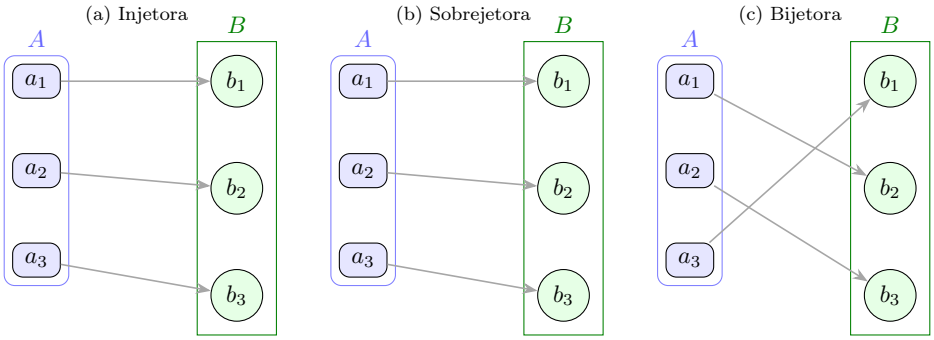


Figura 17: Funções especiais: (a) Injetora — elementos distintos em A têm imagens distintas em B; (b) Sobrejetora — todo elemento de B é imagem; (c) Bijetora — um-para-um e sobre B.

2.2.2 Funções de agregação e somatórios

Além de relacionar elementos de conjuntos, muitas operações familiares em matemática, envolvem *funções* que recebem coleções de números (ou funções) e devolvem um número.

Uma **função de agregação** é uma função que recebe um conjunto (ou sequência) de valores e retorna um único valor que representa algum aspecto agregado desses valores. Exemplos comuns incluem:

- **Média**: A média aritmética de um conjunto de números $x_1, x_2, \ldots, x_n$ é dada por $\frac{1}{n}\sum_{i=1}^{n} x_i$.
- **Máximo e Mínimo**: A função máximo retorna o maior valor em um conjunto, enquanto a função mínimo retorna o menor valor.
- **Produto**: O produto de um conjunto de números $x_1, x_2, \ldots, x_n$ é dado por $\prod_{i=1}^{n} x_i$.
- **Contagem**: A função contagem retorna o número de elementos em um conjunto.

O **somatório**, por exemplo, é uma função de agregação linear que mapeia uma sequência $(x_1, \ldots, x_n)$ em sua soma:

$$\sum_{i=1}^{n} x_i.$$

Esse conceito é especialmente útil em otimização e em análise combinatória: somatórios aparecem o tempo todo e serão explorados ao longo deste trabalho.

Exemplos com grafos. As sessões seguintes explorarão em maiores detalhes grafos e digrafos, mas agora, consideremos a ideia básica: um **grafo** é um conjunto de pontos (*vértices*) ligados por linhas (*arestas*). No caso *não dirigido*, as linhas não têm seta; no caso *dirigido*, cada linha tem um sentido e é chamada de *arco*.

A noção de somatória aparecerá naturalmente quando lidamos com propriedades dos grafos. Por exemplo:

Seja um grafo não dirigido $G = (V, E)$. O **grau** de um vértice $v \in V$, escrito $\deg(v)$, é quantas arestas tocam em v. A soma dos graus de todos os vértices conta cada aresta *duas vezes* (uma por extremidade), portanto:

$$\sum_{v \in V} \deg(v) = 2\,|E|.$$

Em grafos dirigidos, distinguimos $\deg^-(v)$ (quantos arcos *chegam* em v) e $\deg^+(v)$ (quantos arcos *saem* de v). Cada arco contribui com 1 para um grau de saída e 1 para um grau de entrada, logo:

$$\sum_{v \in V} \deg^-(v) = \sum_{v \in V} \deg^+(v) = |E|.$$

Agora suponha que cada aresta/arco $e \in E$ tenha um *peso* (ou *custo*) $c(e) \geq 0$. O **custo total** de um subconjunto $F \subseteq E$ é simplesmente a soma dos pesos das arestas escolhidas:

$$C(F) = \sum_{e \in F} c(e).$$

De maneira análoga, se $X \subseteq V$ é um conjunto de vértices, o **valor total** (ou peso total) dos arcos que *saem* de X é a soma dos pesos dessas setas. Usaremos mais adiante a notação $\delta^+(X)$ para o conjunto de arcos que saem de X (ver a seção de digrafos); com essa notação,

$$\mathrm{val}^+(X) = \sum_{e \in \delta^+(X)} c(e).$$

Esses exemplos mostram como somatórios capturam propriedades estruturais do grafo por meio de funções simples de agregação.

2.2.3 Funções Especiais

Função de custo

Uma **função de custo** é uma função $c : A \to \mathbb{R}^+$ que atribui um valor numérico não negativo (custo) a cada elemento de um conjunto A. Essas funções são amplamente utilizadas em otimização, economia e teoria dos grafos para modelar despesas, penalidades ou recursos associados a escolhas ou ações.

Exemplo: Considere um conjunto de tarefas $T = \{t_1, t_2, t_3\}$. Uma função de custo $c : T \to \mathbb{R}^+$ pode ser definida como:

$$c(t_1) = 5, \quad c(t_2) = 10, \quad c(t_3) = 3.$$

Aqui, $c(t_i)$ representa o custo de realizar a tarefa t_i.

Depende diretamente do conceito de somatório, pois frequentemente queremos minimizar o custo total de um conjunto de escolhas. Se $S \subseteq A$ é um subconjunto de elementos escolhidos, o custo total associado a S é dado por:

$$C(S) = \sum_{a \in S} c(a).$$

Função c-disjunta Uma **função c-disjunta** é uma função $f : A \to B$ que, para quaisquer $a_1, a_2 \in A$ com $a_1 \neq a_2$, as imagens $f(a_1)$ e $f(a_2)$ são disjuntas, ou seja, $f(a_1) \cap f(a_2) = \varnothing$. Em outras palavras, elementos distintos do domínio são mapeados para conjuntos disjuntos no contradomínio.

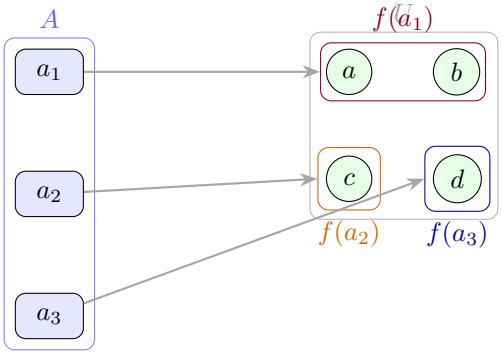


Figura 18: Função c-disjunta: cada $a \in A$ mapeia para um *subconjunto* de B, e imagens de elementos distintos são disjuntas.

Exemplo: Considere $A = \{1, 2, 3\}$ e $B = \{\{a\}, \{b\}, \{c\}, \{d\}\}$. Definimos a função $f : A \to B$ como:

$$f(1) = \{a, b\}, \quad f(2) = \{c\}, \quad f(3) = \{d\}.$$

Aqui, f é c-disjunta, pois $f(1) \cap f(2) = \varnothing$, $f(1) \cap f(3) = \varnothing$ e $f(2) \cap f(3) = \varnothing$.

Função c-viável

Uma **função c-viável** é uma função $g : A \to \mathbb{R}^+$ que satisfaz certas condições de viabilidade relacionadas a um conjunto de restrições ou critérios. Essas funções são frequentemente usadas em otimização e teoria dos grafos para garantir que as soluções propostas atendam a requisitos específicos.

Exemplo: Considere um conjunto de projetos $P = \{p_1, p_2, p_3\}$ e uma função $g : P \to \mathbb{R}^+$ que atribui um valor de viabilidade a cada projeto. Suponha que temos a restrição de que a soma dos valores de viabilidade deve ser menor ou igual a um certo limite L. Se definirmos:

$$g(p_1) = 4, \quad g(p_2) = 6, \quad g(p_3) = 3,$$

então a função g é c-viável se $g(p_1) + g(p_2) + g(p_3) \leq L$.

Essas funções são essenciais para garantir que as soluções propostas em problemas de otimização sejam práticas e atendam aos critérios estabelecidos.

Funções de otimização

Do ponto de vista semiótico, "melhor" exprime uma preferência entre interpretações: ao comparar alternativas, escolhemos aquela cuja significação é mais adequada a um critério. Para tornar isso operacional, a matemática troca "fazer mais sentido" por "ter maior (ou menor) valor" em uma escala formal: fixamos (i) um conjunto de soluções viáveis $\mathcal{F}$ e (ii) uma função numérica sobre $\mathcal{F}$ que induz uma ordem de comparação.

Formalmente, usamos uma **função objetivo** (ou **função de otimização**)

$$h : \mathcal{F} \to \mathbb{R},$$

que atribui um número real a cada solução. Buscamos uma solução $S^* \in \mathcal{F}$ que *minimize* ou *maximize* h (isto é, um argmin ou argmax). Quando esse número resulta da soma de contribuições elementares, obtemos o caso aditivo, em ligação direta com os somatórios apresentados antes.

Caso *aditivo*. Quando cada elemento $a \in A$ tem um custo $c(a) \geq 0$ e as soluções são subconjuntos $S \subseteq A$, a função objetivo mais comum é o custo total

$$C(S) = \sum_{a \in S} c(a),$$

que desejamos *minimizar*. De modo análogo, se cada item tem um benefício $p(a) \geq 0$, podemos *maximizar* o benefício total $P(S) = \sum_{a \in S} p(a)$, possivelmente sujeito a restrições (por exemplo, de orçamento ou limite).

Outro exemplo: considere produtos $X = \{x_1, x_2, x_3\}$ com lucro $p(x_1) = 10$, $p(x_2) = 15$, $p(x_3) = 7$. Se houver um limite de custo que impede escolher todos, o objetivo típico é escolher um subconjunto $S \subseteq X$ que maximize $\sum_{x \in S} p(x)$ respeitando as restrições. Essa forma reflete exatamente os somatórios introduzidos antes.

Em todos os casos, a função objetivo explicita o critério de "melhor", e as restrições determinam quais soluções são aceitáveis.

2.2.4 Otimização

O princípio da navalha de occam nos diz que a explicação mais simples tende a ser a correta. Do ponto de vista semiótico, isso é escolher, entre interpretações possíveis, a que melhor satisfaz um critério. A matemática também se preocupa com identificar a "melhor" solução entre várias alternativas, mas traduz essa ideia em termos quantitativos: fixamos (i) um conjunto de soluções viáveis $\mathcal{F}$ e (ii) uma função numérica sobre $\mathcal{F}$ que induz uma ordem de comparação.

Assim, a otimização envolve a maximização ou minimização de uma função objetivo $h : \mathcal{F} \to \mathbb{R}$ sobre um conjunto de soluções viáveis $\mathcal{F}$.

Esse conceito pode aparecer em muitas necessidades do dia-a-dia: uma empresa pode querer minimizar custos de produção, um viajante pode buscar o caminho mais curto entre dois pontos, ou um investidor pode tentar maximizar o retorno de um portfólio. Em cada caso, a função objetivo quantifica o que significa ser "melhor" ou "mais eficiente".

Pensando em modelagem de problemas em grafos, podemos pensar em exemplos clássicos de otimização:

- **Caminho mais curto**: Dado um grafo com pesos nas arestas, encontrar o caminho entre dois vértices que minimize a soma dos pesos das arestas percorridas.
- **Árvore geradora mínima**: Encontrar uma árvore que conecte todos os vértices de um grafo com o menor custo total das arestas.
- **Fluxo máximo**: Em um grafo direcionado com limites nas arestas, encontrar o fluxo máximo que pode ser enviado de uma fonte a um sumidouro sem exceder esses limites.

2.2.5 A dualidade

O taoísmo chinês fala do yin e yang, forças opostas que se complementam. Um conceito que remete à contrastes, noite e dia, matéria e anti-matéria, máximos e mínimos. Na matemática, um conceito semelhante é a *dualidade*, que conecta problemas de minimização a problemas de maximização.

Em termos matemáticos, para cada problema de otimização (o *primal*), existe um problema associado (o *dual*) que oferece uma perspectiva complementar. Resolver um desses problemas pode fornecer insights ou soluções para o outro.

Exemplo: Considere um problema de otimização onde queremos minimizar o custo de transporte de mercadorias entre diferentes armazéns. O problema primal busca a solução de transporte que minimize os custos totais, enquanto o problema dual pode ser formulado como a maximização do valor dos recursos disponíveis (como a capacidade dos armazéns e a demanda dos clientes).

No nosso texto, consideramos como problema primal a minimização do custo de uma estrutura (como uma árvore geradora mínima) e como dual a maximização de um conjunto de pesos ou preços que justificam esse custo mínimo. A relação entre primal e dual é formalizada por teoremas de dualidade, que garantem que o valor ótimo do primal é igual ao valor ótimo do dual sob certas condições.

Esse ponto de vista leva a "teoremas min–max" que ligam problemas de *minimização* a problemas de *maximização* e fornecem certificados verificáveis de otimalidade.

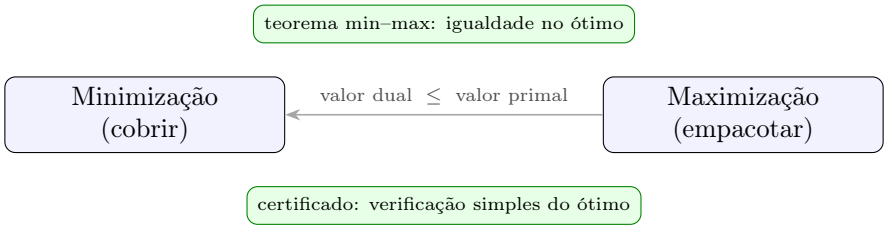


Figura 19: Intuição de min–max: um problema de cobrir (minimização) e um de empacotar (maximização) andam juntos. Sempre vale valor dual ≤ valor primal; quando há igualdade, temos um certificado de otimalidade.

Uma forma didática de ver a dualidade (no caso linear) é a seguinte. Temos um conjunto de soluções possíveis (um poliedro)

$$P = \{x \in \mathbb{R}^n : \ Ax \geq b\},$$

e um vetor $c \in \mathbb{R}^n$ que mede o *custo* de cada coordenada de x; o valor da solução é a soma ponderada $c^\top x$. O problema **dual** escolhe $y \geq 0$ (um "preço" para cada restrição de $Ax \geq b$) exigindo que nenhuma variável fique "subprecificada": isso é expresso por

$$A^\top y \ \leq \ c.$$

Com essa única condição, todo $y \geq 0$ fornece automaticamente um **limitante inferior** para $c^\top x$ em todo P:

$$c^\top x \ \geq \ (A^\top y)^\top x \ = \ y^\top (Ax) \ \geq \ y^\top b.$$

Geometricamente, $y^\top b$ é o nível de um *hiperplano de suporte* que nunca ultrapassa a função objetivo $c^\top x$ sobre P.

No **ótimo**, existem x (primal) e y (dual) viáveis com o *mesmo* valor

$$c^\top x \ = \ y^\top b,$$

isto é, o "vão de dualidade" é zero. Além disso, valem as condições de **complementaridade**, que dizem "onde há folga de um lado, há zero do outro":

$$x \odot \big(c - A^\top y\big) = 0 \quad \text{e} \quad y \odot \big(Ax - b\big) = 0.$$

Interpretando:

- Se $x_j > 0$, então o *custo reduzido* daquela coordenada é nulo: $c_j - (A^\top y)_j = 0$. Caso contrário, pode haver folga $c_j > (A^\top y)_j$ e a variável fica em zero.

- Se $y_i > 0$, então a i-ésima restrição está "apertada" (sem folga): $(Ax - b)_i = 0$. Caso contrário, se há folga $(Ax - b)_i > 0$, o preço y_i zera.

Essas igualdades capturam a intuição central: o dual estabelece preços que justificam o valor mínimo do primal, e as soluções ótimas usam apenas "direções" cujo custo reduzido é zero e apoiam-se em restrições ativas.

No contexto de grafos, a otimização costuma aparecer como a busca por subestruturas (caminhos, árvores, cortes, fluxos) que minimizam ou maximizam um custo, sempre respeitando a topologia do grafo.

Nesta dissertação, essa noção de otimização é central: olhamos para o mesmo problema por dois ângulos que se completam. No lado "primal", queremos montar diretamente a arborescência de menor custo. O algoritmo de Chu–Liu/Edmonds faz isso de forma gulosa: ajusta os custos por vértice, cria arestas de custo zero (0-arestas), contrai ciclos quando aparecem e segue até montar a solução ótima. ([3, 6]).

No lado "dual", em vez de montar a árvore, colocamos custos em cortes do grafo com raiz r. A regra é simples: nenhum custo pode ultrapassar o custo das arestas que cruzam o corte. Buscamos escolher esses custos para somar o máximo possível. As arestas que "batem no limite" viram 0-arestas, e a partir delas conseguimos reconstruir uma arborescência ótima. Essa visão, desenvolvida por Frank, leva a um teorema min–max e a um procedimento em duas etapas: primeiro ajustamos os custos, depois extraímos a solução usando apenas 0-arestas. (cf. [7, 21])

2.3 Problemas interessantes

Qual o número mínimo de cores necessárias para colorir um mapa de países, de modo que países vizinhos tenham cores diferentes? Qual o caminho mais curto entre duas cidades em um mapa rodoviário? Como encontrar a árvore geradora mínima que conecta todas as cidades com o menor custo total? Essas perguntas são exemplos clássicos de problemas que podem ser modelados e resolvidos usando a teoria dos grafos. Vistas sob a lente da navalha de occam, todas elas buscam a solução mais parcimoniosa que atende ao requisito: usar poucas cores, percorrer um caminho curto ou conectar tudo com custo mínimo.

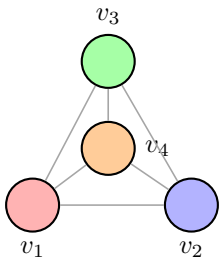


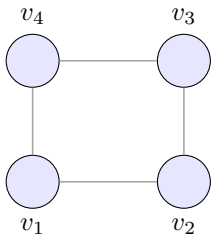
Figura 20: Coloração de grafos: exemplo de coloração própria do grafo completo K_4. Como K_4 é completo, precisamos de 4 cores para colorir seus vértices de modo que vértices adjacentes tenham cores diferentes. Uma coloração é uma função $\varphi : V \to C$ tal que, se $uv \in E$, então $\varphi(u) \neq \varphi(v)$.

Sem a teoria dos grafos, seria difícil formalizar e resolver esses problemas de maneira eficiente. Ao representar situações do mundo real como grafos, tornamos a parcimônia da navalha de occam algo operacional: escolhemos uma medida simples (número de cores, comprimento, custo) e aplicamos algoritmos que, entre as soluções viáveis, minimizam ou maximizam esse critério — produzindo soluções ótimas ou, quando necessário, boas aproximações.

2.4 Grafos

Falamos bastante de grafos ao longo do texto, aqui fixamos a noção básica.

Um **grafo** $G = (V, E)$ é uma estrutura matemática composta por um conjunto V de *vértices* (ou *nós*) e um conjunto E de *arestas* (ou *ligações*) que conectam pares de vértices.



$V = \{v_1, v_2, v_3, v_4\}$
$E = \{\{v_1, v_2\}, \{v_2, v_3\}, \{v_3, v_4\}, \{v_4, v_1\}\}$

Figura 21: Definição de grafo: exemplo de grafo simples $G = (V, E)$. Pontos representam os vértices V e linhas representam as arestas E, que são pares não ordenados de vértices distintos.

O conjunto de vértices V pode ser definido como $V = \{v_1, v_2, \ldots, v_n\}$, onde cada v_i representa um ponto distinto no grafo. O conjunto de arestas E é um conjunto de pares não ordenados de vértices, ou seja, $E \subseteq \{\{u, v\} \mid u, v \in V, u \neq v\}$. Cada aresta $\{u, v\}$ indica uma conexão entre os vértices u e v.

Esses vértices e arestas podem representar uma variedade de entidades e relações no mundo real. Por exemplo, em um grafo que modela uma rede social, os vértices podem representar pessoas, e as arestas podem representar amizades entre elas. Em um grafo que representa uma rede de transporte, os vértices podem ser cidades, e as arestas podem ser estradas ou rotas de voo conectando essas cidades.

Tendo em mente esses problemas, podemos falar de custos associados às arestas. Por exemplo, em um grafo que representa uma rede de transporte, cada aresta pode ter um custo associado, como a distância entre duas cidades ou o tempo necessário para percorrer uma estrada. Em um grafo que modela uma rede de comunicação, as arestas podem ter custos relacionados à largura de banda ou à latência.

Esses custos representam uma função $c : E \to \mathbb{R}^+$ que atribui um valor numérico não negativo a cada aresta do grafo. Assim, para cada aresta $\{u, v\} \in E$, $c(\{u, v\})$ representa o custo associado a essa conexão.

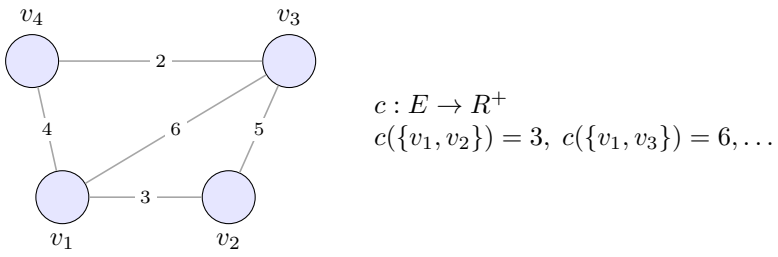


Figura 22: Grafo com custos nas arestas: a função $c : E \to \mathbb{R}^+$ atribui um custo não negativo a cada aresta.

Grafos apresentam diversas estruturas especiais. Essas estruturas são definidas como subgrafos e são fundamentais para entender a topologia e as propriedades dos grafos, e muitas vezes são o foco de problemas de otimização.

2.4.1 Subgrafos

Um **subgrafo** $H = (V_H, E_H)$ de um grafo $G = (V, E)$ é um grafo cujos vértices e arestas são subconjuntos dos vértices e arestas de G. Formalmente, $V_H \subseteq V$ e $E_H \subseteq E$, e cada aresta em E_H conecta dois vértices em V_H.

Alguns subgrafos interessantes incluem: caminhos, ciclos, componentes conexas e árvores. Muitas propriedades e algoritmos em grafos dependem dessas estruturas, como encontrar o caminho mais curto entre dois vértices, detectar ciclos ou construir árvores geradoras mínimas. Dentre essas muitas estruturas especiais, vamos apresentar as que são relevantes para o desenvolvimento desta dissertação:

Caminhos

Um **caminho** em um grafo é uma sequência de vértices conectados por arestas. Formalmente, um caminho P de comprimento $k \geq 1$ é uma sequência de vértices $P = (v_1, v_2, \ldots, v_{k+1})$ tal que cada par consecutivo (v_i, v_{i+1}) é uma aresta em E. O comprimento do caminho é o número de arestas que ele contém, que é k.

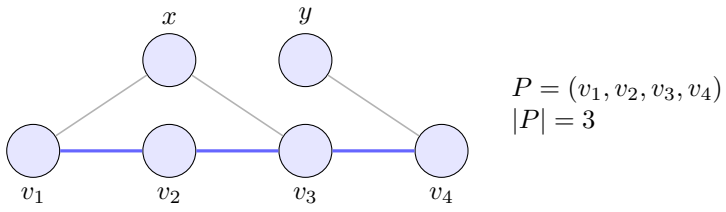


Figura 23: Caminho em grafo não dirigido: o caminho $P = (v_1, v_2, v_3, v_4)$ está destacado em azul. Seu comprimento é o número de arestas percorridas, $|P| = 3$.

Ciclos

Um **ciclo** é um caminho que começa e termina no mesmo vértice, ou seja, $v_1 = v_{k+1}$. Formalmente, um ciclo C é uma sequência de vértices $C = (v_1, v_2, \ldots, v_k, v_1)$ tal que cada par consecutivo (v_i, v_{i+1}) é uma aresta em E e $k \geq 2$. O comprimento do ciclo é o número de arestas que ele contém, que é k.

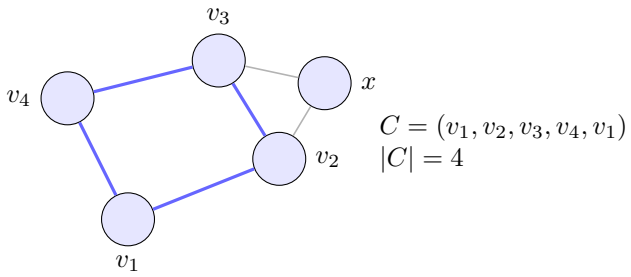


Figura 24: Ciclo em grafo não dirigido: o ciclo $C = (v_1, v_2, v_3, v_4, v_1)$ está destacado em azul. Seu comprimento é o número de arestas, $|C| = 4$.

Componentes conexas

Uma **componente conexa** de um grafo é um subgrafo maximal que é conexo. Formalmente, uma componente conexa C é um subgrafo $C = (V_C, E_C)$ onde $V_C \subseteq V$ e $E_C \subseteq E$, que satisfaz a seguinte propriedade:

- C é conexo: existe um caminho entre qualquer par de vértices em V_C.

Além disso, C é maximal, o que significa que não é possível adicionar mais vértices ou arestas a C sem perder a propriedade de conexidade.

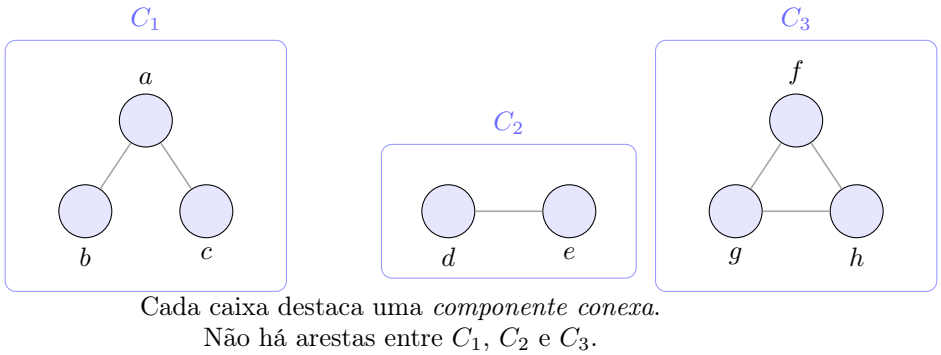


Figura 25: Componentes conexas: o grafo possui três componentes C_1, C_2 e C_3. Cada C_i é conexo e *maximal*, isto é, não pode ser estendido mantendo a conexidade.

Árvores

Uma **árvore** é um grafo conexo e acíclico. Formalmente, uma árvore T é um grafo $T = (V_T, E_T)$ onde $V_T \subseteq V$ e $E_T \subseteq E$, que satisfaz as seguintes propriedades:

- T é conexo: existe um caminho entre qualquer par de vértices em V_T.
- T é acíclico: não contém ciclos.

Além disso, uma árvore com n vértices sempre tem exatamente $n - 1$ arestas.

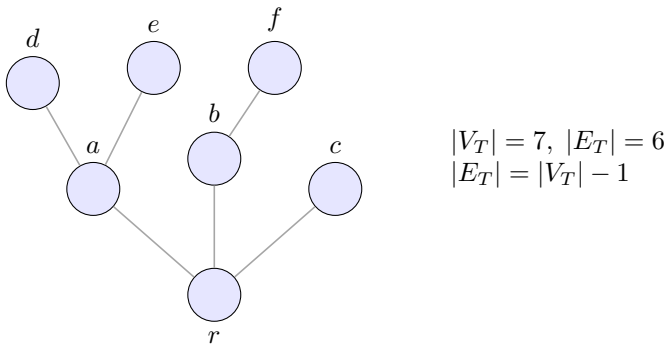


Figura 26: Árvore: grafo conexo e acíclico. No exemplo, $|V_T| = 7$ e $|E_T| = 6$, satisfazendo $|E_T| = |V_T| - 1$. Não há ciclos e existe um único caminho simples entre quaisquer dois vértices.

Para o nosso objetivo principal, nos interessa entendê-las em grafos que a direção das conexões (arestas) importa.

2.5 digrafos: quando a direção importa

Existem problemas que a direção das arestas faz toda a diferença. Por exemplo, em uma rede de tráfego, algumas ruas são de mão única, ou em uma rede de comunicação, os dados podem ser enviados em uma direção específica. Nesses casos, usamos *grafos dirigidos* (ou grafos direcionados ou simplesmente digrafos), onde as arestas são pares de vértices ordenados.

Um **grafo dirigido - digrafo** (grafos direcionados) é uma estrutura matemática composta por um conjunto V de *vértices* e um conjunto A de *arcos* (ou *arestas direcionadas*) que conectam pares ordenados de vértices.

Por vértices entendemos o mesmo conjunto que em grafos comuns, mas agora as arestas entre eles têm uma direção específica. Cada arco $(u, v) \in A$ indica uma conexão direcionada do vértice u para o vértice v, significando que a relação ou fluxo ocorre de u para v.

Assim, temos os conceitos de cabeça e cauda de um arco: em (u, v), u é a *cauda* (origem) e v é a *cabeça* (destino). Esses conceitos podem ser formalizados por meio de funções $s, t : A \to V$, onde $s((u, v)) = u$ (cauda) e $t((u, v)) = v$ (cabeça).

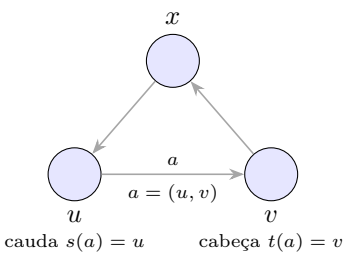


Figura 27: digrafos: arcos têm direção. No arco $a = (u, v)$, u é a *cauda* e v é a *cabeça*.

Em digrafos podem ocorrer laços (arcos que conectam um vértice a ele mesmo, como (u, u)) nesse caso as funções s e t coincidem. Também podem ocorrer múltiplos arcos entre o mesmo par de vértices (como (u, v) e (u, v) distintos). Pictograficamente representamos essas condições com setas com dupla ponta ou com rótulos diferentes.

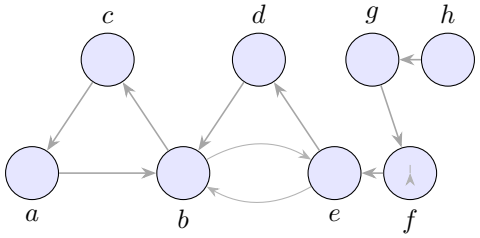
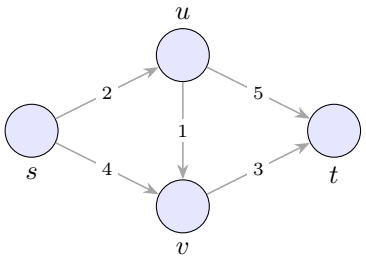


Figura 28: digrafo: exemplo de grafo dirigido $D = (V, A)$. Pontos representam os vértices V e setas representam os arcos A, que são pares ordenados de vértices. Laços (como (f, f)) e múltiplos arcos (como (b, e) e (e, b)) são permitidos.

Tal qual os grafos comuns, os digrafos podem ter custos associados aos arcos. A função de custo $c : A \to \mathbb{R}^+$ atribui um valor numérico geralmente não negativo a cada arco do digrafo. Assim, para cada arco $(u, v) \in A$, $c((u, v))$ representa o custo associado a essa conexão direcionada.



$c : A \to R^+$
$c((s, u)) = 2,\ c((v, t)) = 3, \ldots$
$C(P) = \sum_{a \in P} c(a)$

Figura 29: digrafos com custos nos arcos: a função $c : A \to \mathbb{R}^+$ atribui um custo não negativo a cada arco.

Um conceito importante em digrafos é o de grau de um vértice. O **grau de entrada** (ou *in-degree*) de um vértice v, denotado por $d^-(v)$, é o número de arcos que chegam a v (ou seja, o número de arcos cujo destino é v). O **grau de saída** (ou *out-degree*) de um vértice v, denotado por $d^+(v)$, é o número de arcos que saem de v (ou seja, o número de arcos cuja origem é v). Formalmente, temos:

$$d^-(v) = |\{(u, v) \in A \mid u \in V\}|$$

$$d^+(v) = |\{(v, w) \in A \mid w \in V\}|$$

Esse conceito é útil para analisar conectividade, o que nos leva ao próximo tópico, empacotamento de vértices.

Empacotamento de Vértices

Um **empacotamento de vértices** (conjunto independente) em um digrafo é um conjunto $S \subseteq V$ tal que, no subdigrafo induzido por S, todo vértice tem grau de entrada e de saída iguais a zero. Em notação de graus, se denotamos por $D[S]$ o subdigrafo induzido, então para todo $v \in S$ vale $d^-_{D[S]}(v) = 0$ e $d^+_{D[S]}(v) = 0$. Isso é equivalente a dizer que não existe arco com ambas as extremidades em S (isto é, nenhum $(u, v) \in A$ com $u, v \in S$).

Empacotamento Máximo de Vértices

Um **empacotamento máximo de vértices** é um empacotamento de vértices que contém o maior número possível de vértices. Em outras palavras, é um conjunto $S \subseteq V$ tal que não existem arcos entre vértices em S e S é o maior possível em termos de cardinalidade. Encontrar um empacotamento máximo em um digrafo é um problema NP-difícil, vamos falar sobre o que isso significa na sessão de algoritmos e complexidade.

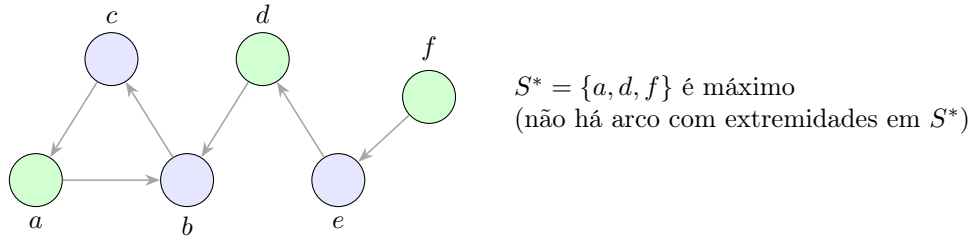


Figura 30: Empacotamento máximo de vértices: para este digrafo, S^* é um conjunto independente de maior cardinalidade.

Esses conceitos de conectividade e empacotamento de vértices nos levam a explorar as subestruturas especiais que existem em digrafos, que são similares às que vimos em grafos comuns, mas com algumas diferenças importantes devido à direção dos arcos.

2.5.1 Subestruturas em digrafos

Tal qual os grafos que discutimos na sessão anterior, os digrafos também possuem as mesmas estruturas especiais, essas estruturas chamadas sub-digrafos mudam um pouco em nomenclatura: caminhos quando direcionados são chamados de trilhas, e ciclos são chamados de circuitos e componentes conexas são componentes fortemente conexas e árvores viram arborescências. Além da nomenclatura, a direção dos arcos traz algumas nuances importantes, discutiremos sobre essas nuances apenas na sessão de arborescências e como os algoritmos de busca mudam bastante em complexidade se estamos tratando de arborescências ou árvores comuns.

Um **subdigrafo** $D' = (V', A')$ de um digrafo $D = (V, A)$ é um digrafo onde $V' \subseteq V$ e $A' \subseteq A$. Ou seja, D' é formado por um subconjunto dos vértices e arcos de D.

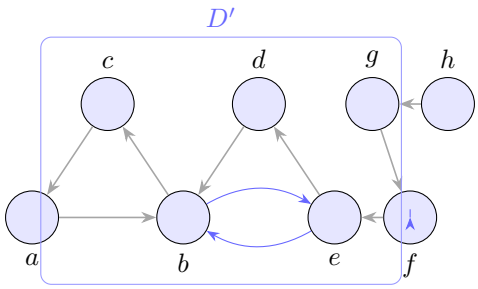


Figura 31: Subdigrafo: o subdigrafo $D' = (V', A')$ está destacado em azul. Aqui, $V' = \{b, c, d, e\}$ e A' contém apenas arcos entre esses vértices.

Subdigrafos Induzidos

Um subdigrafo pode ser *induzido* por um conjunto de vértices $V' \subseteq V$, denotado como $D[V']$. Nesse caso, o conjunto de arcos A' inclui todos os arcos em A que têm ambas as extremidades em V', ou seja, $A' = \{(u, v) \in A \mid u, v \in V'\}$.

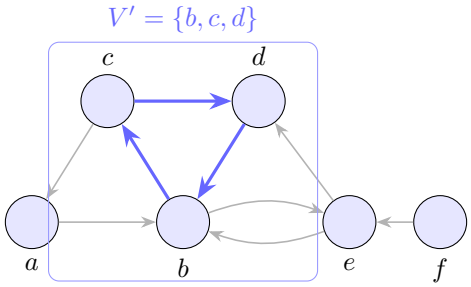


Figura 32: Subdigrafo induzido: para $V' = \{b, c, d\}$, $D[V']$ mantém todos os arcos com ambas as extremidades em V'.

Subdigrafo Maximal

Um subdigrafo é tido como maximal se não é possível adicionar mais vértices ou arcos a ele sem perder alguma propriedade específica, como conexidade ou aciclicidade.

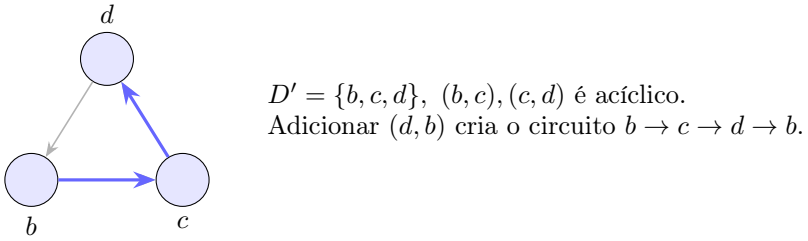


Figura 33: Subdigrafo maximal (por aciclicidade): D' é acíclico e maximal em D; adicionar o arco restante (d, b) cria um circuito.

Subdigrafo Gerador

Um subdigrafo é tido como gerador se inclui todos os vértices do digrafo original, ou seja, $V' = V$. Nesse caso, o subdigrafo é formado por um subconjunto dos arcos do digrafo original.

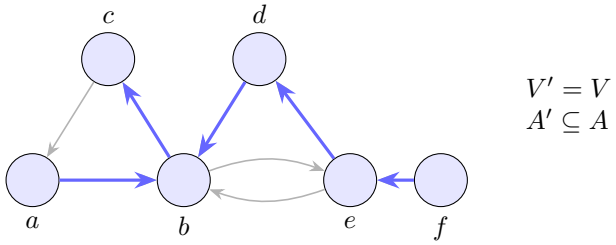


Figura 34: Subdigrafo gerador: inclui todos os vértices do digrafo original ($V' = V$) e apenas um subconjunto dos arcos (em azul).

Com essas definições em mente, podemos explorar as subdigrafos específicos que citaremos ao longo da dissertação, começando pelas trilhas, circuitos, componentes fortemente conexas, componentes-fonte e arborescências.

2.5.2 Subdigrafos Especiais

Trilhas

Uma **trilha** (ou caminho direcionado) em um digrafo é uma sequência de vértices conectados por arcos que respeitam a direção. Formalmente, uma trilha P de comprimento $k \geq 1$ é uma sequência de vértices $P = (v_1, v_2, \ldots, v_{k+1})$ tal que cada par consecutivo (v_i, v_{i+1}) é um arco em A. O comprimento da trilha é o número de arcos que ela contém, que é k.

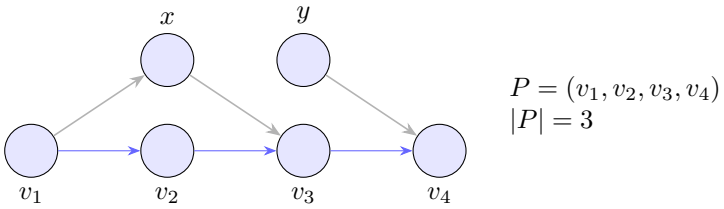


Figura 35: Trilha em digrafo: a trilha $P = (v_1, v_2, v_3, v_4)$ está destacada em azul. Seu comprimento é o número de arcos percorridos, $|P| = 3$.

Uma conceito importante relacionado às trilhas é o de cortes.

Cortes e Min-cortes

Um **corte** em um digrafo é um conjunto de arcos cuja remoção desconecta o digrafo, ou seja, impede que haja uma trilha entre certos pares de vértices. Formalmente, dado um digrafo $D = (V, A)$, um corte C é um subconjunto de arcos $C \subseteq A$ tal que a remoção dos arcos em C resulta em um digrafo $D' = (V, A \setminus C)$ onde não existe mais uma trilha (caminho direcionado) entre pelo menos um par de vértices $u, v \in V$.

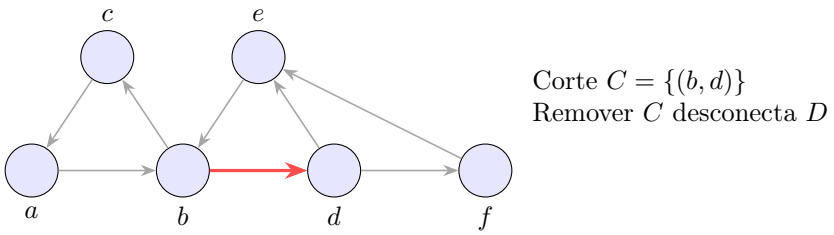


Figura 36: Corte em digrafo: o corte $C = \{(b, d)\}$ remove conectividade de b para d.

De forma resumida, dado um digrafo $D = (V, A)$ e um subconjunto $X \subseteq V$, denotamos por um corte s–t é determinado pela escolha de um conjunto de vértices $X \subseteq V$ tal que $s \in X$ e $t \notin X$; pensa-se nele como a "divisão" do grafo em dois lados: X e $V \setminus X$.

Para tornar a notação precisa e fácil de ler:

- s–t: lê-se "de s para t". Aqui, s é a fonte (onde o fluxo nasce) e t é o sumidouro (onde o fluxo chega).
- $\delta^+(X)$ (fronteira de saída de X): conjunto de todos os arcos que *saem* de X para fora, isto é, para $V \setminus X$.
- $\delta^-(X)$ (fronteira de entrada de X): conjunto de todos os arcos que *entram* em X vindos de $V \setminus X$. Note que $\delta^-(X) = \delta^+(V \setminus X)$.
- **Valor do corte**: dado um peso (ou custo) $c : A \to \mathbb{R}_+$ para cada arco, o valor do corte induzido por X é a soma dos pesos dos arcos que cruzam de X para fora:

$$c(\delta^+(X)) = \sum_{a \in \delta^+(X)} c(a).$$

  No caso não ponderado, esse valor coincide com a *quantidade* de arcos que saem de X.

Exemplo: se $\delta^+(X) = \{(u_1, v_1), (u_2, v_2)\}$ com $c((u_1, v_1)) = 2$ e $c((u_2, v_2)) = 3$, então $c(\delta^+(X)) = 2 + 3 = 5$.

Um min-corte é um corte de tamanho mínimo, ou seja, é o corte com o menor número possível de arcos cuja remoção desconecta o digrafo. Formalmente, dado um digrafo $D = (V, A)$, um min-corte C_{min} é um corte tal que para qualquer outro corte C, $|C_{min}| \leq |C|$. O tamanho do min-corte é o número de arcos em C_{min}.

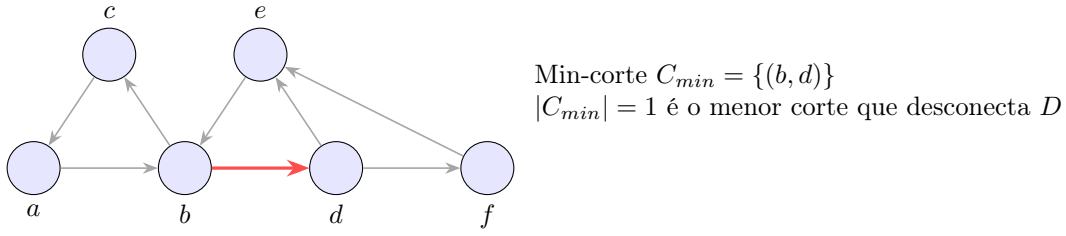


Figura 37: Min-corte em digrafo: o min-corte C_{min} é um corte de menor cardinalidade (ou custo) que separa s de t.

Um teorema importante relacionado a min-cortes é o Teorema do Fluxo Máximo - Corte Mínimo, que estabelece uma relação entre o fluxo máximo que pode ser enviado de uma fonte s para um sumidouro t em um digrafo e o valor do min-corte que separa s de t. Escolhemos apresensentar esse teorema aqui pois ele traz uma intuição interessante sobre a relação entre fluxos e cortes em digrafos, relevantes para os algoritmos que discutiremos mais adiante.

Teorema 2.1: Fluxo–Corte Máximo = Mínimo Corte.

Em digrafos com limites/pesos não negativos nos arcos, o valor de um fluxo máximo de s para t é igual ao valor de um min-corte s–t. Em símbolos: $\max \text{valor}(f) = \min c(\delta^+(X))$, onde o mínimo é sobre $X \subseteq V$ com $s \in X$, $t \notin X$, e $c(\delta^+(X)) = \sum_{a \in \delta^+(X)} c(a)$.

Prova (esboço):

(i) Desigualdade $\leq$. Seja f um fluxo qualquer. Para um corte $(X, V \setminus X)$ com $s \in X$, $t \notin X$, a conservação de fluxo implica que o fluxo líquido que sai de X é exatamente $\text{valor}(f)$.
Logo,

$$\text{valor}(f) = \sum_{a \in \delta^+(X)} f(a) - \sum_{a \in \delta^-(X)} f(a) \leq \sum_{a \in \delta^+(X)} f(a) \leq \sum_{a \in \delta^+(X)} c(a) = c(\delta^+(X)),$$

pois $f(a) \leq c(a)$ para todo arco (limite). Como isso vale para todo X, obtemos $\text{valor}(f) \leq \min_X c(\delta^+(X))$.

(ii) Desigualdade $\geq$ e igualdade. Considere um fluxo máximo f sem caminho aumentante no *grafo residual* R_f (isto é, não há como aumentar o valor do fluxo). Defina X como o conjunto de vértices alcançáveis a partir de s em R_f. Então, não existe arco residual de X para $V \setminus X$; logo, todo arco original que sai de X está saturado ($f(a) = c(a)$), e todo arco que entra em X carrega fluxo zero. Assim,

$$\text{valor}(f) = \sum_{a \in \delta^+(X)} f(a) = \sum_{a \in \delta^+(X)} c(a) = c(\delta^+(X)).$$

Portanto, f atinge exatamente o valor de um corte s–t; em particular, esse corte é mínimo e f é máximo. □

Comentário: Esse resultado é um protótipo de teorema *min–max*: "empacotar" muito fluxo (caminhos) equivale a "cobrir" pouco com um corte. Ver, por exemplo, [21].

Quando falamos em trilhas, precisamos também falar sobre circuitos, que são ciclos direcionados. A diferença entre trilhas e circuitos é que trilhas são caminhos direcionados que não necessariamente retornam ao ponto de origem, enquanto circuitos são caminhos direcionados que começam e terminam no mesmo vértice.

Circuitos

Um **circuito** é um caminho direcionado que começa e termina no mesmo vértice, ou seja, $v_1 = v_{k+1}$. Formalmente, um circuito C é uma sequência de vértices $C = (v_1, v_2, \ldots, v_k, v_1)$ tal que cada par consecutivo (v_i, v_{i+1}) é um arco em A e $k \geq 2$. O comprimento do circuito é o número de arcos que ele contém, que é k.

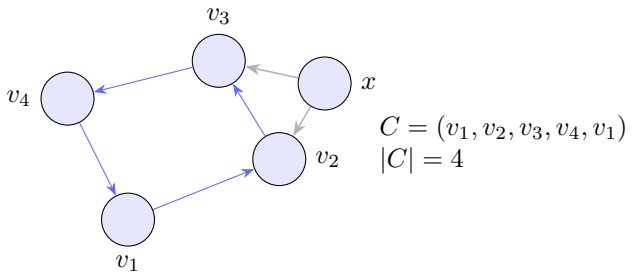


Figura 38: Ciclo direcionado em digrafo: o ciclo $C = (v_1, v_2, v_3, v_4, v_1)$ está destacado em azul. Seu comprimento é o número de arcos, $|C| = 4$.

Propriedades importantes dos circuitos incluem:

- **Circuito simples**: um circuito é dito simples se não repete vértices, exceto o vértice inicial/final. Ou seja, $v_i \neq v_j$ para $1 \leq i < j \leq k$.
- **Circuito Euleriano**: um circuito que percorre cada arco exatamente uma vez. Um digrafo possui um circuito euleriano se e somente se é fortemente conexo e o grau de entrada é igual ao grau de saída para cada vértice.
- **Circuito Hamiltoniano**: um circuito que visita cada vértice exatamente uma vez, exceto o vértice inicial/final. Determinar a existência de um circuito hamiltoniano é um problema que chamamos de NP-completo. Vamos explicar mais sobre isso na seção de algoritmos e complexidade computacional.

Existe um princípio chamado de princípio da casa dos pombos, que diz que se você tem mais pombos do que casas, pelo menos uma casa deve conter mais de um pombo. Em termos de grafos, isso se traduz na ideia de que se um grafo tem mais arestas do que vértices, ele deve conter pelo menos um ciclo.

Esse princípio também se aplica a digrafos, mas com uma nuance importante: em digrafos, o critério correto para garantir a existência de um circuito dirigido é que o grau mínimo de saída (ou de entrada) seja pelo menos 1. Ou seja, se cada vértice em um digrafo tem pelo menos um arco saindo dele (ou entrando nele), então o digrafo deve conter pelo menos um circuito dirigido. Abaixo apresentamos esse resultado formalmente.

Lema 2.1: Princípio da casa dos pombos para circuitos.

Se $D = (V, A)$ é um digrafo finito em que todo vértice tem grau de saída ao menos 1, isto é, $d^+(v) \geq 1$ para todo $v \in V$, então D contém pelo menos um circuito

dirigido. (De forma equivalente, a afirmação vale trocando "saída" por "entrada".)

Prova: Escolha para cada $v \in V$ um arco $(v, f(v))$ que sai de v (possível porque $d^+(v) \geq 1$). Fixado um vértice v_0, considere a sequência $v_0, v_1 = f(v_0), v_2 = f(v_1), \ldots$. Como V é finito, algum vértice repete: existem $i < j$ com $v_i = v_j$. O trecho $v_i \to v_{i+1} \to \cdots \to v_j = v_i$ é um circuito dirigido em D. A versão com graus de entrada segue aplicando o argumento ao digrafo com arcos invertidos. □

Observação: A condição $|A| > |V|$ *não* garante a existência de circuito dirigido em geral (há orientações acíclicas com muitos arcos). O critério correto, simples e útil, é o grau mínimo de saída (ou de entrada) ser pelo menos 1. Ver, por exemplo, [21].

Esse princípio ajuda a entender o comportamento básico dos métodos que os algoritmos empregam para encontrar as arborescências de custo mínimo. Vamos elaborar melhor esse ponto na seção de algoritmos em arborescências, especialmente ao discutir o algoritmo de Chu–Liu/Edmonds.

No algoritmo de Chu–Liu/Edmonds que exploraremos em detalhes em capítulo posterior, para cada vértice $v \neq r$, escolhemos a aresta de menor custo que entra em v, o subgrafo obtido fica com grau de entrada igual a 1 em todos os $v \neq r$. Pelo lema, enquanto esse subgrafo ainda não for uma arborescência, ele necessariamente contém um circuito: assim o algoritmo se baseia em detectar o circuitos, contraí-lo a um único vértice e repetir a seleção sob custos reduzidos. Quando não houver mais circuitos, as arestas escolhidas formam uma arborescência ótima enraizada em r.

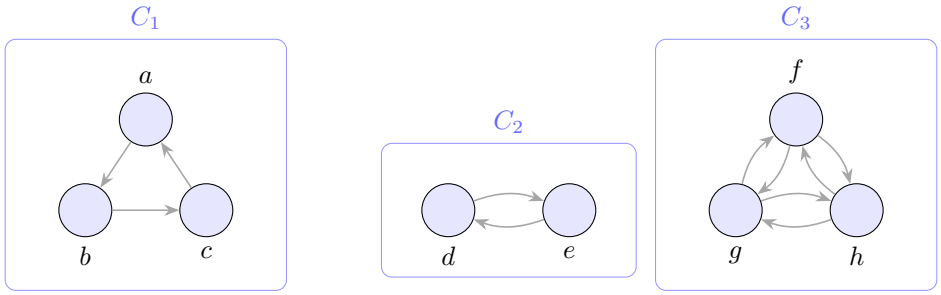
Componentes fortemente conexas

Uma **componente fortemente conexa** (abreviaremos como **CFC**) é um subdigrafo maximal onde existe um caminho direcionado (trilha) entre qualquer par de vértices.

Formalmente, uma componente fortemente conexa C é um subdigrafo $C = (V_C, A_C)$ onde $V_C \subseteq V$ e $A_C \subseteq A$, que satisfaz a seguinte propriedade:

- C é fortemente conexo: existe um caminho direcionado entre qualquer par de vértices em V_C.

Além disso, C é maximal, o que significa que não é possível adicionar mais vértices ou arcos a C sem perder a propriedade de forte conexidade.



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

Figura 39: Componentes fortemente conexas: o digrafo possui três componentes C_1, C_2 e C_3. Cada C_i é fortemente conexo e *maximal*.

A seguir falaremos sobre a propriedade de alcançabilidade mútua em CFC, que é fundamental para entendermos a relação entre elas e arborescências (que apenas mencionaremos aqui, para aprofundarmos na sessão sobre arborescências e com os grafos acíclicos dirigidos (DAGs))[1]:

- **CFCs e alcançabilidade mútua:** dois vértices u e v pertencem à mesma CFC se, e somente se, $u \rightsquigarrow v$ e $v \rightsquigarrow u$. Ou seja, existe um caminho direcionado de u para v e um caminho direcionado de v para u. A relação de pertencer à mesma CFC é uma relação de equivalência que particiona o conjunto de vértices V em subconjuntos disjuntos, cada um correspondendo a uma CFC.

- **Arborescências em digrafos fortemente conexos:** um digrafo é fortemente conexo se, e somente se, para algum (equiv., para todo) vértice r existem uma *arborescência de saída* enraizada em r que alcança todos os vértices e uma *arborescência de entrada* enraizada em r que é alcançada por todos os vértices. De fato, se D é fortemente conexo, basta rodar buscas a partir de r; no sentido inverso, $r \rightsquigarrow v$ pela arborescência de saída e $v \rightsquigarrow r$ pela de entrada, implicando alcançabilidade mútua entre quaisquer dois vértices via r.

- **CFC e DAG.** Ao *contrair* cada CFC a um único vértice obtemos o grafo condensado $\mathrm{Cond}(D)$[2]. Não há circuitos dirigidos em $\mathrm{Cond}(D)$; portanto, ele é um DAG.

  Consequências: todo DAG tem ao menos uma componente-fonte (falaremos em seguida sobre eles) e uma componente-sumidouro; logo, $\mathrm{Cond}(D)$ também tem ao menos uma CFC-fonte e ao menos uma CFC-sumidouro.

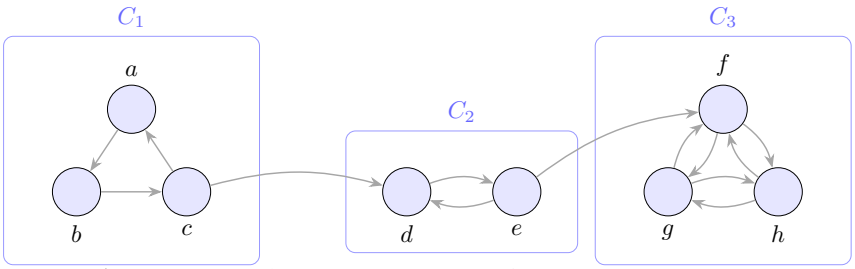
Componentes-fonte

[1] Um DAG (Directed Acyclic Graph) é um grafo direcionado que não contém ciclos. Em outras palavras, não é possível começar em um vértice e seguir uma sequência de arcos que retorne ao mesmo vértice. Os DAGs são estruturas fundamentais em muitas áreas da computação, incluindo a representação de dependências e a modelagem de processos.

[2] O grafo condensado é uma representação simplificada do digrafo original, onde as CFCs são representadas como vértices únicos.

Uma **componente-fonte** é uma componente fortemente conexa que não possui arcos direcionados saindo dela para outras componentes. Formalmente, uma componente-fonte C é uma componente fortemente conexa $C = (V_C, A_C)$ onde $V_C \subseteq V$ e $A_C \subseteq A$, que satisfaz a seguinte propriedade:

- Não existem arcos $(u, v) \in A$ tais que $u \in V_C$ e $v \notin V_C$.
- C é maximal: não é possível adicionar mais vértices ou arcos a C sem perder a propriedade de forte conexidade.
- C é fortemente conexo: existe um caminho direcionado entre qualquer par de vértices em V_C.



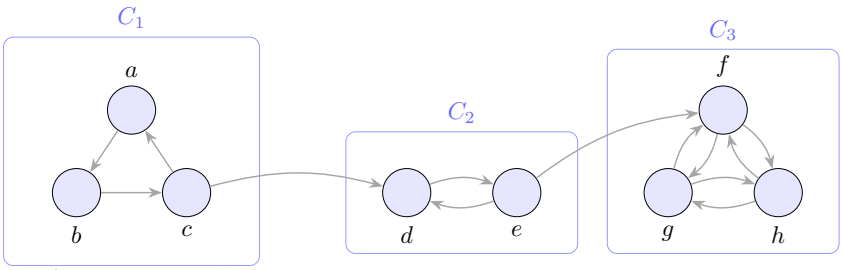
A componente C_1 é uma *componente-fonte* porque não há arcos saindo dela para outras componentes. Já C_2 e C_3 não são componentes-fonte, pois há arcos entrando nelas.

Figura 40: Componente-fonte: o digrafo possui três componentes C_1, C_2 e C_3. A componente C_1 é uma componente-fonte porque não há arcos saindo dela para outras componentes. Já C_2 e C_3 não são componentes-fonte, pois há ar cos entrando nelas.

Componentes-sumidouro

Uma **componente-sumidouro** é uma componente fortemente conexa que não possui arcos direcionados entrando nela vindos de outras componentes. Formalmente, uma componente-sumidouro C é uma componente fortemente conexa $C = (V_C, A_C)$ onde $V_C \subseteq V$ e $A_C \subseteq A$, que satisfaz a seguinte propriedade:

- Não existem arcos $(u, v) \in A$ tais que $u \notin V_C$ e $v \in V_C$.
- C é maximal: não é possível adicionar mais vértices ou arcos a C sem perder a propriedade de forte conexidade.
- C é fortemente conexo: existe um caminho direcionado entre qualquer par de vértices em V_C.



A componente C_3 é uma *componente-sumidouro* porque não há arcos entrando nela vindos de outras componentes. Já C_1 e C_2 não são componentes-sumidouro, pois há arcos saindo delas.

Figura 41: Componente-sumidouro: o digrafo possui três componentes C_1, C_2 e C_3. A componente C_3 é uma componente-sumidouro porque não há arcos entrando nela vindos de outras componentes. Já C_1 e C_2 não são componentes-sumidouro, pois há arcos saindo delas.

Existe um resultado clássico que relaciona o número de componentes-fonte e componentes-sumidouro em um digrafo com o número mínimo de arcos necessários para torná-lo fortemente conexo.

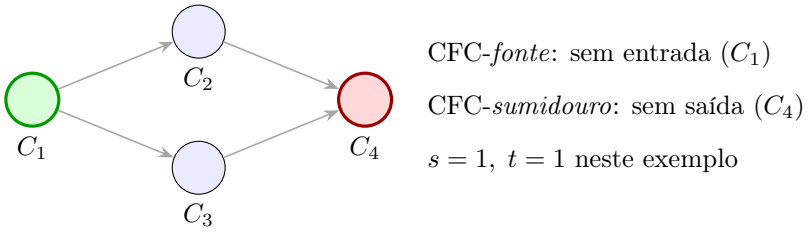


Figura 42: Grafo condensado Cond(D): cada CFC é contraída a um vértice e não há circuitos dirigidos (DAG).

Seja $D = (V, A)$ um digrafo com k componentes fortemente conexas (CFCs):

Denote por Cond(D) o grafo *condensado*, obtido ao contrair cada CFC de D em um único vértice; Cond(D) é sempre um DAG. Escreva s para o número de CFCs-*fonte* (vértices de Cond(D) sem arcos de *entrada*) e t para o número de CFCs-*sumidouro* (vértices de Cond(D) sem arcos de *saída*). Com essa notação, vale o seguinte:

Lema 2.2: Mínimo de arcos para conexidade forte

Se $k = 1$, então D já é fortemente conexo e o mínimo de arcos a adicionar é 0. Se $k \geq 2$, o número mínimo de arcos a adicionar para tornar D fortemente conexo é

$$\boxed{\max\{s,\, t\}}\,.$$

Prova (esboço).

Necessidade: em qualquer supergrafo fortemente conexo, cada CFC-fonte deve receber ao menos um arco *entrando* e cada CFC-sumidouro deve ter ao menos um

arco *saindo*; logo, são necessários ao menos $\max\{s, t\}$ arcos novos.

Suficiência: numa ordenação topológica de Cond(D), conecte CFCs-sumidouro a CFCs-fonte de modo a formar um ciclo que percorra todas as CFCs. Isso pode ser feito com exatamente $\max\{s, t\}$ arcos, mesmo quando $s \neq t$, emparelhando sobras de um lado com o outro.

Ver, por exemplo, textos clássicos sobre digrafos (e.g., [21]).

Esse lema explica por que todo digrafo com mais de uma CFC-fonte ou mais de uma CFC-sumidouro não pode ser fortemente conexo. Além disso, ele é útil em algoritmos que buscam tornar um digrafo fortemente conexo, pois fornece um limite inferior para o número de arcos que precisam ser adicionados.

As noções de CFCs e componentes-fonte conversam diretamente com o conceito de *arborescências*. Pois, em uma arborescência, todos os vértices são alcançáveis a partir da raiz, o que não implica que a arborescência é fortemente conexa, mas encontrar essas componentes dentro de um digrafo pode nos ajudar em uma busca otimizada por arborescências.

Arborescências

Uma **arborescência** é um digrafo acíclico e conexo, onde há um vértice especial chamado *raiz* que tem um caminho direcionado para todos os outros vértices. Formalmente, uma arborescência T é um digrafo $T = (V_T, A_T)$ onde $V_T \subseteq V$ e $A_T \subseteq A$, que satisfaz as seguintes propriedades:

- T é conexo: existe um caminho direcionado da raiz para qualquer vértice em V_T.
- T é acíclico: não contém ciclos direcionados.

Além disso, uma arborescência com n vértices sempre tem exatamente $n - 1$ arcos.

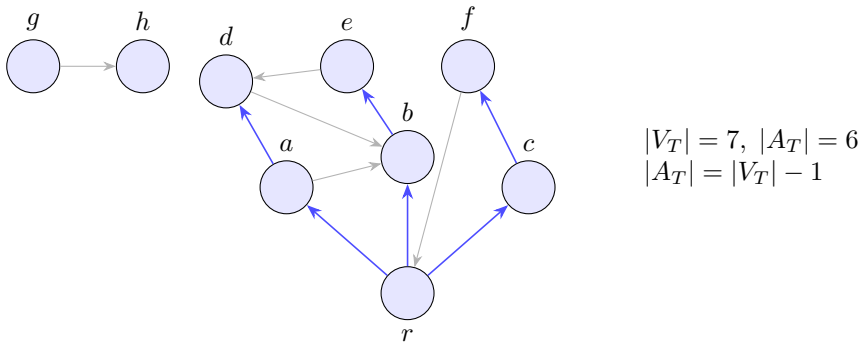


Figura 43: Arborescência: digrafo conexo e acíclico com raiz r de onde há um caminho direcionado para todos os outros vértices em azul. No exemplo, $|V_T| = 7$ e $|A_T| = 6$, satisfazendo $|A_T| = |V_T| - 1$. Em cinza, arcos que não fazem parte da arborescência.

Definições e notação adicionais

Para facilitar a discussão sobre arborescências, introduzimos algumas definições e notações adicionais: seja $D = (V, A)$ um digrafo e $r \in V$ um vértice específico (a raiz). Denotamos por $d_D^+(v)$ o grau de saída de um vértice v em D, ou seja, o número de arcos que saem de v. Analogamente, $d_D^-(v)$ é o grau de entrada de v, o número de arcos que entram em v.

As arborescências são o principal objeto de investigação desse trabalho, portanto vamos usar uma sessão dedicada a elas para apresentar suas características, variações e aplicações.

2.6 Arborescências em foco

Já tratamos do conceito básico de arborescência, agora falaremos de arborescências especiais:

Arborescência Geradora: Uma arborescência é considerada geradora se inclui todos os vértices do digrafo original, ou seja, $V_T = V$. Nesse caso, a arborescência é formada por um subconjunto dos arcos do digrafo original.

Arborescência Maximal: Uma arborescência é dita maximal se não é possível adicionar mais vértices ou arcos a ela sem perder a propriedade de ser uma arborescência, ou seja, sem criar ciclos ou desconectar o digrafo.

Ramificações Geradoras

Uma **ramificação geradora** é um subdigrafo que é uma arborescência que inclui todos os vértices do digrafo original. Formalmente, uma ramificação geradora R é um subdigrafo $R = (V_R, A_R)$ onde $V_R = V$ e $A_R \subseteq A$, que satisfaz as seguintes propriedades:

- R é uma arborescência: existe um vértice especial chamado raiz que tem um caminho direcionado para todos os outros vértices.
- R é maximal: não é possível adicionar mais arcos a R sem perder a propriedade de ser uma arborescência.

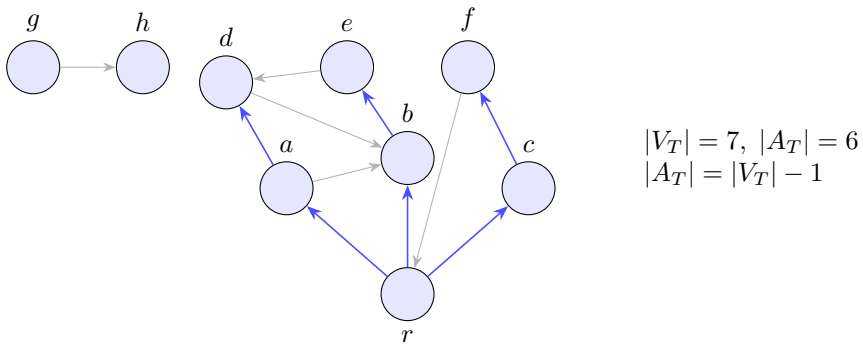


Figura 44: Ramificação geradora: arborescência que inclui todos os vértices do digrafo original, em azul. No exemplo, $|V_T| = 7$ e $|A_T| = 6$, satisfazendo $|A_T| = |V_T| - 1$. Em cinza, arcos que não fazem parte da ramificação geradora.

Quando falamaos de ramificações geradoras, podemos falar de uma estrutura que fixa um vértice raiz r e constrói uma arborescência que alcança todos os outros vértices a partir dessa raiz. Essa estrutura é conhecida como *r-arborescência*.

Arborescência de Raiz Específica: uma arborescência de raiz específica é uma arborescência onde a raiz é um vértice pré-determinado do digrafo. Isso é útil em situações onde um ponto de origem específico deve ser o início dos caminhos direcionados para todos os outros vértices. Podemos chamá-la de r-arborescência, onde r é o vértice raiz.

Em uma arborescência $T = (V_T, A_T)$ enraizada em r, temos as seguintes propriedades:

- A raiz r tem grau de entrada zero: $d_T^-(r) = 0$.
- Todo outro vértice $v \in V_T \setminus \{r\}$ tem grau de entrada exatamente um: $d_T^-(v) = 1$. Isso significa que há exatamente um arco direcionado entrando em cada vértice, exceto na raiz.
- O grau de saída $d_T^+(v)$ pode variar, mas para garantir que T seja conexo, deve haver pelo menos um arco saindo de r para alcançar os outros vértices.

Arborescência inversa (in-arborescência): uma *arborescência inversa* enraizada em r — também chamada de *in-arborescência* — é o resultado de inverter a orientação de todos os arcos de uma arborescência (out-arborescência) enraizada em r. Equivalentemente: é um digrafo acíclico no qual, para todo $v \neq r$, existe *exatamente um* caminho direcionado de v até r (isto é, todos os arcos estão orientados *em direção* à raiz). Em termos de graus, numa in-arborescência cada vértice $v \neq r$ tem grau de saída igual a 1 (o arco para seu "pai") e a raiz r tem grau de saída 0; os graus de entrada são complementares aos de uma out-arborescência.

As arborecências podem ter custos associados aos seus arcos, o que nos leva ao conceito de arborescência de custo mínimo.

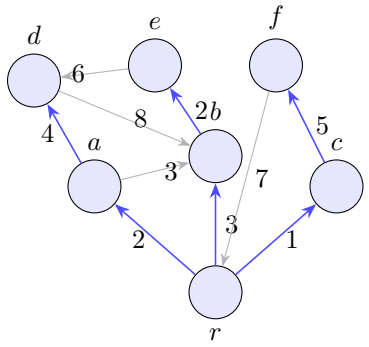
Arborescência de Custo Mínimo:

Uma **arborescência de custo mínimo** é uma arborescência que minimiza a soma dos pesos dos arcos que a compõem. Esse conceito é especialmente relevante em aplicações onde os arcos têm custos associados, como em redes de transporte ou comunicação.

Finalmente podemos conceituar a principal estrutura que estudaremos nesta dissertação: a r-arborescência de custo mínimo e sua variante, a r-arborescência inversa de custo mínimo.

r-arborescência de custo mínimo: é uma arborescência enraizada em um vértice específico r que minimiza a soma dos pesos dos arcos que a compõem. Formalmente, dada uma função de custo $c : A \to \mathbb{R}_{\geq 0}$ que atribui um custo a cada arco do digrafo $D = (V, A)$, uma r-arborescência de custo mínimo T é uma arborescência $T = (V_T, A_T)$ onde $V_T \subseteq V$ e $A_T \subseteq A$, que satisfaz as seguintes propriedades:

- T é uma arborescência enraizada em r: existe um caminho direcionado de r para qualquer vértice em V_T.
- T minimiza o custo total: a soma dos custos dos arcos em A_T é mínima, ou seja, $\sum_{a \in A_T} c(a)$ é minimizada.

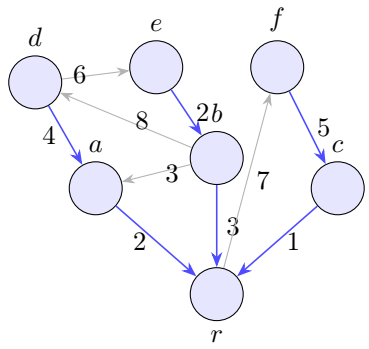


Custo total da r-arborescência: $2 + 3 + 1 + 4 + 2 + 5 = 17$

Figura 45: r-arborescência de custo mínimo: arborescência enraizada em r que minimiza a soma dos custos dos arcos, em azul. No exemplo, o custo total é 17. Em cinza, arcos que não fazem parte da r-arborescência de custo mínimo.

r-arborescência inversa de custo mínimo: é uma arborescência inversa enraizada em um vértice específico r que minimiza a soma dos pesos dos arcos que a compõem. Formalmente, dada uma função de custo $c : A \to \mathbb{R}_{\geq 0}$ que atribui um custo a cada arco do digrafo $D = (V, A)$, uma r-arborescência inversa de custo mínimo T é uma arborescência inversa $T = (V_T, A_T)$ onde $V_T \subseteq V$ e $A_T \subseteq A$, que satisfaz as seguintes propriedades:

- T é uma arborescência inversa enraizada em r: existe um caminho direcionado de qualquer vértice em V_T até r.
- T minimiza o custo total: a soma dos custos dos arcos em A_T é mínima, ou seja, $\sum_{a \in A_T} c(a)$ é minimizada.



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

Figura 46: r-arborescência inversa de custo mínimo: arborescência inversa enraizada em r que minimiza a soma dos custos dos arcos, em azul. No exemplo, o custo total é 17. Em cinza, arcos que não fazem parte da r-arborescência inversa de custo mínimo.

As arborescências são a principal estrutura que exploraremos ao longo desta dissertação, especialmente a r-arborescência de custo mínimo e r-arborescência inversa de custo mínimo, abordaremos o problema de encontrá-las eficientemente em digrafos com custos associados aos arcos.

Noções aprofundadas em arborescências

Vamos explorar algumas propriedades e teoremas importantes relacionados a arborescências, que serão úteis para entender os algoritmos que discutiremos posteriormente.

Grau e contagem de arcos:

Seja T uma out-arborescência enraizada em r com n vértices. Ela é exatamente a estrutura que satisfaz as três condições combinadas abaixo (todas muito fáceis de checar):

1. (Contagem) $|A_T| = n - 1$.
2. (Entrada única) Cada vértice $v \neq r$ recebe exatamente um arco: $d_T^-(v) = 1$.
3. (Raiz sem entrada) A raiz não recebe arcos: $d_T^-(r) = 0$.

De forma simétrica, numa in-arborescência (arborescência inversa) valem as versões "espelhadas": cada $v \neq r$ tem exatamente um arco *saindo* ($d_T^+(v) = 1$) e a raiz tem grau de saída zero ($d_T^+(r) = 0$).

Reciprocamente, qualquer subdigrafo que satisfaça (1)–(3) é uma out-arborescência enraizada em r (e análogamente no caso inverso).

Discussões importantes sobre arborescências

Dado um digrafo $D = (V, A)$ e um vértice raiz $r \in V$, uma questão fundamental é determinar quando existe uma arborescência enraizada em r. Existem alguns resultados clássicos que caracterizam a existência de arborescências em digrafos, bem como condições para a existência de múltiplas arborescências disjuntas. Vamos apresentar dois teoremas fundamentais nesse contexto.

Teorema de Fulkerson

Existem várias formas de caracterizar a existência de arborescências em um digrafo. Uma delas é via a condição de cortes, que estabelece uma relação entre a existência de arborescências e a estrutura dos cortes no digrafo.

Esse resultado é conhecido como o **Teorema de Fulkerson** e para entendermos ele precisamos ter em mente as seguintes definições:

- Seja $D = (V, A)$ um digrafo e $X \subseteq V$ um subconjunto de vértices. O conjunto $\delta^-(X)$ é definido como o conjunto de todos os arcos que entram em X vindos de $V \setminus X$. Formalmente,

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

- Um corte em um digrafo é uma partição dos vértices em dois subconjuntos disjuntos. O conjunto $\delta^-(X)$ representa o corte que separa X do resto do grafo.

A seguir apresentamos o teorema propriamente dito e um esboço de sua prova.

Teorema 2.2: Condição de existência via cortes (Fulkerson)

Seja $D = (V, A)$ e $r \in V$. Existe uma out-arborescência (arborescência dirigida) enraizada em r se, e somente se,

$$\forall\, X \subseteq V \setminus \{r\},\ X \neq \emptyset :\ \delta^-(X) \neq \emptyset.$$

Isto é: todo subconjunto não vazio que não contém a raiz recebe ao menos um arco vindo de fora.

Prova (esboço):

(Só se:) Suponha que T é uma out-arborescência enraizada em r. Pegue qualquer $X \neq \emptyset$ sem r. Considere o primeiro vértice de X alcançado a partir de r no caminho dentro de T; o arco imediatamente anterior entra em X e pertence a $\delta^-(X)$. Logo $\delta^-(X) \neq \emptyset$.

(Se:) Agora suponha que toda parte X não vazia sem r recebe um arco. Construímos T iterativamente: comece com $S = \{r\}$. Enquanto $S \neq V$, tome um vértice $v \in V \setminus S$ tal que existe um arco (u, v) com $u \in S$ (existe porque, caso contrário, o conjunto $X = V \setminus S$ não receberia arco). Adicione v e o arco (u, v).

Não criamos ciclos porque cada novo vértice entra com exatamente um arco e só aponta para frente (a direção é de um vértice já inserido para um novo). Ao final, cada $v \neq r$ tem exatamente um arco de entrada e o grafo é conexo a partir de r, logo obtivemos uma out-arborescência.

Intuição curta. A condição "todo X tem um arco entrando" impede que qualquer bloco de vértices fique isolado da raiz; o processo guloso de anexar o primeiro arco que entra em cada bloco produz a arborescência sem retrocessos.

Referência: ver, por exemplo, [21].

Outro resultado clássico é o teorema que caracteriza a existência de múltiplas arborescências arcodisjuntas em um digrafo, conhecido como o **Teorema de Edmonds**. Precisamos de algumas definições antes de enunciá-lo:

- Duas arborescências são ditas *arcodisjuntas* se não compartilham nenhum arco, ou seja, $A_{T_1} \cap A_{T_2} = \emptyset$.
- A condição de cortes para múltiplas arborescências estabelece que, para qualquer subconjunto $X \subseteq V \setminus \{r\}$, o número de arcos que entram em X deve ser pelo menos igual ao número de arborescências desejadas.
- Uma out-arborescência enraizada em r é uma arborescência onde todos os caminhos direcionados partem de r e alcançam todos os outros vértices.
- O conjunto $\delta^-(X)$ é definido como o conjunto de todos os arcos que entram em X vindos de $V \setminus X$. Formalmente,
$$\delta^-(X) = \{(u, v) \in A : u \in V \setminus X, v \in X\}.$$
- Um corte em um digrafo é uma partição dos vértices em dois subconjuntos disjuntos. O conjunto $\delta^-(X)$ representa o corte que separa X do resto do grafo.

Antes de enunciar o teorema, vale a pena mencionar o conceito de *interseção de matroides*, mas, para não alongar demais, deixamos a explicação detalhada para o Apêndice A. Aqui, apenas uma breve introdução:

- Matroides, são estruturas combinatórias que generalizam a noção de independência linear em álgebra linear. A interseção de matroides é um conceito que permite combinar duas ou mais estruturas de matroides para formar uma nova estrutura que mantém certas propriedades de independência.
- A interseção de matroides é frequentemente utilizada em problemas de otimização combinatória, onde é necessário encontrar soluções que satisfaçam múltiplas condições de independência simultaneamente.
- Familia de conjuntos independentes: cada matroide é definido por uma coleção de subconjuntos de um conjunto finito, chamados de conjuntos independentes, que satisfazem certas propriedades.
- No contexto de arborescências, a interseção de matroides pode ser usada para modelar a seleção de arcos que formam múltiplas arborescências arcodisjuntas, garantindo que cada arborescência mantenha suas propriedades de independência.

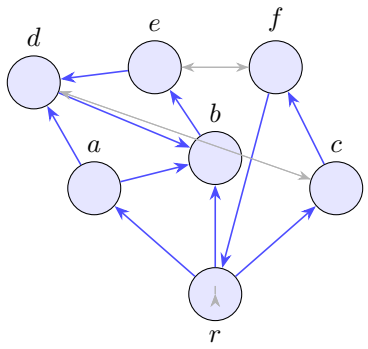


Figura 47: digrafo de exemplo para múltiplas arborescências arcodisjuntas.

Agora podemos enunciar o teorema de Edmonds, que fornece uma condição necessária e suficiente para a existência de k arborescências arcodisjuntas enraizadas em um vértice r.

Teorema 2.3: k arborescências arcodisjuntas (Edmonds)

Seja $D = (V, A)$, $r \in V$ e $k \geq 1$ inteiro. São equivalentes:

1. Existem k out-arborescências enraizadas em r que são par a par *arcodisjuntas*.
2. (Condição de cortes) Para todo subconjunto $X \subseteq V \setminus \{r\}$ vale $|\delta^-(X)| \geq k$.

Em palavras: cada "bloco" X que não contém a raiz precisa ter pelo menos k arcos distintos chegando de fora; isso é exatamente o que permite "alimentar" X a partir de r em k estruturas de ramificação independentes.

Prova (esboço): *(1 ⇒ 2)* Se temos k out-arborescências arcodisjuntas, então cada arborescência deve entrar em qualquer X (senão não alcançaria seus vértices). Como os arcos são distintos entre as k estruturas, precisamos de pelo menos k arcos entrando em X; logo $|\delta^-(X)| \geq k$.

(2 ⇒ 1) Trata-se a construção como um problema de *interseção de matroides* ou aplicamos um procedimento incremental de troca ("*augmenting*"). O conjunto de arcos pode suportar no máximo $k(n-1)$ arcos selecionados se quisermos k arborescências, onde cada vértice $v \neq r$ recebe exatamente k arcos de entrada (um de cada arborescência). A condição de cortes impede gargalos: se algum subconjunto X tivesse menos que k arcos entrando, seria impossível abastecer seus vértices com k escolhas independentes.

Uma prova clássica (Edmonds) formula o problema como interseção de duas famílias independentes:

(i) uma família que limita a quantidade de arcos entrando em cada vértice a no máximo k;

(ii) uma família que evita a criação de ciclos dirigidos ao selecionar arcos (estrutura de matroide de partição + matroide gráfico orientado).

A hipótese de cortes garante que o algoritmo de aumento (que tenta adicionar um arco e, se criar ciclo ou saturar um vértice, realiza trocas) nunca fica travado antes de atingir $k(n-1)$ arcos. Agrupando, particionamos esses $k(n-1)$ arcos em k coleções de $(n-1)$ arcos cada, que formam as k out-arborescências arcodisjuntas.

Referências: Edmonds (teorema das branchings) [6], apresentações modernas em [21].

Teoremas como esses servem para responder perguntas do tipo "quando existe?" e "quão rica pode ser a estrutura?". Em particular, eles nos dizem que:

- A existência de uma arborescência enraizada em r é garantida se, e somente se, todo subconjunto não vazio que não contém r recebe pelo menos um arco vindo de fora (Teorema de Fulkerson).
- A existência de k arborescências arcodisjuntas enraizadas em r é garantida se, e somente se, todo subconjunto não vazio que não contém r recebe pelo menos k arcos vindos de fora (Teorema de Edmonds).

Mas, agora estamos interessados em achar essas arborescências de forma eficiente, especialmente quando os arcos têm custos associados. Queremos encontrar a r-arborescência de custo mínimo, ou seja, a arborescência enraizada em r que minimiza a soma dos custos dos arcos que a compõem.

Um resultado central agora é a caracterização de *optimalidade* para r-arborescências de custo mínimo: as chamadas *condições de Fulkerson*. Elas conectam a solução primal (os arcos escolhidos) a um certificado dual (potenciais em subconjuntos) via custos reduzidos.

Terminologia:

Para um digrafo $D = (V, A)$, raiz r e custos $c : A \to \mathbb{R}_{\geq 0}$, um subconjunto não vazio $X \subseteq V \setminus \{r\}$ é dito **apertado** (para uma família de pesos y) se exatamente um arco da solução escolhida entra em X e $y(X) > 0$.

Diremos que um arco $a = (u, v)$ *entra* em X se $u \notin X$ e $v \in X$.

Dada uma família de pesos $y : \{X \subseteq V \setminus \{r\} : X \neq \emptyset\} \to \mathbb{R}_{\geq 0}$, definimos o **custo reduzido** de a por

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

Teorema 2.4: Optimalidade de Fulkerson (r-arborescência de custo mínimo)

Seja $D = (V, A)$, raiz r e custos $c : A \to \mathbb{R}_{\geq 0}$. Seja T uma out-arborescência enraizada em r. As afirmações são equivalentes:

1. T tem custo mínimo entre todas as out-arborescências enraizadas em r.
2. Existem pesos $y(X) \geq 0$ para cada $\emptyset \neq X \subseteq V \setminus \{r\}$ tais que:
   (a) $c'(a) \geq 0$ para todo arco $a \in A$ (não negatividade dos custos reduzidos);
   (b) $c'(a) = 0$ para todo arco $a \in T$ (complementaridade em arcos usados);
   (c) Para todo X com $y(X) > 0$ entra *exatamente um* arco de T em X (complementaridade em conjuntos apertados).

Além disso, quando (2) vale, o valor $\sum_X y(X)$ é exatamente o custo de T.

Prova (esboço):

(2 ⇒ 1). Para qualquer arborescência B temos

$$\textbf{custo}(B) = \sum_{a \in B} c(a) = \sum_{a \in B} \Big(c'(a) + \sum_{X : a \text{ entra } X} y(X) \Big).$$

Trocando a ordem da soma:

$$\textbf{custo}(B) = \sum_{a \in B} c'(a) + \sum_{X} y(X) \, \big| \{ a \in B : a \text{ entra } X \} \big|.$$

Pelas condições, $c'(a) \geq 0$, logo a primeira soma é ≥ 0. Como uma out-arborescência entra em qualquer $X \neq \emptyset$ (senão X estaria desconectado de r), temos $|\{a \in B : a \text{ entra } X\}| \geq 1$. Assim

$$\textbf{custo}(B) \geq \sum_{X} y(X).$$

Para $B = T$, pela complementaridade $c'(a) = 0$ se $a \in T$ e para cada X com $y(X) > 0$ entra *exatamente* um arco de T, obtendo

$$\textbf{custo}(T) = 0 + \sum_{X} y(X),$$

logo $\textbf{custo}(T) \leq \textbf{custo}(B)$ para qualquer B; T é ótimo.

(1 ⇒ 2). (Ideia) Execute o procedimento clássico: enquanto houver vértice (ou componente contraída) $v \neq r$ sem arco de custo reduzido zero entrando, subtraia do custo de todos os arcos que entram em v o menor custo positivo entre eles (isso equivale a aumentar uniformemente $y(X)$ para cada subconjunto X cujo primeiro arco zero estamos "criando"). Quando um ciclo de arcos de custo reduzido zero surge, contraia-o e continue no digrafo comprimido. Ao final, os arcos de custo reduzido zero selecionados formam T. As quantidades subtraídas definem y: cada vez que subtraímos $\alpha > 0$ para um subconjunto/componente X, somamos α a $y(X)$. Construção garante (a)–(c).

Intuição. Os pesos y "pagam" parcialmente cada arco de fora para dentro dos subconjuntos; arcos da solução ficam exatamente "quitados" (custo reduzido 0). Se algum arco restante tivesse custo reduzido negativo, poderíamos baixar o custo da solução trocando-o por um arco de T, contradizendo optimalidade. Conjuntos com $y(X) > 0$ exigem uso único de um arco para não desperdiçar potencial.

Referências: Fulkerson (condições de optimalidade), apresentações modernas em [7, 21].

O teorema anterior nos diz como reconhecer, de forma concreta, que a arborescência encontrada é realmente de custo mínimo. Fazemos uma normalização simples de custos[3]: para cada "parte" do grafo, subtraímos, dos arcos que entram nessa parte, o menor custo observado; com isso, pelo menos um arco que entra em cada parte zera. A solução ótima pode ser construída usando apenas arcos com custo reduzido zero e, sob esse ajuste, não sobra nenhuma troca que diminua o custo.

Na prática, a verificação de optimalidade se reduz a checar condições locais:

- não há arcos com custo reduzido negativo;
- todo arco que compõe a arborescência tem custo reduzido zero;
- para cada conjunto "apertado" (isto é, que recebeu desconto positivo no ajuste), entra exatamente um arco da arborescência.

Se alguma dessas condições falhar, existe uma troca que barateia a solução; se todas forem satisfeitas, temos um certificado de optimalidade curto e fácil de verificar.

Com essa ideia em mãos, saímos do "o que é ótimo?" para "como chegar lá, passo a passo?". No próximo capítulo apresentamos a noção de algoritmo que adotaremos e descrevemos os métodos clássicos para este problema: o algoritmo de Chu–Liu/Edmonds (que cria arcos de custo zero e contrai ciclos) e o procedimento em duas fases de András Frank. Veremos as etapas, a intuição por trás e como vamos implementá-los no projeto.

2.7 Algoritmos

Quando falamos em passo a passo é muito comum vir à mente a ideia de receitas de cozinha, instruções de montagem ou manuais de operação. Em ciência da computação, o termo *algoritmo* captura essa ideia de forma mais formal e precisa.

O primeiro uso documentado do termo "algoritmo" em inglês data de 1230, em uma tradução latina do trabalho de Al-Khwarizmi. No entanto, o conceito de algoritmos é muito mais antigo, remontando a procedimentos matemáticos e lógicos desenvolvidos ao longo dos séculos.

[3]Por "normalização de custos" entendemos subtrair a mesma constante de todos os arcos que entram em um mesmo subconjunto (ou componente) do grafo, para simplificar os valores e criar arcos de custo reduzido zero, sem alterar qual solução é ótima; em outras palavras, trabalhar com custos reduzidos.

Um dos primeiros algoritmos conhecidos é o *método de Euclides* para encontrar o máximo divisor comum (mdc) de dois números inteiros, descrito por Euclides em sua obra "Os Elementos" por volta de 300 a.C.

Algoritmo 2.1: Método de Euclides (mdc)

Dados inteiros positivos a e b:

1. enquanto $b > 0$, substitua (a, b) por $(b, a \bmod b)$;
2. quando $b = 0$, devolva a.

Mas, o que diferencia uma mera receita de um algoritmo? A resposta está na clareza, precisão e capacidade de execução repetitiva das instruções. Um algoritmo deve ser:

- **Não ambiguidade**: cada passo deve ser definido de maneira inequívoca, sem ambiguidade.
- **Especificidade**: as instruções devem ser detalhadas o suficiente para que possam ser seguidas sem interpretação subjetiva.
- **Executabilidade**: deve ser possível executar o algoritmo de forma sistemática, sem necessidade de criatividade ou intuição.

Por isso, que chamamos o método de Euclides de algoritmo: os passos são não ambíguos; termina porque a segunda coordenada diminui até zerar; é correto pois mantém o invariante $\gcd(a, b) = \gcd(b, a \bmod b)$; e o custo é baixo (proporcional ao número de dígitos de a e b).

Além dessas características, precisamos citar mais alguns conceitos úteis na análise de algoritmos:

- **Invariante**: uma propriedade que permanece verdadeira durante a execução do algoritmo. Por exemplo, em um algoritmo de ordenação, um invariante pode ser que os elementos à esquerda de um índice específico estão sempre ordenados. (ex.: "não há custos reduzidos negativos" ou "cada componente tem ao menos um arco zero entrando").
- **Correção**: a garantia de que o algoritmo produz a saída correta para todas as entradas válidas. Isso geralmente é demonstrado por meio de provas formais ou argumentos lógicos. (ex.: justificativa de que o resultado final é uma arborescência válida e de custo mínimo)
- **Terminação**: a garantia de que o algoritmo sempre chegará a um ponto final, ou seja, que não entrará em um loop infinito. Isso pode ser demonstrado mostrando que alguma medida (como o tamanho da entrada) diminui a cada passo. (ex.: cada contração reduz $|V|$; cada ajuste cria um novo arco zero).

Essas características não definem formalmente o que é um algoritmo, mas ajudam a entender o conceito. A definição formal envolve a ideia de *computabilidade*[4], e isso envolve uma discussão profunda demais para o escopo deste trabalho.

2.7.1 Complexidade de Algoritmos

Porém, precisamos nos aprofunda em um dos conceitos que estão envolvidos em computabilidade: o de *complexidade de algoritmos*, que se refere à quantidade de recursos computacionais (tempo e espaço) que um algoritmo consome em função do tamanho da entrada.

A complexidade de um algoritmo pode ser analisada em termos de *complexidade de tempo* e *complexidade de espaço*. A complexidade de tempo refere-se ao tempo que um algoritmo leva para ser executado, enquanto a complexidade de espaço refere-se à quantidade de memória que um algoritmo utiliza durante sua execução.

A notação assintótica é frequentemente usada para expressar a complexidade de algoritmos, permitindo descrever o comportamento do algoritmo à medida que o tamanho da entrada cresce. As notações mais comuns são:

- **O grande (Big O)**: descreve um limite superior para o crescimento da função. Por exemplo, se um algoritmo tem complexidade $O(n^2)$, isso significa que o tempo de execução do algoritmo cresce no máximo proporcional a n^2 para entradas grandes.
- **Ômega (Ω)**: descreve um limite inferior para o crescimento da função. Se um algoritmo tem complexidade $\Omega(n)$, isso significa que o tempo de execução do algoritmo cresce no mínimo proporcional a n para entradas grandes.
- **Theta (Θ)**: descreve um limite assintótico preciso, indicando que a função cresce exatamente proporcional a uma determinada função. Se um algoritmo tem complexidade $\Theta(n \log n)$, isso significa que o tempo de execução do algoritmo cresce proporcional a $n \log n$ para entradas grandes.

Essas notações ajudam a comparar a eficiência de diferentes algoritmos e a entender como eles se comportam à medida que o tamanho da entrada aumenta[5]. Ao analisar a complexidade de um algoritmo, é importante considerar o pior caso, o caso médio e o melhor caso, dependendo do contexto em que o algoritmo será utilizado.

[4] A computabilidade é um conceito na teoria da computação, que se refere à capacidade de um problema ser resolvido por um algoritmo em um tempo finito. *Comentário formal.* Esse conceito é formalizado por meio de modelos como máquinas de Turing, funções recursivas, RAM, entre outros. Essas formalizações são equivalentes quanto ao que é computável (Tese de Church–Turing) e permitem discutir com rigor correção e complexidade (tempo e memória).

[5] Por "tamanho da entrada" entendemos a quantidade de símbolos necessária para codificar a instância (tipicamente, o número de bits). Exemplos: (i) para grafos, mede-se usualmente por $n = |V|$ e $m = |E|$; se há pesos, também se contabiliza o número de bits para representá-los; (ii) para inteiros, é o número de dígitos; (iii) para strings, o comprimento. Em análises de alto nível, é comum expressar custos como funções de n e m no modelo RAM (palavra de $\Theta(\log n)$ bits), mas quando os pesos são grandes a complexidade em bits pode prevalecer.

Para ilustrar a análise de complexidade, consideremos o exemplo da busca linear vs busca binária em um vetor ordenado[6].

Exemplo: Busca Linear vs Busca Binária

O algoritmo de busca linear percorre cada elemento do vetor até encontrar o valor desejado ou chegar ao final do vetor. A complexidade desse algoritmo é $O(n)$ no pior caso, onde n é o tamanho do vetor, pois pode ser necessário verificar todos os elementos.

Algoritmo 2.2: Busca Linear

Dado um vetor V de tamanho n e um valor x:

1. Para cada índice i de 0 a $n-1$:
   (a) Se $V[i] = x$, retorne i.
2. Retorne -1 (indica que x não está no vetor).

Já o algoritmo de busca binária aproveita o fato de que o vetor está ordenado para reduzir o espaço de busca pela metade a cada iteração.

Algoritmo 2.3: Busca Binária

Dado um vetor ordenado V de tamanho n e um valor x:

1. Defina início $= 0$ e fim $= n-1$.
2. Enquanto início $\leq$ fim:
   (a) Calcule $meio = \left\lfloor \dfrac{\text{início} + \text{fim}}{2} \right\rfloor$.
   (b) Se $V[meio] = x$, retorne $meio$.
   (c) Se $V[meio] < x$, defina início $= meio + 1$.
   (d) Caso contrário, defina fim $= meio - 1$.
3. Retorne -1 (indica que x não está no vetor).

O algoritmo de busca linear tem a seguinte complexidade:

- **Melhor caso**: $O(1)$ - o elemento procurado está na primeira posição.
- **Caso médio**: $O(n)$ - em média, metade dos elementos precisam ser verificados.
- **Pior caso**: $O(n)$ - o elemento procurado está na última posição ou não está no vetor.

[6]Por vetor ordenado entendemos um arranjo (array) em que os elementos estão armazenados em posições consecutivas e dispostos segundo uma ordem total (tipicamente crescente ou não decrescente). Essa organização permite algoritmos como a busca binária, que dependem de comparações para descartar metades do intervalo.

Já o algoritmo de busca binária tem a seguinte complexidade:

- **Melhor caso**: $O(1)$ - o elemento procurado está no meio do vetor.
- **Caso médio**: $O(\log n)$ - em média, a cada iteração, o tamanho do vetor é reduzido pela metade.
- **Pior caso**: $O(\log n)$ - o elemento procurado não está no vetor ou está na extremidade.

Esse exemplo ilustra como a análise de complexidade pode fornecer insights sobre a eficiência de um algoritmo em diferentes cenários. A busca binária é muito mais eficiente do que uma busca linear ($O(n)$) para grandes vetores, graças à sua capacidade de reduzir o espaço de busca pela metade a cada iteração.

2.7.2 Os problemas e suas complexidades

Não avaliamos só o desempenho de *um algoritmo*; também queremos saber *quão difícil é o próprio problema*, assim temos a seguinte forma de classificar problemas:

- **Problemas de decisão**: problemas que podem ser respondidos com "sim" ou "não". Ex.: "Existe um caminho entre dois vértices em um grafo?"
- **Problemas de otimização**: problemas que envolvem encontrar a melhor solução possível entre várias opções. Ex.: "Qual é o caminho mais curto entre dois vértices em um grafo ponderado?"
- **Problemas de contagem**: problemas que envolvem contar o número de soluções possíveis. Ex.: "Quantos caminhos existem entre dois vértices em um grafo?"

Cada classe de problemas pode ter diferentes níveis de dificuldade, que avaliamos em termos de *complexidade computacional*, que mede os recursos necessários (tempo e espaço) para resolver o problema.

Problemas são considerados "fáceis" quando são resolvíveis em tempo polinomial, enquanto outros são "difíceis" quando não se conhece nenhum algoritmo eficiente para resolvê-los.

Classes de complexidade de problemas

Como regra prática, consideramos *tratáveis* os problemas que admitem soluções em tempo (ou espaço) *polinomial* no tamanho da entrada e *intratáveis* os que não admitem. Essa distinção é formalizada por meio de *classes de complexidade*, que agrupam problemas segundo sua dificuldade intrínseca. Abaixo apresentamos-as:

- **P** (tempo polinomial). "Resolver é fácil": existe um algoritmo que encontra a resposta em tempo que cresce como n^k para algum k. Exemplos: conectividade em grafos, árvore geradora mínima (MST), caminho mínimo com pesos não negativos, fluxo máximo.
- **NP** (verificação polinomial). "Conferir é fácil": se alguém propõe uma solução, conseguimos *verificar* em tempo polinomial se ela está correta (achar pode ser difícil). Exemplos: *SAT* (satisfatibilidade booleana), *Clique*, *Vertex Cover*.

- **co-NP**. Complementos dos problemas em NP — "conferir o 'não' é fácil" em vez do "sim". Exemplo: *TAUT* (verificar se uma fórmula é tautologia) está em co-NP.
- **NP-difícil**. "Tão difíceis quanto o mais difícil de NP": todo problema de NP reduz-se (em tempo polinomial) a eles. Podem ser de decisão, otimização ou contagem e *não precisam* estar em NP. Em geral, não se espera algoritmo polinomial para todos os casos. Exemplos: versão de otimização do *TSP* (caixeiro-viajante), programação inteira, coloração mínima de grafos.
- **NP-completo**. "Os mais difíceis *dentro* de NP": problemas de decisão que estão em NP e são NP-difíceis. Se algum NP-completo tiver algoritmo polinomial, então P = NP. Exemplos: *SAT*, *3-SAT*, problema *Hamiltoniano* (existe ciclo hamiltoniano?).
- **PSPACE** (espaço polinomial). "Memória polinomial, tempo possivelmente enorme": resolvíveis usando memória que cresce polinomialmente com o tamanho da entrada. Exemplo: *QBF* (satisfatibilidade com quantificadores) é PSPACE-completo.

Reduções polinomiais

Para comparar dificuldades, usamos reduções de problemas, reduzimos o problema A ao problema B (escrevemos $A \leq_p B$) quando conseguimos transformar qualquer instância de A em uma instância de B em tempo polinomial, de modo que resolver B nos dê a resposta de A com apenas um sobrecusto polinomial. Logo: se $A \leq_p B$, então B **é pelo menos tão difícil quanto** A (um resolvedor para B resolveria A via redução). [7]

Relações conhecidas

Temos inclusões básicas: P $\subseteq$ NP, P $\subseteq$ co-NP e NP $\subseteq$ PSPACE $\subseteq$ EXP. Acredita-se que muitas dessas inclusões sejam estritas, mas isso não foi provado; em particular, o problema P vs NP permanece em aberto. Também não se sabe se NP = co-NP.

Essas classes não só categorizam problemas por sua dificuldade intrínseca, como também orientam a *estratégia de solução*. Em linhas gerais: (i) quando o problema está em P, preferimos **algoritmos exatos** com tempo polinomial; (ii) para problemas NP-completos/NP-difíceis, *não se conhecem* algoritmos exatos polinomiais (a menos que P = NP), e métodos gerais costumam ter pior caso exponencial. Por isso, são comuns **heurísticas**, **algoritmos de aproximação** e abordagens de **complexidade parametrizada** (FPT), além de algoritmos **pseudo-polinomiais** em casos numéricos. Muitas vezes, estruturas especiais (ex.: largura de árvore pequena, aciclicidade, graus limitados) também permitem soluções exatas polinomiais para subclasses. A seguir, explicitamos essa distinção entre *algoritmos exatos* e *heurísticas*.

[7] Consequência útil: se $A \leq_p B$ e B tem algoritmo polinomial, então A também tem. Para mostrar que um problema C é NP-difícil, reduzimos *de* um NP-completo conhecido P *para* C (isto é, $P \leq_p C$). Para provar que C é NP-completo, além disso precisamos que $C \in$ NP. Exemplo: 3-SAT $\leq_p$ Clique — resolver *Clique* eficientemente daria um método eficiente para *3-SAT*.

2.7.3 Tipificando Algoritmos

É comum ouvirmos que "o ótimo é inimigo do bom". Essa frase, atribuída a Voltaire, expressa a ideia de que buscar a perfeição pode impedir que se alcance um resultado satisfatório. Essa tensão entre buscar o ideal do "ótimo" ou aceitar o "suficientemente bom" quando recursos e tempo são limitados[8] essa noção se materializa em uma forma de tipificação de algoritmos.

Algoritmos Exatos vs Heurísticos

Essa distinção é especialmente relevante em problemas de otimização, onde o objetivo é encontrar a melhor solução possível entre um conjunto de soluções viáveis. Existem dois tipos principais de algoritmos para abordar esses problemas: os *algoritmos exatos* e os *algoritmos heurísticos*.

- **Algoritmos Exatos**: são aqueles que garantem encontrar a solução ótima para um problema, se uma solução existe. Eles exploram todas as possibilidades ou utilizam técnicas matemáticas rigorosas para garantir a optimalidade. Exemplos incluem algoritmos de programação linear, algoritmos de busca exaustiva e algoritmos baseados em teoria dos grafos, como o algoritmo de Dijkstra para caminhos mínimos.
- **Algoritmos Heurísticos**: são métodos que buscam soluções boas (mas não necessariamente ótimas) para problemas complexos, especialmente quando o espaço de soluções é muito grande ou quando o problema é NP-difícil. Eles utilizam regras práticas, aproximações ou estratégias de busca para encontrar soluções rapidamente. Exemplos incluem algoritmos genéticos, algoritmos de busca local e algoritmos de otimização por enxame de partículas.

Um tipo de algoritmo heurístico que merece destaque são os *algoritmos gulosos*, pois os algoritmos que estudaremos para encontrar r-arborescências de custo mínimo se enquadram nessa categoria.

Algoritmos Gulosos

Os algoritmos gulosos são uma classe de algoritmos heurísticos que tomam decisões locais ótimas em cada etapa, na esperança de que essas escolhas levem a uma solução globalmente ótima. Eles são frequentemente utilizados em problemas de otimização, onde uma solução ótima é desejada, mas encontrar essa solução pode ser computacionalmente inviável.

Os algoritmos gulosos são caracterizados por:

1. **Escolha local ótima**: Em cada etapa do algoritmo, uma escolha é feita com base em algum critério de otimização local. Essa escolha é feita sem considerar as consequências futuras, ou seja, o algoritmo "se contenta" com a melhor opção disponível no momento.

[8] Na teoria da decisão, essa postura pragmática é conhecida como *satisficing*, termo introduzido por Herbert A. Simon.

2. **Decisões definitivas**: Uma vez que uma escolha é feita, o algoritmo não reconsidera essa decisão. Isso significa que, se uma escolha levar a uma solução subótima, o algoritmo não tentará corrigir esse erro mais tarde.

3. **Eficiência**: Os algoritmos gulosos tendem a ser mais eficientes em termos de tempo de execução do que métodos exatos, pois não exploram todo o espaço de soluções. No entanto, essa eficiência pode vir à custa da qualidade da solução encontrada.

Exemplos clássicos de algoritmos gulosos incluem:

- **Kruskal**: seleciona as arestas de menor peso, evitando formar ciclos; produz uma árvore geradora mínima.
- **Prim**: inicia em um vértice e, a cada passo, adiciona a aresta mais leve que cruza o corte entre a árvore e o restante do grafo.
- **Dijkstra**: em digrafos (ou grafos) com pesos não negativos, expande pelo vértice de menor distância conhecida e relaxa suas saídas.
- **Kahn** (ordenação topológica): em DAGs, remove repetidamente vértices de grau de entrada zero e elimina suas saídas, construindo uma ordem topológica.
- **Chu–Liu/Edmonds** (arborescência mínima): escolhe, para cada $v \neq r$, o arco de menor custo que entra em v; ao formar ciclos, contrai-os e usa custos reduzidos até obter a r-arborescência de custo mínimo.
- **Frank** (arborescência mínima em duas fases): na primeira fase, constrói uma arborescência qualquer; na segunda, ajusta os custos reduzidos e troca arcos para minimizar o custo total.

Com esses conceitos em mente, estamos prontos para explorar os algoritmos específicos para encontrar r-arborescências de custo mínimo, que serão detalhados no próximo capítulo.

3 Em busca da Arborescência Perdida

Vamos usar esse capítulo para situar a evolução do problema: começamos revisitando como se encontra conectividade de menor custo em grafos não dirigidos por meio de *árvores geradoras mínimas* (MST), onde estratégias gulosas são corretas graças aos princípios de *corte* e de *ciclo*. Em seguida veremos por que, ao passar para *digrafos* e buscar uma **r-arborescência de custo mínimo**, essas mesmas receitas não se aplicam literalmente: surgem ciclos dirigidos e falta um "corte seguro" direto. Essa transição motiva as ferramentas certas — *custos reduzidos* e *contração de ciclos* — que aparecem no algoritmo de **Chu–Liu/Edmonds** e, adiante, no procedimento em duas fases de **Frank**.

3.1 Contexto Histórico

O problema de encontrar uma r-arborescência de custo mínimo em um digrafo ponderado é de certa forma uma evolução do problema de conectividade de menor custo em grafos não dirigidos, mas traz desafios adicionais que exigem novas ferramentas e estratégias.

A busca em grafos

Antes de tratarmos do caso *dirigido*, vamos falar sobre a intuição dominante de *como construir estruturas de conectividade de menor custo* vinha do caso de *grafos não dirigidos*: as **árvores geradoras mínimas**[9] (*Minimum Spanning Trees*, MST).

De modo geral, funciona a seguinte regra para esse caso: "escolha sempre a aresta mais barata disponível e encontraremos uma estrutura ótima". Existem dois princípios que justificam essa intuição:

- **Princípio do corte seguro.** Um *corte* é uma separação do conjunto de vértices em duas partes S e $V \setminus S$. Dizemos que uma aresta "cruza" o corte se tem uma ponta em cada lado. O princípio afirma: *a aresta de menor peso que cruza qualquer corte é segura*, ou seja, pode ser incluída em alguma MST sem perder optimalidade. É o mesmo que dizer intuitivamente, que se alguma solução ótima usa uma aresta e^* que cruza um certo corte e existe outra aresta e *mais barata* cruzando o mesmo corte, podemos trocar e^* por e. A troca mantém o grafo conectado (o corte continua sendo cruzado) e não aumenta o custo. Portanto, a mais barata do corte é sempre segura.

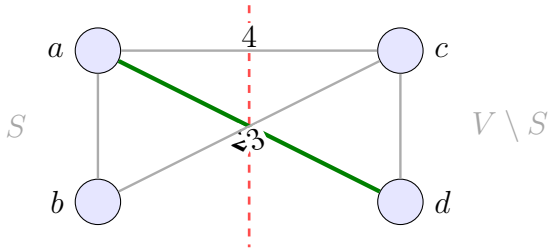


Figura 48: Princípio do corte seguro: entre as arestas que cruzam $(S, V \setminus S)$, a de menor peso (em verde) é *segura* — pode ser incluída em alguma MST sem perder optimalidade.

- **Princípio do uso da aresta mais pesada em um ciclo.** Em qualquer *ciclo*, a aresta de maior peso *não* pode pertencer a uma MST, pois existe uma troca que reduz (ou não aumenta) o custo total removendo essa aresta pesada, podemos entender que em um ciclo C, remover a aresta *mais pesada* não desconecta o grafo (há caminho alternativo dentro do próprio ciclo). Como é a mais cara, retirá-la só pode reduzir (ou manter) o custo. Logo, nenhuma MST precisa conter a aresta mais pesada de um ciclo.

[9]Definição. Em um grafo não dirigido, conexo e ponderado $G = (V, E)$ com pesos $w : E \to \mathbb{R}$, uma árvore geradora mínima é um subconjunto de arestas $T \subseteq E$ que forma uma árvore (conecta todos os vértices, é acíclica e tem $|T| = |V| - 1$) e que minimiza $\sum_{e \in T} w(e)$.

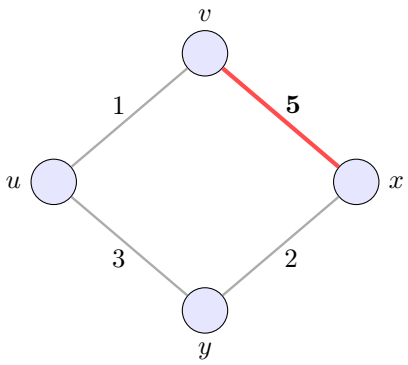


Figura 49: Princípio do ciclo: em qualquer ciclo, a aresta mais pesada (em vermelho) pode ser removida sem desconectar o grafo, reduzindo (ou não aumentando) o custo. Portanto, nenhuma MST contém a aresta mais pesada de um ciclo.

Assim o problema de encontrar uma árvore geradora mínima (MST) consiste em dado um grafo (não dirigido), conexo e ponderado $G = (V, E)$ com pesos $w : E \to \mathbb{R}$, queremos um subconjunto de arestas $T \subseteq E$ que conecta todos os vértices sem ciclos (forma uma árvore) e minimiza $\sum_{e \in T} w(e)$. Com base nos princípios acima, duas soluções gulosas são propostas: **Kruskal** e **Prim**.

- **Kruskal**: ordena as arestas por peso e adiciona enquanto não formar ciclo; usa o princípio do ciclo para evitar carregar a aresta mais pesada de um ciclo.
- **Prim**: começa em um vértice e cresce a árvore; a cada passo escolhe a aresta mais leve que cruza o corte entre "dentro" e "fora" — aplicação direta do princípio do corte.

Algoritmo 3.1: Kruskal (MST, guloso por peso crescente)

Entrada: grafo não dirigido $G = (V, E)$, pesos w.

1. Ordene as arestas por peso crescente.
2. Inicialize um *Union-Find*[a] com cada vértice em seu próprio conjunto; comece com $T = \emptyset$.
3. Para cada aresta $e = \{u, v\}$ na ordem: se u e v estão em componentes diferentes, una as componentes e adicione e a T.
4. Pare quando $|T| = |V| - 1$. Devolva T.

[a] Também conhecido como *Disjoint Set Union* (DSU) ou *estrutura de conjuntos disjuntos*. Mantém uma partição dinâmica dos vértices em componentes, oferecendo operações `find` (descobrir o representante do conjunto) e `union` (unir dois conjuntos). Com as heurísticas de *união por rank/tamanho* e *compressão de caminhos*, ambas operam em tempo amortizado $\alpha(n)$ (a função inversa de Ackermann), efetivamente constante na prática. Em Kruskal, essa estrutura detecta ciclos rapidamente ao verificar se as pontas de uma aresta pertencem a componentes distintas.

Kruskal é correto pelos princípios de corte e ciclo; com Union-Find eficiente, roda em $O(m \log m)$ (ou $O(m \log n)$).

Algoritmo 3.2: Prim (MST, expansão por corte mínimo)

Entrada: grafo não dirigido $G = (V, E)$, pesos w, vértice inicial s.

1. Inicialize $T = \{s\}$ e uma fila de prioridades[a] com as arestas que saem de T, chaveando pelo menor peso.
2. Enquanto $|T| < |V|$: extraia a aresta mais leve $\{u, v\}$ com $u \in T$ e $v \notin T$; adicione v e $\{u, v\}$ à árvore; atualize as chaves das arestas que cruzam o novo corte.
3. Devolva a árvore construída.

[a]Estrutura que mantém elementos com chaves de prioridade e permite extrair rapidamente o de menor (ou maior) chave. Implementações típicas: *heap* binário (push/decrease-key/pop em $O(\log n)$), *heap* de Fibonacci (decrease-key amortizado $O(1)$, pop em $O(\log n)$), e, em grafos com pesos pequenos, fila bucket (Dial) com tempos quase-lineares. No Prim, a fila é chaveada pelo menor peso de aresta que conecta o vértice fora da árvore ao conjunto T.

Prim também é correto pelo princípio do corte; com fila de prioridades binária, executa em $O(m \log n)$ (ou em $O(m + n \log n)$); em grafos densos com *heaps* de Fibonacci[10], pode-se obter $O(m + n \log n)$ [4, 13, 29, 5].

Os algoritmos gulosos de MST são corretos em grafos não dirigidos, mas a passagem para digrafos *não* é direta. No caso dirigido, buscamos uma **r-arborescência de custo mínimo**: para cada $v \neq r$, exatamente um arco entra em v, e o conjunto deve ser acíclico e alcançável a partir de r. Se imitarmos a receita de MST (escolher sempre o arco de entrada mais barato), aparecem *ciclos dirigidos*, e não há um análogo imediato do "corte seguro".

Duas abordagens clássicas contornam essas dificuldades: (i) **Chu–Liu/Edmonds**, que mantém *custos reduzidos* (criando arcos de custo reduzido zero) e resolve conflitos por *contração de ciclos*; e (ii) o procedimento em duas fases de **Frank**, que parte de uma arborescência qualquer e a refina via ajustes de custos e trocas de arcos. Em ambos os casos, escolhas locais são acopladas a um mecanismo global de consistência, garantindo otimalidade no caso dirigido [21].

3.1.1 A busca em digrafos

O primeiro avanço significativo na busca por arborescências de custo mínimo em digrafos foi feito por Y. Chu e T. Liu em 1965, que propuseram um algoritmo para encontrar

[10]*Heaps* (montes) são implementações clássicas de filas de prioridades. O **heap binário** mantém uma árvore quase completa e executa `insert`/`extract-min`/`decrease-key` em $O(\log n)$. Já o **heap de Fibonacci** é uma estrutura amortizada com `decrease-key` e `meld` (união) em $O(1)$ amortizado e `extract-min` em $O(\log n)$. Em algoritmos como Prim e Dijkstra, onde `decrease-key` é frequente, isso leva a $O(m + n \log n)$. Apesar da melhor garantia assintótica, constantes e implementação mais simples fazem heaps binários (ou *pairing heaps*) serem frequentemente competitivos na prática.

a arborescência de custo mínimo em um digrafo. Esse algoritmo foi posteriormente aprimorado por Jack Edmonds em 1967, que introduziu o conceito de custos reduzidos e a técnica de contração de ciclos, tornando o algoritmo mais eficiente e robusto.

Desde então, a pesquisa nessa área tem se concentrado em melhorar a eficiência dos algoritmos existentes, bem como em explorar novas técnicas e abordagens para lidar com diferentes tipos de digrafos e restrições adicionais. A contribuição de András Frank, que propôs um procedimento em duas fases para encontrar arborescências de custo mínimo, é um exemplo notável dessa evolução contínua.

3.2 Os meios para um fim

Maquiavel é conhecido por uma de suas citações: "Os fins justificam os meios". Essa frase ficou famosa porque tem uma interpretação polêmica: em certas circunstâncias, qualquer ação pode ser justificada se o resultado final for considerado positivo ou benéfico. Muitos não concordam com essa visão, argumentando que os meios também importam e que ações imorais não podem ser justificadas por bons resultados.

Por um lado, na matemática essa frase pode ser validada quando falamos em duas formas distintas de construir algoritmos: por *iteratividade* ou *recursividade*.

Iteratividade

Muitos algoritmos — incluindo os gulosos — são construídos por *iteração*: repetimos um bloco de instruções enquanto uma condição não é satisfeita. Para projetar e analisar laços com clareza, três ideias são centrais:

- **Invariante de laço**: uma propriedade que é verdadeira antes do laço e permanece verdadeira a cada iteração. Ela explica *o que* está sendo mantido correto durante a construção.
- **Variante (medida de progresso)**: uma quantidade que melhora estritamente a cada iteração (ex.: aumenta $|T|$, diminui $|V|$, reduz um potencial). Garante *terminação*.
- **Critério de parada e pós-condição**: quando o laço termina, o invariante implica a especificação desejada.

Na prática. Em **Kruskal**, o invariante é "T é uma floresta acíclica e cada componente foi conectado por arestas seguras"; a variante é $|T|$, que cresce até $|V| - 1$. Em **Prim**, "T conecta um conjunto de vértices e as chaves refletem o menor corte atual"; a variante é o tamanho de T. Muitas análises usam *custo por iteração* ou *análise amortizada*[11] para capturar o desempenho agregado.

Recursividade

[11] A análise amortizada distribui o custo de operações caras sobre uma sequência, garantindo um custo médio por operação (ex.: `decrease-key` em heaps de Fibonacci).

Recursão resolve instâncias grandes chamando o próprio algoritmo em subinstâncias menores. Um projeto claro inclui:

- **Casos base**: instâncias mínimas resolvidas diretamente.
- **Passo recursivo**: como decompor e combinar soluções dos subproblemas.
- **Medida decrescente**: uma grandeza que estritamente diminui a cada chamada (ex.: número de vértices após contração), assegurando terminção.
- **Corretude por indução**: assumimos corretas as chamadas recursivas (hipótese indutiva) e provamos que a combinação produz uma solução correta.
- **Custo por recorrência**: tempo expresso por $T(n)$ e resolvido por *árvore de recursão* ou Teorema Mestre[12].

Na prática. Em **Chu–Liu/Edmonds**, o passo recursivo contrai um ciclo dirigido e ajusta custos; a medida decrescente é $|V|$ (a cada contração reduzimos o número de vértices do problema), e a expansão final preserva otimalidade. No procedimento em duas fases de **Frank**, a primeira fase produz uma arborescência inicial; a segunda aplica *refinamentos iterativos* guiados por custos reduzidos — um exemplo de mistura entre recursão estrutural e iteração local.

Para ilustrar, resolvemos o mesmo problema — calcular o fatorial $n!$ — de forma *iterativa* e *recursiva*.

Algoritmo 3.3: Fatorial (iterativo)

Entrada: inteiro $n \geq 0$

1. Se $n = 0$, devolva 1.
2. Defina $r \leftarrow 1$.
3. Para i de 1 até n: $r \leftarrow r \cdot i$.
4. Devolva r.

Algoritmo 3.4: Fatorial (recursivo)

Entrada: inteiro $n \geq 0$

1. Se $n \leq 1$, devolva 1. (caso base)
2. Caso contrário, devolva $n \cdot \textsc{Fatorial}(n-1)$. (passo recursivo)

Ambas as versões computam a mesma função. A iterativa evidencia a *variante* (o contador i) e um *invariante* simples ($r = i!$ ao fim da iteração i); a recursiva explicita o *caso base* e o *passo indutivo*, e corresponde, operacionalmente, a empilhar chamadas com parâmetros decrescentes até $n = 1$.

[12]Esboço: quando $T(n) = aT(n/b) + f(n)$, com $a \geq 1$, $b > 1$, comparamos $f(n)$ a $n^{\log_b a}$. Não é necessário aqui, mas a técnica guia estimativas assintóticas.

Existe inclusive uma prova matemática[13] que diz que qualquer algoritmo iterativo pode ser reescrito de forma recursiva, e vice-versa. Ou seja, *os fins justificam os meios*: a escolha entre iteração e recursão é muitas vezes uma questão de preferência ou conveniência, já que ambos podem alcançar o mesmo objetivo. Porém, algoritmos recursivos podem ser mais elegantes e fáceis de entender, enquanto algoritmos iterativos podem ser mais eficientes em termos de uso de memória (evitando a sobrecarga da pilha de chamadas). Ou seja, os meios também importam. E os trabalhos de Chu–Liu/Edmonds e Frank ilustram bem essa tensão:

- **Chu–Liu/Edmonds** é um algoritmo recursivo que contrai ciclos e resolve o problema em subinstâncias menores, usando custos reduzidos para manter a otimalidade.
- O procedimento em **Frank** é iterativo, começando com uma arborescência qualquer e refinando-a por trocas locais guiadas por custos reduzidos, até alcançar a otimalidade.

Ambos os algoritmos são corretos e eficientes, mas adotam meios diferentes para alcançar o mesmo fim: encontrar uma r-arborescência de custo mínimo em um digrafo ponderado.

Antes de entrar nos detalhes, faremos uma breve revisão dos conceitos e técnicas que utilizaremos adiante, para uniformizar a notação e tornar a leitura mais fluida.

Revisão: conceitos fundamentais e técnicas

Antes de mergulharmos nos algoritmos específicos, é importante revisitar alguns conceitos fundamentais que serão cruciais para nossa compreensão:

Conceitos Fundamentais:

- **digrafo**: um grafo direcionado onde os arcos têm uma direção específica, indo de um vértice a outro.
- **Arborescência**: uma árvore direcionada onde todos os caminhos partem de um vértice raiz e alcançam todos os outros vértices.
- **Custo dos Arcos**: cada arco em um digrafo pode ter um custo associado, representando, por exemplo, o custo de transporte ou a distância.
- **r-Arborescência de Custo Mínimo**: uma arborescência enraizada em um vértice r que minimiza a soma dos custos dos arcos que a compõem.
- **Cortes e Conectividade**: a importância dos cortes em digrafos para garantir a conectividade e a existência de arborescências.
- **Condicionalidade de Fulkerson**: as condições necessárias e suficientes para a existência de uma r-arborescência de custo mínimo.

[13] Em modelos padrão de computação (máquinas de Turing, RAM), *iteração* e *recursão* têm o mesmo poder expressivo: laços podem ser reescritos como recursão (sobre um contador/estado) e chamadas recursivas podem ser eliminadas por uma simulação explícita da pilha (iteração com uma estrutura *stack*). Demonstrações e variantes aparecem em textos clássicos de teoria da computação e projeto de algoritmos; ver, por exemplo, [4] (eliminação de recursão via pilha explícita). Aqui registramos apenas a equivalência conceitual, sem apresentá-la.

Técnicas e Abordagens:

Para encontrar r-arborescências de custo mínimo, utilizaremos algumas técnicas e abordagens específicas:

- **Normalização de Custos**: ajustaremos os custos dos arcos para facilitar a identificação de arcos de custo reduzido zero.
- **Contração de Ciclos**: quando formos encontrar ciclos de arcos de custo reduzido zero, os contrairemos para simplificar o digrafo.
- **Custos Reduzidos**: utilizaremos custos reduzidos para identificar arcos que podem ser incluídos na arborescência sem aumentar o custo total.
- **Estratégias Gulosas**: aplicaremos estratégias gulosas para selecionar arcos de forma eficiente, garantindo que cada escolha local contribua para a solução global ótima.
- **Recursão e Expansão**: usaremos recursão para resolver subproblemas em digrafos contraídos e expandiremos as soluções para o digrafo original.
- **Análise de Complexidade**: avaliaremos a eficiência dos algoritmos em termos de tempo e espaço, garantindo que sejam viáveis para grandes digrafos.
- **Provas de Corretude**: forneceremos argumentos formais para garantir que os algoritmos realmente produzem a r-arborescência de custo mínimo.

No capítulo seguinte, detalharemos o algoritmo de Chu–Liu/Edmonds, bem como os detalhes da implementação em Python; o subsequente será dedicado ao procedimento em duas fases de Frank e à respectiva implementação.

4 Algoritmo de Chu–Liu/Edmonds

O algoritmo de Chu–Liu/Edmonds é um método clássico para encontrar uma r-arborescência de custo mínimo em um digrafo ponderado. Ele combina estratégias gulosas com técnicas de normalização de custos e contração de ciclos para garantir a otimalidade da solução.

Se tentarmos copiar a receita das MSTs — dar a cada vértice $v \neq r$ o arco de entrada mais barato — corremos o risco de fechar um *ciclo dirigido* que não chega à raiz r.

Por que isso é proibido? Em uma r-arborescência cada $v \neq r$ deve ter exatamente um arco de entrada e r tem grau de entrada zero. Se houvesse um ciclo dirigido C, todos os vértices de C já receberiam seu único arco de entrada de dentro do próprio C, logo nenhum arco entraria em C a partir de $V \setminus C$ (o corte $\delta^-(C)$ ficaria vazio). Como $r \notin C$, não existe caminho de r para os vértices de C, contrariando a alcançabilidade exigida. Portanto, ciclos dirigidos são incompatíveis com a estrutura de r-arborescência.

Dessa forma, o desafio é duplo: preservar a informação local de "mais barato por vértice" (que é valiosa) e, ao mesmo tempo, impedir que essas escolhas locais se combinem em ciclos.

Uma maneira direta de enxergar isso é com um microexemplo: tome três vértices a, b, c (todos fora de r). Se o arco mais barato que entra em b vem de a, o de c vem de b e o de a vem de c, então as escolhas "mais baratas" formam o ciclo $a \to b \to c \to a$. Note que, entre essas escolhas locais, nenhum deles recebe o arco vindo de r (mesmo que existam $r \to a$, $r \to b$, $r \to c$ mais caros); ficamos presos dentro do ciclo e não alcançamos a raiz. É exatamente esse impasse que o algoritmo resolve ao zerar custos por vértice e contrair ciclos, deixando para decidir qual arco interno remover apenas na expansão.

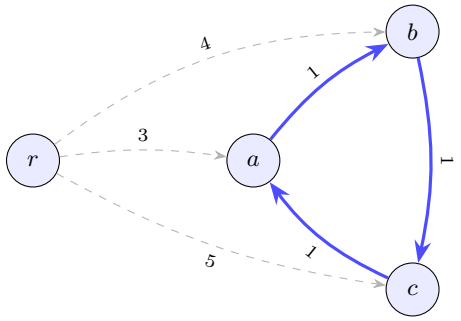


Figura 50: Ciclo gerado pelas escolhas locais "mais baratas por vértice". Os arcos grossos (custo 1) entram em a, b, c e formam $a \to b \to c \to a$. Os arcos tracejados partindo de r existem, mas são mais caros e por isso não são escolhidos pelo critério local.

Partindo desse cenário, a ideia é *normalizar os custos por vértice*: para cada $v \neq r$, "descontamos" de todo arco que entra em v o menor custo que chega a v. Após esse ajuste (custos reduzidos), cada $v \neq r$ passa a ter ao menos um arco de custo reduzido zero entrando. Se os arcos de custo zero forem acíclicos, já temos a r-arborescência. Se formarem um ciclo C, isso indica que, dentro de C, todos os vértices atingiram seus mínimos locais; então *contraímos* C em um **supervértice** x_C e repetimos o processo no grafo menor. Ao final, *expandimos* as contrações e, em cada ciclo expandido, removemos exatamente um arco para manter grau de entrada 1 e a aciclicidade global.

Supervértices e contração de ciclos

Dado um subconjunto $C \subseteq V$ que forma um ciclo dirigido, a *contração de* C substitui todos os vértices de C por um único vértice x_C - **o supervértice**. Todo arco com exatamente uma ponta em C passa a ser incidente a x_C: arcos (u, w) com $u \notin C$, $w \in C$ tornam-se (u, x_C); arcos (w, v) com $w \in C$, $v \notin C$ tornam-se (x_C, v); e arcos com as duas pontas em C tornam-se laços em x_C e são descartados. Quando trabalhamos com *custos reduzidos*, ajustamos em particular os arcos que *entram* em C para preservar a comparação relativa: para um arco (u, w) com $w \in C$, definimos $c'(u, x_C) = c(u, w) - c(a_w)$, onde a_w é o arco mais barato que entra em w. Essa normalização garante que, ao resolver no grafo contraído, decisões ótimas podem ser traduzidas de volta na etapa de expansão.

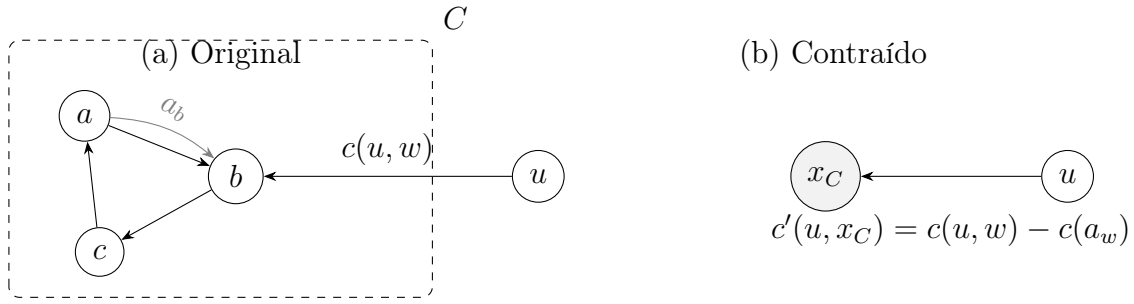


Figura 51: Ajuste de custo reduzido para um arco *entrando*: ao contrair um ciclo C, o arco (u, w) com $w \in C$ passa a (u, x_C) com custo reduzido $c'(u, x_C) = c(u, w) - c(a_w)$, onde a_w é o arco de menor custo que entra em w.

Para cada arco (u, w) com $w \in C$, ajustamos o custo reduzido para $c'(u, x_C) = c(u, w) - c(a_w)$, onde a_w é o arco mais barato que entra em w. No exemplo da Figura 51, o arco (u, b) com custo 7 torna-se (u, x_C) com custo reduzido $7 - 5 = 2$, já que $a_b = (a \to b)$ tem custo 5. Arcos que saem de C ou laços internos não são ajustados.

4.1 Descrição do algoritmo

Em computação falamos em linguagem de alto nível quando uma linguagem de programação é próxima da linguagem humana, com abstrações que facilitam o entendimento. Em contraste, linguagens de baixo nível são mais próximas do código de máquina, exigindo detalhes explícitos.

Podemos falar também em visão operacional de alto nível quando descrevemos um algoritmo focando na lógica e nos passos principais, sem entrar em detalhes de implementação específicos.

Aqui, apresentamos o algoritmo de Chu–Liu/Edmonds em uma visão operacional de alto nível, focando na lógica e nos passos principais, sem entrar em detalhes de implementação específicos. E dedicaremos a próxima subseção apenas para discutir detalhes práticos de implementação.

Denotamos por A' o conjunto de arcos escolhidos na construção da r-arborescência.

Construa A' escolhendo, para cada $v \neq r$, um arco de menor custo que *entra* em v. Se (V, A') é acíclico, então, pela caracterização de arborescências (grau de entrada 1 para todo $v \neq r$ e ausência de ciclos), A' já é uma r-arborescência; e é *ótima*, pois realizamos o menor custo de entrada em cada vértice e nenhuma troca pode reduzir o custo mantendo as restrições [13, Sec. 4.9].

Se A' contiver um ciclo dirigido C (que não inclui r), normalizamos os custos de entrada (custos reduzidos) e *contraímos* C em um supervértice x_C, ajustando apenas arcos que *entram* em C por $c'(u, x_C) = c(u, w) - c(a_w)$.

Resolvemos recursivamente no grafo contraído. As arborescências do grafo contraído correspondem, em bijeção, às arborescências do grafo original com exatamente um arco entrando em C; como os arcos de C têm custo reduzido zero, os custos são preservados na ida e na volta.

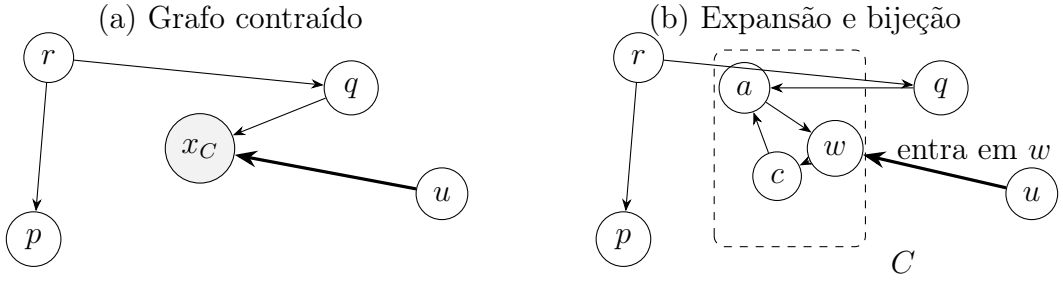


Figura 52: Bijeção entre arborescências no grafo contraído e no original: (a) toda arborescência em D' escolhe exatamente um arco que entra em x_C; (b) ao expandir C, esse arco corresponde a um (u, w) que entra em algum $w \in C$ e os arcos internos (de custo reduzido zero) são mantidos. O custo total é preservado.

A Figura 52 ilustra a bijeção entre arborescências no grafo contraído e no grafo original, destacando a preservação de custos e a manutenção das propriedades estruturais necessárias.

Na expansão, reintroduzimos C e removemos exatamente um arco interno para manter grau de entrada 1 e aciclicidade global [21, 13].

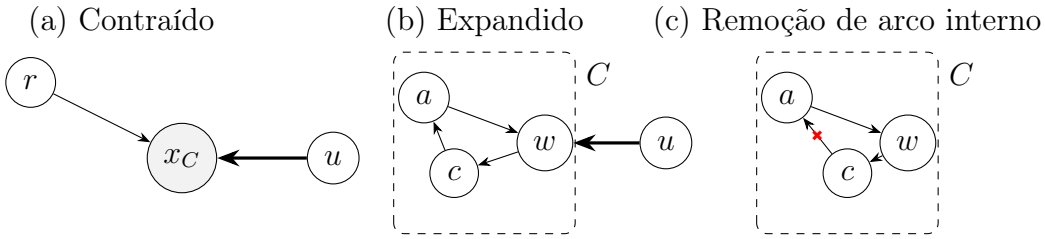


Figura 53: Reexpansão de C: (a) no grafo contraído seleciona-se um único arco que entra em x_C; (b) ao expandir, x_C é substituído por C e o arco selecionado passa a entrar em algum $w \in C$; (c) remove-se exatamente um arco interno de C para eliminar o ciclo, preservando conectividade a partir de r e o custo total (os arcos internos têm custo reduzido zero).

A Figura 53 detalha o processo de expansão e remoção de um arco interno para quebrar o ciclo, garantindo que a estrutura de r-arborescência seja mantida.

Abaixo, temos a descrição formal do algoritmo.

Algoritmo 4.1: Chu–Liu/Edmonds (visão operacional)

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

1. Para cada $v \neq r$, escolha $a_v \in \operatorname{argmin}_{(u,v)\in A} c(u, v)$. Defina $y(v) := c(a_v)$ e $F^* := \{a_v : v \neq r\}$.
2. Se (V, F^*) é acíclico, devolva F^*. Por [13, Obs. 4.36], trata-se de uma r-arborescência de custo mínimo.
3. Caso contrário, seja C um ciclo dirigido de F^* (com $r \notin C$). **Contração:** contraia C em um supervértice x_C e defina custos c' por

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

   descartando laços em x_C e permitindo paralelos. Denote o digrafo contraído por $D' = (V', A')$.
4. **Recursão:** compute uma r-arborescência ótima T' de D' com custos c'.
5. **Expansão:** seja $(u, x_C) \in T'$ o único arco que entra em x_C. No grafo original, ele corresponde a (u, w) com $w \in C$. Forme

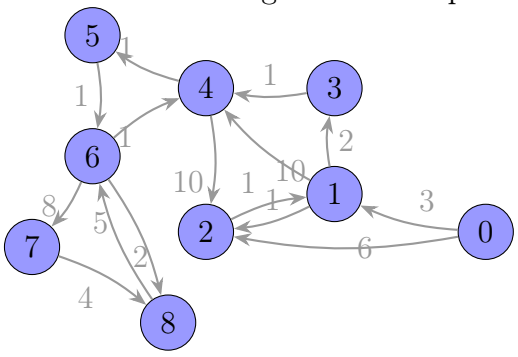
$$T := \big(T' \setminus \{\text{arcos incidentes a } x_C\}\big) \ \cup\ \{(u, w)\} \ \cup\ \big((F^* \cap A(C)) \setminus \{a_w\}\big).$$

   Então T tem grau de entrada 1 em cada $v \neq r$, é acíclico e tem o mesmo custo de T'; logo, é uma r-arborescência ótima de D [13, 21, Sec. 4.9].

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

4.1.1 Exemplo prático: Chu–Liu/Edmonds

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

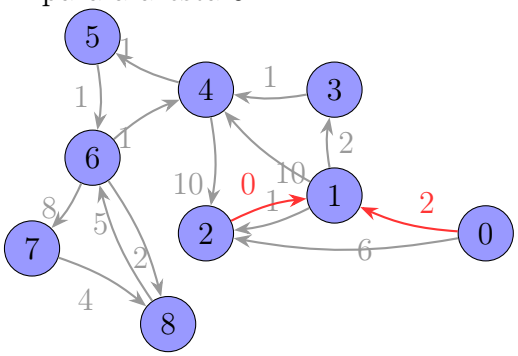


O primeiro passo do nosso algoritmo seria remover as arestas que entram na raiz (vértice 0), porém não há nenhuma nesse caso, logo não existe a necessidade de alterar o grafo.

Dessa forma, o próximo passo é normalizar os pesos das arestas de entrada para cada vértice, nessa etapa, Para cada vértice X (exceto a raiz), o algoritmo encontra a aresta de

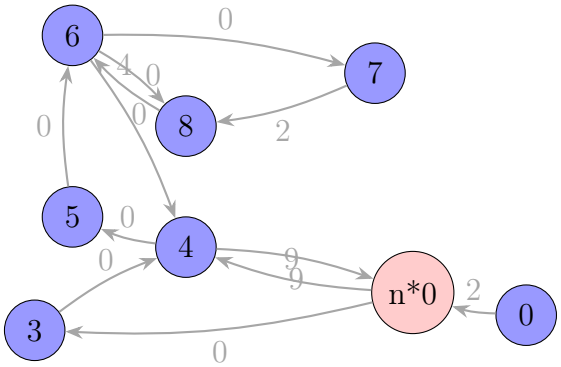
menor peso que entra em X e subtrai esse menor peso de todas as arestas que entram em X (relembrando que isso serve para zerar o peso da aresta mínima de entrada em cada vértice)

Normalizando pesos de arestas de entrada para '1': Nesse processo notamos que as únicas arestas de entrada são 0 e 2 onde $(0 \rightarrow 1)$ tem peso 3.0 e $(2 \rightarrow 1)$ tem peso 1.0, elegendo a aresta 2 como a de menor peso podemos subtrair o peso das arestas restantes (no caso, o peso da aresta 0) pelo valor do peso da aresta 2, resultando em um novo peso de '2' para a aresta 0

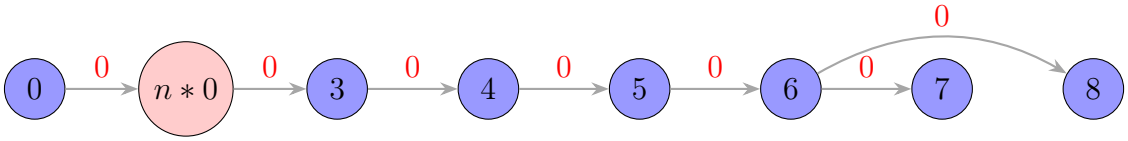


Repetiremos o passo anterior para todas as outras arestas

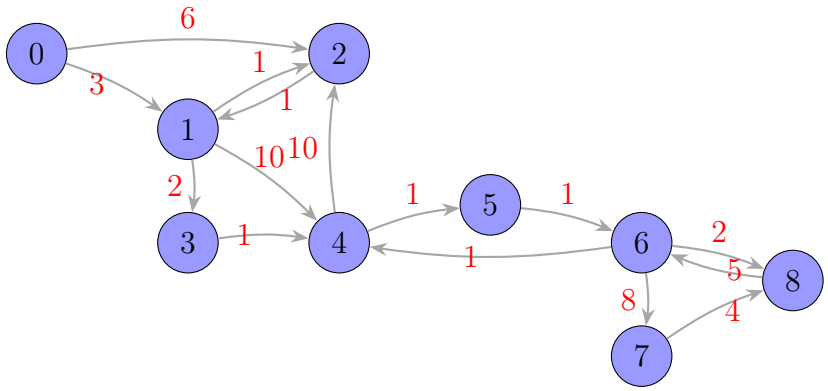
Com os pesos normalizados, o próximo passo é construir F^*, para isso, selecionamos para cada vértice, a aresta de menor custo de entrada. Além disso, detectamos um ciclo em F^*, formado pelos vértices {1 e 2}. Portanto, precisamos contrair esse ciclo em um supervértice $n * 0$. O resultado é o seguinte:



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



Após validarmos que a F* não possuí mais ciclos e notarmos que F* forma uma arborescência iremos começar o processo de expanção do ciclo contraído para obter a arborescência final no grafo original. Dessa forma, Adicionamos a aresta de entrada ao ciclo: (0, 1), (1, 2) e a aresta externa de saída: (1, 3), chegando em uma arborescência válida.



4.1.2 Corretude

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

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

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

   pois há exatamente um arco de T entrando em cada $v \neq r$. O termo $\sum_{v \neq r} y(v)$ é constante (independe de T); assim, minimizar $\sum c$ equivale a minimizar $\sum c'$ [13, Obs. 4.37]. Em particular, os arcos a_v de menor custo que entram em v têm custo reduzido zero e formam F^*.

2. *Caso acíclico:* se (V, F^*) é acíclico, então já é uma r-arborescência e, por realizar o mínimo custo de entrada em cada $v \neq r$, é ótima [13, Obs. 4.36].

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

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

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

   Como os arcos de C têm custo reduzido zero e $c'(u, x_C) = c(u,w) - y(w)$, a soma dos custos reduzidos é preservada na ida e na volta; logo, T' ótimo em D' mapeia para T ótimo em D para c'. Pela equivalência entre c e c', T também é ótimo para c. Repetindo o argumento a cada contração, obtemos a corretude por indução [13, 21, Sec. 4.9].

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

4.1.3 Complexidade

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

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

4.2 Implementação em Python

Esta seção apresenta uma implementação em Python do algoritmo de Chu–Liu/Edmonds. A arquitetura segue os passos teóricos: recebe como entrada um digrafo ponderado, os custos das arestas e o vértice raiz. O procedimento seleciona, para cada vértice, o arco de menor custo de entrada, verifica se o grafo é acíclico e, se necessário, contrai ciclos e ajusta custos. Ao final, retorna como saída a r-arborescência ótima: um conjunto de arestas que conecta todos os vértices à raiz com custo mínimo.

Especificação da interface (entradas, saídas e hipóteses)

- **Entrada:** digrafo ponderado $D = (V, A)$, custos $c : A \to \mathbb{R}$, raiz $r \in V$.
- **Hipóteses:**
  - D é representado como um objeto `networkx.DiGraph`, com pesos armazenados no atributo de arestas `'w'`.
  - D é conexo a partir de r:
  - (i) todo $v \neq r$ é alcançável a partir de r (caso contrário, não há r-arborescência); (ii) para todo subconjunto não vazio $X \subseteq V \setminus \{r\}$, existe ao menos um arco que entra em X ($\delta^-(X) \neq \emptyset$; condições clássicas de existência à la Edmonds [21]).
  - Os custos são não negativos: $c(a) \geq 0$ para todo $a \in A$.
- **Saída:** conjunto $A^* \subseteq A$ com $|A^*| = |V| - 1$, tal que cada $v \neq r$ tem grau de entrada 1, todos os vértices são alcançáveis a partir de r e $\sum_{a \in A^*} c(a)$ é mínimo.
- **Convenções:** arcos paralelos (múltiplos arcos entre o mesmo par de vértices) são permitidos após contrações; laços (self-loops) são descartados.

4.2.1 Funções principais

O código a seguir implementa o algoritmo de Chu–Liu/Edmonds em Python, utilizando a biblioteca NetworkX para manipulação de grafos. A implementação segue a estrutura descrita na seção anterior, com funções auxiliares para cada etapa do algoritmo.

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

Normalização por vértice: para um vértice alvo v, a função normaliza[14] os custos das arestas que *entram* em v: calcula $y(v) = \min\{w(u, v)\}$ e substitui cada peso $w(u, v)$ por $w(u, v) - y(v)$.

Com isso, ao menos uma entrada em v passa a ter custo reduzido zero e a ordem relativa entre as entradas é preservada. A função modifica o grafo *in-place*[15], ou seja, sem criar uma cópia.

A função executa em $O(\deg^-(v))$. Pois, dado o vértice v, obtemos suas arestas de entrada em $O(\deg^-(v))$; calcular o mínimo e ajustar os pesos também leva $O(\deg^-(v))$.

Normalização por vértice: custos reduzidos

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

```
% def normalize_incoming_edge_weights(D: nx.DiGraph, node: str
    ):

%     # Get the incoming edges of the node with their weights
%     predecessors = list(D.in_edges(node, data="w"))

%     if not predecessors:
%         return

%     # Calculate the minimum weight among the incoming edges
%     yv = min((w for _, _, w in predecessors))

%     # Subtract Yv from each incoming edge
%     for u, _, _ in predecessors:
%         D[u][node]["w"] -= yv
```

Construção de F^*: a função constrói o subdigrafo F^* selecionando, para cada vértice $v \neq r_0$, uma única aresta de menor custo que entra em v (isto é, um $\text{argmin}_{(u,v)\in A} w(u, v)$).

Se os custos já foram normalizados por vértice, o arco escolhido tem custo reduzido zero; caso contrário, ele devolve None. O resultado é um digrafo com exatamente uma aresta entrando em cada $v \neq r_0$ e nenhuma aresta entrando em r_0. A função executa em $O(m)$, onde m é o número de arestas.

Isso ocorre porque a função itera sobre todos os vértices do grafo e, para cada vértice, verifica suas arestas de entrada para encontrar a de menor peso. Como cada aresta é

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

[15] No jargão de programação, significa "no próprio lugar": a estrutura de dados original é alterada diretamente, sem criar uma cópia. Isso economiza memória e tempo, mas introduz efeitos colaterais — outras referências ao mesmo objeto verão as mudanças.

considerada no máximo uma vez durante essa iteração, o tempo total é proporcional ao número de arestas, ou seja, $O(m)$. A função não modifica o grafo original, mas cria um novo grafo direcionado F^*.

Construção de F star

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

```
1 % def get_Fstar(D: nx.DiGraph, r0: str):
2 %
3 %     # Create an empty directed graph for F_star
4 %     F_star = nx.DiGraph()
5 %
6 %     for v in D.nodes():
7 %         if v != r0:
8 %             in_edges = list(D.in_edges(v, data="w"))
9 %             if not in_edges:
10 %                 continue  # No edges entering v
11 %             u = next((u for u, _, w in in_edges if w \eq\eq
     0), None)
12 %             if u:
13 %                 F_star.add_edge(u, v, w\eq0)
14 %     return F_star
```

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

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

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

Detecção de ciclo dirigido em F^*

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

```
1 % def find_cycle(F_star: nx.DiGraph):
2 %     try:
3 %         nodes_in_cycle = set()
4 %         # Extract nodes involved in the cycle
5 %         for u, v, _ in nx.find_cycle(F_star, orientation="
     original"):
```

```
6 %                 nodes_in_cycle.update([u, v])
7 %             # Create a subgraph containing only the cycle
8 %             return F_star.subgraph(nodes_in_cycle).copy()
9
10 %        except nx.NetworkXNoCycle:
11 %            return None
```

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

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

Essas escolhas garantem a *equivalência de custo* entre soluções ótimas no grafo contraído e no original após a reexpansão: os arcos internos de C têm custo reduzido zero e apenas as entradas em C recebem o desconto $y(w)$, mantendo a bijeção entre arborescências descrita anteriormente.
A expressão "no próprio lugar (inplace)" no docstring abaixo[16] indica que o grafo D é modificado diretamente.

Contração de ciclo

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

```
1 % def contract_cycle(D: nx.DiGraph, C: nx.DiGraph, label: str)
    :
2
3
4 %     cycle_nodes: set[str] = set(C.nodes())
5
6 %     # Stores the vertex u outside the cycle and the vertex v
      inside the cycle that receives the minimum weight edge
7 %     in_to_cycle: dict[str, tuple[str, float]] = {}
8
```

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

```
9 %     for u in D.nodes:
10 %         if u not in cycle_nodes:
11 %             # Find the minimum weight edge that u has to any
      vertex in C
12 %             min_weight_edge_to_cycle = min(
13 %                 ((v, w) for _, v, w in D.out_edges(u, data="
     w") if v in cycle_nodes),
14 %                 key=lambda x: x[1],
15 %                 default=None,
16 %             )
17 %             if min_weight_edge_to_cycle:
18 %                 in_to_cycle[u] = min_weight_edge_to_cycle
19 %
20 %     for u, (v, w) in in_to_cycle.items():
21 %         D.add_edge(u, label, w=w)
22 %
23 %     # Stores the vertex v outside the cycle that receives
     the minimum weight edge from a vertex u inside the cycle
24 %     out_from_cycle: dict[str, tuple[str, float]] = {}
25 %
26 %     for v in D.nodes:
27 %         if v not in cycle_nodes:
28 %             # Find the minimum weight edge that v receives
     from any vertex in C
29 %             min_weight_edge_from_cycle = min(
30 %                 ((u, w) for u, _, w in D.in_edges(v, data="w
     ") if u in cycle_nodes),
31 %                 key=lambda x: x[1],
32 %                 default=None,
33 %             )
34 %             if min_weight_edge_from_cycle:
35 %                 out_from_cycle[v] =
     min_weight_edge_from_cycle
36 %
37 %     for v, (u, w) in out_from_cycle.items():
38 %         D.add_edge(label, v, w=w)
39 %
40 %     # Remove all nodes in the cycle from G
41 %     D.remove_nodes_from(cycle_nodes)
42 %
43 %     return in_to_cycle, out_from_cycle
44 %
```

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

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

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

Remoção de arestas que entram na raiz

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

```
% def remove_edges_to_r0(
%     D: nx.DiGraph, r0: str
% ):
%     # Remove all edges entering r0
%     in_edges = list(D.in_edges(r0))
%     if in_edges:
%         D.remove_edges_from(in_edges)
%     return D
```

Remoção de arco interno: ao expandir o ciclo C, a função remove o arco interno que entra no vértice de entrada v do ciclo, já que v agora recebe um arco externo do grafo. A função modifica o subgrafo do ciclo *in-place* e executa em $O(\deg^-(v))$.

Remover arco interno na reexpansão

Remove a aresta interna que entra no vértice de entrada 'v' do ciclo C, pois 'v' passa a receber uma aresta externa do grafo.

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

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

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

Procedimento principal (recursivo): A função principal implementa o algoritmo de Chu–Liu/Edmonds de forma recursiva e atua como um orquestrador das fases do método. Em alto nível, ela mantém a seguinte lógica:

O procedimento principal do algoritmo segue estes passos: prepara a instância removendo entradas na raiz, normaliza os custos das arestas que entram em cada vértice (exceto a raiz) para garantir pelo menos uma entrada de custo reduzido zero, constrói o grafo funcional F^* escolhendo para cada vértice a entrada de menor custo reduzido, verifica se F^* é acíclico (se for, retorna como r-arborescência ótima), e, caso haja ciclo, contrai o ciclo em um supervértice, ajusta os custos das entradas e resolve recursivamente; ao retornar, expande o ciclo e remove uma aresta interna para garantir aciclicidade e grau de entrada igual a 1.

Mais especificamente, o procedimento garante as seguintes propriedades e passos:

- **Função (entradas/saídas):** Entrada: digrafo ponderado $D = (V, A)$, raiz r_0, e, opcionalmente, funções `draw_fn` e `log` para visualização e registro. Saída: um subdigrafo dirigido T de D com $|V| - 1$ arcos em que todo $v \neq r_0$ tem grau de entrada 1, todos os vértices alcançam r_0 e o custo total $\sum_{a \in T} c(a)$ é mínimo.
- **Invariantes:** Após a normalização por vértice, cada $v \neq r_0$ tem pelo menos uma entrada de custo reduzido zero; o conjunto F^* contém exatamente uma entrada por vértice distinto de r_0; em toda contração, apenas arcos que *entram* no componente têm seus custos reduzidos ajustados por $c'(u, x_C) = c(u, w) - c(a_w)$, preservando comparações relativas.
- **Detecção de ciclo e contração:** Se F^* contém um ciclo C, todos os seus arcos têm custo reduzido zero. O procedimento forma o supervértice x_C, reescreve arcos incidentes (descarta laços internos) e prossegue na instância menor. Essa etapa pode manter arcos paralelos e ignora laços.
- **Recursão e expansão:** Ao obter T' ótimo no grafo contraído, o método mapeia T' de volta para D: substitui o arco (u, x_C) por um (u, w) apropriado (com $w \in C$) e remove uma única aresta interna de C, restaurando a propriedade "uma entrada por vértice" e a aciclicidade.
- **Empates e robustez:** Empates de custo são resolvidos de modo determinístico/local, sem afetar a otimalidade. Arcos paralelos podem surgir após contrações e são tratados normalmente; laços são descartados por construção.
- **Logs e desenho (opcionais):** Na implementação disponibilizada no repositório do projeto integramos o solver com a interface do projeto de forma que se fornecidos, `log` recebe mensagens estruturadas por nível de recursão, e `draw_fn` e `draw_step` pode ser chamado para ilustrar passos relevantes (normalização, detecção/contração de ciclos, retorno da recursão e expansão).
- **Casos-limite:** Se algum $v \neq r_0$ não possui arco de entrada na instância corrente, detecta-se inviabilidade (não existe r-arborescência). Se F^* já é acíclico, retorna imediatamente (base da recursão).
- **Complexidade:** Em uma implementação direta, cada nível de recursão executa seleção/checagem/ajustes em tempo proporcional a $O(m)$, e há no máximo $O(n)$ níveis devido às contrações, totalizando $O(mn)$ e memória $O(m + n)$.

Essa rotina encapsula, portanto, a estratégia primal do método: induzir arestas de custo reduzido zero por normalização local, extrair uma estrutura funcional F^* de uma entrada por vértice, e resolver conflitos cíclicos por contração/expansão, preservando custos e correção em todas as etapas.

Procedimento principal (recursivo)

Função recursiva que encontra a arborescência ótima em um digrafo D com raiz r0 usando o algoritmo de Chu–Liu/Edmonds.

```
1 % def find_optimum_arborescence_chuliu(
2 %     D: nx.DiGraph,
```

```
3 %     r0: str,
4 %     level=0,
5 % ):
6 %
7 %     D_copy = D.copy()
8 %
9 %     for v in D_copy.nodes:
10 %         if v != r0:
11 %             normalize_incoming_edge_weights(D_copy, v)
12 %
13 %     # Build F_star
14 %     F_star = get_Fstar(D_copy, r0)
15 %
16 %     if nx.is_arborescence(F_star):
17 %         for u, v in F_star.edges:
18 %             F_star[u][v]["w"] = D[u][v]["w"]
19 %         return F_star
20 %
21 %     else:
22 %         C: nx.DiGraph = find_cycle(F_star)
23 %
24 %         contracted_label = f"\n n*{level}"
25 %         in_to_cycle, out_from_cycle = contract_cycle(
26 %             D_copy, C, contracted_label
27 %         )
28 %
29 %         # Recursive call
30 %         F_prime = find_optimum_arborescence_chuliu(
31 %             D_copy,
32 %             r0,
33 %             level + 1
34 %         )
35 %
36 %         # Identify the vertex in the cycle that received the
    only incoming edge from the arborescence
37 %         in_edge = next(iter(F_prime.in_edges(
    contracted_label, data="w")), None)
38 %
39 %         u, _, _ = in_edge
40 %
41 %         v, _ = in_to_cycle[u]
42 %
43 %         # Remove the internal edge entering vertex `v` from
    cycle C
44 %         remove_internal_edge_to_cycle_entry(
45 %             C, v
46 %         )  # Note: w is coming from F_prime, not from G
47 %
48 %         # Add the external edge entering the cycle (
    identified by in_edge), the weight will be corrected at the
```

```
        end using G
49 %          F_prime.add_edge(u, v)
50 %
51 %          # Add the remaining edges of the modified cycle C
52 %          for u_c, v_c in C.edges:
53 %              F_prime.add_edge(u_c, v_c)
54 %
55 %          # Add the external edges leaving the cycle
56 %          for _, z, _ in F_prime.out_edges(contracted_label,
    data=True):
57 %
58 %              u_cycle, _ = out_from_cycle[z]
59 %              F_prime.add_edge(u_cycle, z)
60 %
61 %          F_prime.remove_node(contracted_label)
62 %
63 %          # Update the edge weights with the original weights
    from G
64 %          for u, v in F_prime.edges:
65 %              F_prime[u][v]["w"] = D[u][v]["w"]
66 %
67 %          return F_prime
```

Notas finais sobre a implementação

A implementação acima segue diretamente a descrição do algoritmo de Chu–Liu/Edmonds, enfatizando clareza e correção. Para aplicações práticas, otimizações podem ser introduzidas, como estruturas de dados eficientes para seleção de mínimos, detecção rápida de ciclos e manipulação de grafos dinâmicos. Além disso, a função pode ser adaptada para lidar com casos especiais, como grafos desconexos ou múltiplas raízes, conforme necessário.

A complexidade da implementação direta é $O(mn)$ no pior caso, onde m é o número de arestas e n o número de vértices, devido à potencial profundidade de recursão e ao processamento linear em cada nível. Implementações mais sofisticadas podem reduzir isso para $O(m \log n)$ usando estruturas avançadas, como heaps e union-find, mas a versão apresentada prioriza a compreensão do algoritmo fundamental.

Decisões de projeto e implicações práticas

Antes de prosseguir para uma visão alternativa do mesmo problema, vale destacar algumas decisões de projeto e implicações práticas da implementação de Chu–Liu/Edmonds:

- **Estruturas e efeitos colaterais:** Optamos por modificar grafos *in-place* (por exemplo, durante a normalização e a contração de ciclos) para reduzir alocações e facilitar a visualização incremental. Isso exige invariantes explícitos e cuidado com referências ativas ao grafo original.

- **Empates, paralelos e laços:** Empates são resolvidos de forma determinística/local sem afetar a otimalidade. A contração pode induzir *arcos paralelos*; preservamos apenas o de menor custo. Laços (self-loops) são descartados por construção.
- **Validação e testes:** O repositório inclui artefatos úteis para experimentação (por exemplo, `tests.py`, `test_results.csv`, `test_log.txt`). Onde um volume de grafos é gerado aleatoriamente, a função é executada e os resultados são validados são comparados com soluções de força bruta.
- **Integração com visualização e logs:** A função `draw_fn` permite registrar *snapshots* (normalização, formação de F^*, contração/expansão). O `log` facilita auditoria e depuração em execuções recursivas.
- **Extensões:** Variantes com múltiplas raízes, restrições adicionais (p.ex., proibições por partição) e empacotamento de arborescências exigem ajustes na fase de extração/expansão ou formulações via matroides.

Transição para a abordagem primal-dual

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

No capítulo seguinte, revisitaremos o mesmo problema sob uma ótica primal–dual em duas fases, proposta por András Frank. Essa perspectiva organiza a normalização via potenciais[17] $y(\cdot)$, explica os custos reduzidos e introduz a noção de cortes apertados (família laminar) como guias das contrações. Veremos como a mesma mecânica operacional (normalizar $\rightarrow$ contrair $\rightarrow$ expandir) emerge de condições duais que também sugerem otimizações e generalizações.

5 Procedimento em Duas Fases de András Frank

Contexto histórico e motivação

Os primeiros algoritmos para arborescência de custo mínimo foram propostos por Chu–Liu (1965) e Edmonds (1967). Eles introduziram a ideia de contrair ciclos e de caracterizar quando um conjunto de arcos forma uma arborescência. Mais tarde, essa mesma lógica passou a ser lida no arcabouço *primal–dual*.

[17] No contexto primal–dual, "potenciais" são valores escalares $y(v)$ atribuídos aos vértices para definir custos reduzidos $c'(u,v) = c(u,v) - y(v)$. Ajustar y desloca uniformemente os custos das arestas que entram em v, sem mudar a otimalidade global: preserva a ordem relativa entre entradas e torna "apertadas" (custo reduzido zero) as candidatas corretas, habilitando contrações e uma prova de corretude via cortes apertados.

Em 1981, András Frank apresentou um algoritmo em duas fases que resolve o problema de arborescência mínima usando técnicas primal–duais [8]. Enquanto Chu–Liu/Edmonds opera de forma mais combinatória e direta, Frank explicita a estrutura de otimização subjacente, revelando *por que* as operações funcionam e conectando-as a certificados de otimalidade. A seguir descrevemos esse método.

5.1 Descrição do Algoritmo

O algoritmo de Frank consiste em duas fases principais:

- **Fase I (dual/potenciais):**

  Elevamos os potenciais por vértice $\tilde{y}$: para cada vértice v, $\tilde{y}(v)$ é um número que funciona como um "desconto" aplicado igualmente a todas as arestas que *entram* em v, definindo o custo reduzido $c'(u,v) = c(u,v) - \tilde{y}(v)$. Aumentamos $\tilde{y}$ até que, para todo $v \neq r$, haja ao menos uma entrada *apertada* (com $c'(u,v) = 0$).

  Sempre que surgir um ciclo feito só de arcos apertados, são contraídos e continua-se no grafo menor. Para-se quando, no grafo corrente, todo $v \neq r$ tiver alguma entrada apertada.

  Saída: o subgrafo D_0 das arestas apertadas, com ao menos uma entrada por vértice distinto de r; invariantes: $c' \geq 0$ e laminaridade da família ativa.

- **Fase II (extração primal):**

  Em D_0, selecione exatamente uma entrada para cada $v \neq r$.

  Se as escolhas formarem um ciclo, contraia-o e continue a seleção na instância contraída, *sem* alterar potenciais; ao final, reexpanda cada contração removendo exatamente uma aresta interna (a que entra no vértice que recebe o arco externo).

  O resultado é uma r-arborescência; como todas as arestas escolhidas são apertadas ($c' = 0$), a otimalidade segue por complementaridade primal–dual [8, 7, 21].

O algoritmo termina quando uma r-arborescência viável é encontrada. A correção e otimalidade do método são garantidas pelas propriedades primal–duais e pela manutenção de cortes apertados durante as contrações. A seguir, explicamos por que essa abordagem é chamada de primal–dual e como as duas fases se relacionam com as formulações primal e dual do problema.

5.1.1 A intuição Primal–dual

Informalmente, o primal decide quais arcos entram para minimizar o custo total. Já o dual escolhe "descontos" (potenciais) a aplicar nos arcos, sem permitir que nenhum fique com preço negativo; para cada arco, a soma dos descontos permitidos não pode ultrapassar o seu preço original. Toda escolha viável desses descontos já estabelece um limite inferior para o menor custo possível.

Esses problemas podem ser modelados de forma natural como *programas lineares* (PLs). Portanto, vamos apresentar brevemente alguns conceitos de álgebra linear antes de descrever as formulações primal e dual.

Nota breve de conceitos e notações em álgebra linear

Para quem não está acostumado com a notação, eis um guia rápido:

- Vetores (por exemplo, x, c, y, b) são listas de números: $x = (x_1, \ldots, x_n)$. Escrever $x \geq 0$ significa "cada componente x_i é não-negativa".
- Produto interno: $c^\top x = \sum_i c_i x_i$ (já explicado na nota de rodapé) — é "soma de preços vezes quantidades".
- Matriz–vetor: Ax é outro vetor; sua j-ésima entrada é $(Ax)_j = \sum_i A_{ji}\, x_i$. Ler $Ax \geq b$ é: "para cada restrição j, a soma à esquerda é pelo menos b_j".
- Transposta: $A^\top$ troca linhas por colunas. Assim, $(A^\top y)_i = \sum_j A_{ji}\, y_j$. Ler $A^\top y \leq c$ é: "para cada variável i, os descontos somados não passam de c_i".
- Produto componente a componente (Hadamard): $x \odot z = (x_1 z_1, \ldots, x_n z_n)$. A condição $x \odot z = 0$ significa: "em cada posição, ou $x_i = 0$ ou $z_i = 0$". É assim que lemos as condições de complementaridade mais adiante.

Agora, a formulação primal–dual do problema de arborescência mínima.

Uma formulação padrão usa variáveis x_a para cada arco $a \in A$ indicando sua seleção (na versão inteira $x_a \in \{0, 1\}$; na relaxação contínua permitimos valores fracionários $0 \leq x_a \leq 1$, obtendo um programa linear (PL)):

$$\begin{aligned} \min \quad & \sum_{a \in A} c(a)\, x_a \\ \text{s.a.} \quad & \sum_{a \in \delta^-(X)} x_a \geq 1 \quad \forall\, \emptyset \neq X \subseteq V \setminus \{r\}, \\ & x_a \geq 0 \qquad\qquad \forall\, a \in A. \end{aligned}$$

O problema dual associa uma variável $y(X) \geq 0$ a cada corte $\emptyset \neq X \subseteq V \setminus \{r\}$ e toma a forma:

$$\begin{aligned}
\max \quad & \sum_{\emptyset \neq X \subseteq V \setminus \{r\}} y(X) \\
\text{s.a.} \quad & \sum_{X:\ v \in X,\ u \notin X} y(X) \leq c(u,v) \quad \forall\ (u,v) \in A, \\
& y(X) \geq 0 \qquad\qquad\qquad\qquad \forall\, X.
\end{aligned}$$

As expressões acima conduzem diretamente às noções de *custos reduzidos*

$$c'(u,v) = c(u,v) - \sum_{X:\, v \in X, u \notin X} y(X)$$

e justificam a estratégia primal–dual (elevar potenciais/contrações de ciclos) explorada por Frank.

Em fórmulas, no primal, minimizamos $c^\top x$[18] com x dentro do conjunto viável:

$$P \;=\; \{x \in \mathbb{R}^n :\; Ax \geq b,\; x \geq 0\},$$

onde A reúne as restrições, b é a "demanda" de cada restrição e c são os custos. No dual, escolhemos multiplicadores $y \geq 0$ que respeitam

$$A^\top y \;\leq\; c.$$

Assim, para cada variável (ou arco), a soma dos "descontos" aplicáveis não pode passar do seu preço original. No caso de grafos, para um arco (u,v), isso vira

$$\sum_{X:\, v \in X,\ u \notin X} y(X) \;\leq\; c(u,v),$$

ou seja: some os $y(X)$ de todos os cortes X que contêm v e não contêm u; esse total é o "desconto máximo" permitido para (u,v). Isso garante que os custos reduzidos $c' = c - A^\top y$ fiquem sempre não negativos. Com isso, vale sempre (dualidade fraca): para todo x viável e y viável,

$$c^\top x \;\geq\; y^\top b.$$

Aqui, $y^\top b = \sum_j y_j b_j$ é a "soma de preços (y) vezes demandas (b)", que serve como limitante inferior para o melhor custo primal. Quando ambos são ótimos, os valores empatam (dualidade forte): $c^\top x = y^\top b$.

Agora, a correspondência com o algoritmo: define-se o "custo reduzido" como o preço após descontos:

$$c_{\text{red}} \;=\; c - A^\top y.$$

As condições de complementaridade dizem que o primal e o dual "se encontram no limite":

$$x \odot (c - A^\top y) = 0 \quad \text{e} \quad y \odot (Ax - b) = 0,$$

onde $\odot$ é produto por componente.

[18] Produto interno (soma ponderada) entre os vetores c e x: $c^\top x = \sum_i c_i x_i$. No nosso contexto, c_i é o custo de um arco e $x_i \in \{0,1\}$ indica se o arco é escolhido; logo $c^\top x$ é o custo total da solução.

De forma resumida:

(i) só escolhemos ($x_i > 0$) arcos cujo custo reduzido ficou exatamente zero — arcos "apertados";

(ii) só atribuímos peso dual ($y_j > 0$) às restrições que ficam exatamente justas (valem com igualdade). É por isso que elevar potenciais até criar arestas de custo reduzido zero guia a construção da solução.

5.1.2 Laminaridade, potenciais e contrações

Com variáveis duais $y(X)$ associadas a cortes, podemos, por um argumento de *uncrossing*[19], assumir sem perda de generalidade que a família ativa

$\mathcal{L} = \{X : y(X) > 0\}$

é *laminar*: quaisquer dois conjuntos são aninhados (um contém o outro) ou disjuntos, nunca "se cortam" parcialmente.

Essa laminaridade organiza os "preços por corte" em uma hierarquia e permite acumulá-los "por vértice": $\tilde{y}(v) = \sum_{X \in \mathcal{L}:\, v \in X} y(X)$. Com isso, os custos reduzidos podem ser escritos de forma direta

$$c'(u,v) \;=\; c(u,v) \;-\; \sum_{X \in \mathcal{L}:\, v \in X,\; u \notin X} y(X),$$

e a própria hierarquia dos X guia as contrações: blocos (conjuntos) *apertados* correspondem a componentes que podemos contrair e depois reexpandir. Em resumo: descruzar $\Rightarrow$ família laminar $\Rightarrow$ estrutura simples para calcular c' e conduzir contrações/expansões com eficiência [7, 21].

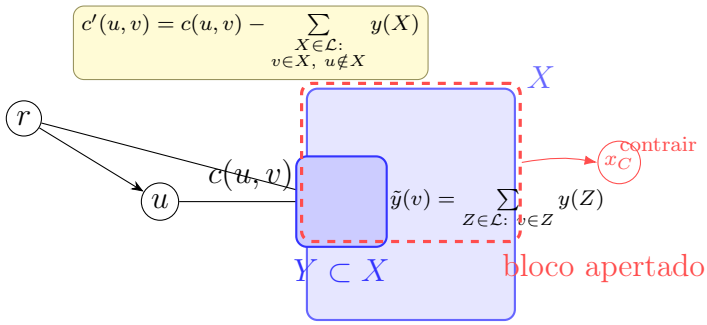


Figura 54: Família laminar de cortes ($Y \subset X$), potenciais por vértice ($\tilde{y}$), custo reduzido em uma aresta e a ideia de contrair um bloco *apertado*.

Assim, devemos nos preocupar com as operações de elevar potenciais, manter custos reduzidos não negativos, identificar cortes apertados e conduzir contrações/expansões. A seguir, detalhamos esses pontos e apresentamos o algoritmo completo.

[19] "Uncrossing" (descruzamento) troca dois conjuntos que se cruzam, X e Y, por $X \cap Y$ e $X \cup Y$. Isso preserva a viabilidade das desigualdades por cortes e não diminui o valor dual. Repetindo, chega-se a uma família sem cruzamentos (*laminar*). Ver [21, 7].

5.1.3 Potenciais por vértice

Na prática, manteremos um único número por vértice, o *potencial* $\tilde{y}(v)$, que funciona como um "desconto" aplicado a todas as arestas que entram em v:

$$c'(u, v) = c(u, v) - \tilde{y}(v).$$

Se $\tilde{y}$ vier do dual por cortes, ele é a soma dos $y(X)$ dos conjuntos laminares X que contêm v. Ao aumentar $\tilde{y}(v)$, todos os arcos que entram em v ficam igualmente mais baratos na mesma medida; a ordem entre as entradas de v não muda. Por isso, resolver com c ou com c' é equivalente. Na Fase I, elevamos $\tilde{y}$ até que todo $v \neq r$ tenha ao menos uma entrada com $c'(u, v) = 0$.

5.1.4 Cortes apertados e laminaridade

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

- Um arco (u, v) é *apertado* quando $c'(u, v) = 0$ (folga zero).
- Um corte X é *apertado* quando a restrição está *justa*: $\sum_{a \in \delta^-(X)} x_a = 1$ no primal e $y(X) > 0$ no dual.

Pelo descruzamento (*uncrossing*; ver nota acima), existe solução dual ótima cuja família ativa $\mathcal{L} = \{X : y(X) > 0\}$ é *laminar* (os conjuntos não se cruzam). Essa estrutura guia contrações eficientes [7, 21].

extbfComo usamos no algoritmo. Fase I: elevamos potenciais até que todo $v \neq r$ tenha uma entrada apertada; sempre que surgir um ciclo só de apertados, contraímos e continuamos no grafo menor, mantendo $c' \geq 0$. Fase II: extraímos a arborescência usando apenas arcos apertados, contraindo/expandindo ciclos quando necessário.

Em resumo, primeiro criamos "entradas de custo efetivo zero" elevando potenciais; se aparecerem ciclos só de zeros, contraímos. Depois, escolhemos apenas arcos de custo reduzido zero para formar a arborescência. Como todas as escolhas finais são *apertadas* em relação a y, a otimalidade decorre por complementaridade.

As Fases I e II estão formalizadas nos Alg. 5.1 e 5.2 [7].

Algoritmo 5.1: Frank: fase I

Entrada: digrafo $D = (V, A)$, custos $c : A \to \mathbb{R}$, raiz r.

Fase I — normalização primal–dual (potenciais e cortes).

1. Inicialize potenciais $y(v) \leftarrow 0$ para todo $v \in V$. Defina custos reduzidos $c'(u, v) \leftarrow c(u, v) - y(v)$ e mantenha o subgrafo D_0 apenas com arcos (u, v) tais que $c'(u, v) = 0$ (arcos *apertados*).
2. Enquanto existir $v \neq r$ sem arco de custo zero *entrando* em v em D_0:
   (a) Seja $X \leftarrow \mathrm{Anc}_{D_0}(v) \cup \{v\}$ (os ancestrais de v por arcos de D_0, mais v).
   (b) Calcule $\Delta \leftarrow \min\{\, c'(u, x) :\ u \notin X,\ x \in X \,\} = \min c'(\delta^-(X))$ e *eleve* os potenciais dos vértices de X: para todo $x \in X$, faça $y(x) \leftarrow y(x) + \Delta$.

Isso torna $c'(u,x) \leftarrow c'(u,x) - \Delta$ para arcos que entram em X e preserva c' para os demais.

(c) Adicione a D_0 todos os arcos que zerarem (novos $c'(u,x) = 0$ com $u \notin X, x \in X$). Se D_0 passar a conter um ciclo dirigido C apenas de arcos de custo reduzido zero, contraia C em um supervértice x_C; redirecione arestas incidentes preservando a regra de custos reduzidos e registre mapeamentos para a reexpansão.

3. Termine a Fase I quando, após eventuais contrações, todo $v \neq r$ tiver ao menos uma entrada com $c'(u,v) = 0$ no grafo corrente.

A Fase I termina quando, após eventuais contrações, todo $v \neq r$ tiver ao menos uma entrada com $c'(u,v) = 0$ no grafo corrente. Nesse ponto, o subgrafo D_0 de arestas de custo reduzido zero pode conter ciclos, mas cada vértice (exceto a raiz) tem ao menos uma entrada. A Fase II extrai uma r-arborescência desse subgrafo, tratando ciclos por contração e, no retorno, expandindo-os adequadamente.

Algoritmo 5.2: Frank: fase II

Fase II — extração sobre o subgrafo de zeros.

1. No grafo (possivelmente contraído), selecione para cada $v \neq r$ exatamente um arco (u,v) com $c'(u,v) = 0$. Evite ciclos; se um ciclo surgir, contraiao e prossiga no grafo menor.
2. Ao concluir a seleção (no grafo reduzido), expanda as contrações em ordem inversa. Em cada expansão de um componente X_C:
   (a) mantenha o arco externo (de custo reduzido zero) que *entra* em X_C como a única entrada do vértice de entrada correspondente;
   (b) remova exatamente uma aresta interna do ciclo de X_C para restaurar grau de entrada igual a 1 e garantir aciclicidade.
3. O resultado é uma r-arborescência T composta apenas por arcos *apertados*. Pela complementaridade primal–dual com os potenciais y, T é ótima para os custos originais c [7, 21].

5.2 Corretude

Provamos por primal–dual com contrações. Recorde as formulações primal/dual por cortes e os *custos reduzidos* $c'(u,v) = c(u,v) - \sum_{X:\, v \in X,\ u \notin X} y(X)$. Mantemos os seguintes invariantes ao longo da Fase I:

- (I1 — dual) y é viável: $c'(u,v) \geq 0$ para todo arco.
- (I2 — zeros) D_0 contém exatamente os arcos *apertados* ($c' = 0$).
- (I3 — laminaridade) A família ativa $\mathcal{L} = \{X : y(X) > 0\}$ pode ser tomada laminar por *uncrossing*.

Lema 1: elevação de potenciais:

Seja $X \subseteq V \setminus \{r\}$ e $\Delta = \min c'(\delta^-(X))$. Atualizar $y(X) \leftarrow y(X) + \Delta$ (ou, na versão por vértices, $\tilde{y}(x) \leftarrow \tilde{y}(x) + \Delta$ para todo $x \in X$) mantém $c' \geq 0$ e torna *apertado* ao menos um arco que entra em X.

Prova: Para $(u, x) \in \delta^-(X)$, c' é reduzido em Δ; pela definição de Δ, nenhum fica negativo e os de menor folga zeram. Os demais arcos não mudam. □

Lema 2: contração de ciclo a zero:

Se D_0 contém um ciclo dirigido C de arcos com $c' = 0$, então contrair C em um supervértice preserva viabilidade primal e dual e há uma bijeção de soluções ótimas entre a instância contraída e a original; ao reexpandir, basta manter o arco (de custo reduzido zero) que entra no componente e remover uma única aresta interna de C.

Prova: Todo arco interno de C tem $c' = 0$. Ao contrair, redirecionamos apenas arcos que entram/saem de C, preservando $c' \geq 0$. Dado um r-arborescência ótima T' no grafo contraído, reexpandir C e remover um arco interno restaura grau de entrada 1 e aciclicidade. Reciprocamente, contrair qualquer T ótimo produz T' viável no digrafo menor. A soma dos custos reduzidos é preservada porque os internos de C valem 0 e entradas em C descontam exatamente o potencial aplicado, mantendo equivalência de custos [21, Sec. 4.9]. □

Lema 3: cortes ativos entram exatamente uma vez

Seja T uma r-arborescência e $X \subseteq V \setminus \{r\}$. Se contraímos X a um vértice, o grau de entrada desse vértice na imagem de T é 1; logo $\sum_{a \in \delta^-(X)} x_a = 1$. Em particular, para todo $X \in \mathcal{L}$ com $y(X) > 0$, a restrição primal do corte é *justa*.
Prova. Em uma r-arborescência, todo vértice distinto de r tem grau de entrada 1. Após contrair X, o vértice contraído também não é r e deve ter grau de entrada 1. Isso conta exatamente um arco de $\delta^-(X)$. □

Teorema: o algoritmo devolve uma r-arborescência ótima para c

Prova: Ao término da Fase I (após contrações de Lema 2), todo $v \neq r$ possui ao menos uma entrada com $c'(u, v) = 0$. A Fase II seleciona apenas arcos *apertados* e trata ciclos por contração/expansão, produzindo uma r-arborescência T. Logo, para todo $(u, v) \in T$, a desigualdade dual do arco é *justa* ($c'(u, v) = 0$). Pelo Lema 3, para todo $X \in \mathcal{L}$ com $y(X) > 0$, a restrição primal do corte é *justa* (entra exatamente um arco de T). Assim, valem as condições de *complementaridade*:

$$x_{(u,v)} > 0 \;\Rightarrow\; c'(u, v) = 0, \qquad y(X) > 0 \;\Rightarrow \sum_{a \in \delta^-(X)} x_a = 1.$$

Pela identidade $c(u,v) = c'(u,v) + \sum_{X:\, v\in X,\ u\notin X} y(X)$, somando em $a \in T$ e usando as igualdades acima:

$$\sum_{a\in T} c(a) \;=\; \underbrace{\sum_{a\in T} c'(a)}_{=\,0} \;+\; \sum_{X} y(X) \underbrace{\sum_{a\in T\cap \delta^-(X)} 1}_{=\,1 \ \text{ para } X\in\mathcal{L}} \;=\; \sum_{X} y(X).$$

Como y é dual viável (I1), $\sum_X y(X)$ é um limitante inferior (dualidade fraca). Obtemos igualdade primal–dual, logo T é ótimo para c' e, pela equivalência entre c e c', também para c [7, 21]. □

Complexidade

- **Por nível:** manter o subgrafo de zeros D_0, calcular o próximo $\Delta = \min c'(\delta^-(X))$ e detectar/contrair ciclos custa $O(m)$ por varredura nas arestas.
- **Níveis:** no máximo $O(n)$, pois cada contração reduz o número de vértices.
- **Total/memória:** tempo $O(mn)$ e memória $O(m+n)$.

Com otimizações (atingindo $O(m \log n)$).

- **Mínimo por vértice:** para cada v, manter em uma *heap* o menor custo reduzido de entrada; atualizar chaves quando $y(v)$ aumenta (atualizações amortizadas).
- **Ciclos em D_0:** detectar incrementalmente à medida que surgem novas arestas com custo zero (DFS/union-find).
- **Contrações:** realizar redirecionamentos em bloco via representantes, sem tocar cada aresta individualmente.

O fator $\log n$ decorre da seleção eficiente de mínimos; o número de contrações segue $O(n)$ [7, 21].

Nós realizamos duas implementações: uma didática, focada na clareza do algoritmo, e outra otimizada, visando desempenho. Ambas estão disponíveis no repositório do projeto (`https://github.com/lorenypsum/GraphVisualizer`).

5.3 Implementação em Python

Implementamos duas versões do algoritmo de Frank: uma primeira versão didática, e uma versão otimizada, que emprega heaps como estrutura auxiliar. Manter ambas permitiu validar o método por meio da comparação sistemática dos resultados.

Utilizamos a biblioteca NetworkX para a manipulação de grafos e funções auxiliares para modularizar o código, o que facilita a leitura e a manutenção. A seguir, apresentamos essas funções e os detalhes de implementação de ambas as versões.

Arestas que entram em X e peso mínimo: a função `get_arcs_entering_X` recebe um digrafo D e um conjunto de vértices X, retornando uma lista de arestas que entram em X. Cada aresta é representada como uma tupla `(u, v, data)`, onde u é o vértice de origem, v é o vértice de destino e `data` contém os atributos da aresta, incluindo o peso.

Função 5.1: Arestas que entram em X e mínimo

```
1 def get_arcs_entering_X(D, X):
2     """
3     Obter as arestas que entram em um conjunto X em um
          digrafo D.
4     A função retorna uma lista de tuplas que representam as
          arestas que entram em X com seus respectivos pesos.
5
6     Parâmetros:
7     - D: digrafo (DiGraph)
8     - X: conjunto de vértices
9
10    Retorno:
11    - arcs: lista de tuplas (u, v, data) em que u ∉ X e v ∈ X
12    """
13
14    arcs = []
15
16    for u, v, data in D.edges(data=True):
17        if u not in X and v in X:
18            arcs.append((u, v, data))
19    return arcs
```

Peso mínimo de um corte: a função `get_minimum_weight_cut` recebe uma lista de arestas e retorna o peso mínimo entre elas; isso é útil para determinar o valor Δ na elevação de potenciais.

Função 5.2: Peso mínimo de um corte

```
1 def get_minimum_weight_cut(arcs):
2     """
3         Obter o peso mínimo em uma lista de arestas.
4         A função retorna o menor peso encontrado.
5
6         Parâmetros:
7         - arcs: lista de tuplas (u, v, data)
8
9         Retorno:
10        - min_weight: menor peso encontrado entre as arestas
11    """
12
13    return min(data["w"] for _, _, data in arcs)
```

Atualizar pesos em X: a função `update_weights_in_X` atualiza os pesos das arestas que entram em um conjunto X em um digrafo D. Ela subtrai um valor mínimo dos pesos

dessas arestas e registra aquelas que atingem peso zero em uma lista e em um novo digrafo. Essa função tem efeito colateral, pois modifica o digrafo original.

Função 5.3: Atualizar pesos em X

```
def update_weights_in_X(D, arcs, min_weight, A_zero, D_zero):
    """
    Update the weights of the arcs in a directed graph D for
       the nodes in set X.
    ATTENTION: The function produces collateral effect in the
        provided directed graph by updating its arcs weights.

    Parameters:
        - D: directed graph (DiGraph)
        - arcs: list of tuples (u, v, data) where u not in X
           and v in X
        - min_weight: minimum weight to be subtracted from
           the arcs weights
        - A_zero: list to store the arcs that reach weight
           zero
        - D_zero: directed graph (DiGraph) to store the arcs
           that reach weight zero

    Returns:
        - None
    """

    for u, v, _ in arcs:
        D[u][v]["w"] -= min_weight
        if D[u][v]["w"] == 0:
            A_zero.append((u, v))
            D_zero.add_edge(u, v)
```

Identificar arborescência: a função `has_arborescence` verifica se um digrafo D possui uma arborescência com raiz $r0$. Ela retorna True se uma arborescência existir, caso contrário, retorna False. Isso é feito verificando se o digrafo é uma árvore de busca em profundidade (DFS) com raiz $r0$.

Função 5.4: Identificar Arborescência

```
def has_arborescence(D, r0):
    """
    Check if a directed graph D has an arborescence with root
        r0.
    The function returns True if an arborescence exists,
       otherwise False.

```

```
6       Parameters:
7           - D: directed graph (DiGraph)
8           - r0: root node
9
10      Returns:
11          - bool: True if an arborescence exists, otherwise
                False
12      """
13
14      # Verify if the graph is a DFS tree with root r0
15      tree = nx.dfs_tree(D, r0)
16
17      return tree.number_of_nodes() == D.number_of_nodes()
```

Fase 1 do algoritmo de Frank: a função `phase1_find_minimum_arborescence` mantém três estruturas: (i) `D_zero`, o subgrafo das arestas com peso 0; (ii) `A_zero`, a lista dessas 0-arestas; e (iii) `Dual_list`, com pares (X, Δ) que registram os incrementos aplicados.

O laço faz o seguinte, de forma direta:

- *Construção do condensado:* constrói o condensado $C = \mathrm{Cond}(D_0)$ via `nx.condensation(D_zero)` e coleta as *fontes* de C (componentes com grau de entrada zero)[20];
- *Fontes:* se restar apenas uma fonte (a que contém $r0$), termina;
- *Processamento das fontes:* para cada fonte u com $X = \mathrm{members}(u)$ e $r0 \notin X$:
  - `arcs = get_arcs_entering_X(D_copy, X)`;
  - $\Delta =$ `get_minimum_weight_cut(arcs)` (menor peso dentre as entradas de X);
  - `update_weights_in_X(D_copy, arcs, Δ, A_zero, D_zero)` para subtrair Δ das entradas e adicionar a `D_zero`/`A_zero` as que zerarem;
  - se $\Delta > 0$, registra (X, Δ) em `Dual_list`.

Na prática, `update_weights_in_X` é o passo que "cria" novas 0-arestas: ele reduz os pesos das entradas de X e insere em `D_zero`/`A_zero` aquelas que atingirem 0. O processo se repete até que toda componente diferente da raiz tenha ao menos uma entrada de peso 0, preparando o terreno para a Fase II.

Parâmetros opcionais `draw_fn` e `log` controlam visualização e mensagens.

[20] No grafo condensado C (um DAG cujos nós são as componentes fortemente conexas de D_0), uma "fonte" é um nó com grau de entrada zero. Isso corresponde a um bloco de D_0 que não recebe arcos de custo reduzido zero vindos de fora do próprio bloco. Na Fase I, elevamos potenciais apenas para fontes distintas da que contém r_0 para garantir ao menos uma entrada apertada.

Função 5.5: Fase 1 - algoritmo de Frank

```
1 def phase1_find_minimum_arborescence(
2     D_original, r0, draw_fn=None, log=None, boilerplate: bool
          = True, lang="pt"
3 ):
4     """
5     Find the minimum arborescence in a directed graph D with
         root r0.
6     The function returns the minimum arborescence as a list
         of arcs.
7
8     Parameters:
9         - D_original: directed graph (DiGraph)
10        - r0: root node
11
12    Returns:
13        - A_zero: list of arcs (u, v) that form the minimum
             arborescence
14        - Dual_list: list of tuples (X, z(X)) representing
             the dual variables
15    """
16
17    D_copy = D_original.copy()
18    A_zero = []
19    Dual_list = []  # List to store the dual variables (X, z(
         X))
20    D_zero = build_D_zero(D_copy)
21
22    iteration = 0
23
24    if boilerplate and draw_fn:
25        if lang == "en":
26            draw_fn(D_zero, title="Initial D_zero")
27        elif lang == "pt":
28            draw_fn(D_zero, title="D_zero Inicial")
29
30    while True:
31        iteration += 1
32        if boilerplate and log:
33            if lang == "en":
34                log(f"\nIteration {iteration} ----------------
                       -------------")
35            elif lang == "pt":
36                log(f"\nIteração {iteration} ---------------
                       ------------")
37
38        # Calculate the strongly connected components of the
             graph D_zero.
```

```
39          C = nx.condensation(D_zero)
40          if boilerplate and draw_fn:
41              if lang == "en":
42                  draw_fn(
43                      C,
44                      title=f"Strongly connected components in
                            D_zero - Iteration {iteration}",
45                  )
46              elif lang == "pt":
47                  draw_fn(
48                      C,
49                      title=f"Componentes fortemente conexos em
                            D_zero - Iteração {iteration}",
50                  )
51
52          # The sources are where there are no incoming arcs,
              R0 is always a source.
53          sources = [x for x in C.nodes() if C.in_degree(x) ==
              0]
54
55          if boilerplate and log:
56              if lang == "en":
57                  log(f"\nSources: {sources}")
58              elif lang == "pt":
59                  log(f"\nFontes: {sources}")
60
61          if len(sources) == 1:
62              # If there is only one source, it means it is R0
                  and there are no more arcs to be processed.
63              if boilerplate and log:
64                  if lang == "en":
65                      log(f"\nOnly one source found, algorithm
                            finished.")
66                  elif lang == "pt":
67                      log(f"\nApenas uma fonte encontrada,
                            algoritmo finalizado.")
68              break
69
70          for u in sources:
71              X = C.nodes[u]["members"]
72              if r0 in X:
73                  continue
74              arcs = get_arcs_entering_X(D_copy, X)
75              min_weight = get_minimum_weight_cut(arcs)
76
77              if boilerplate and log:
78                  if lang == "en":
79                      log(f"\nSet X: {X}")
80                      log(f"\nArcs entering X: {arcs}")
```

```
81                      log(f"\nMinimum weight found: {min_weight
                            }")
82                  elif lang == "pt":
83                      log(f"\nConjunto X: {X}")
84                      log(f"\nArestas que entram em X: {arcs}")
85                      log(f"\nPeso mínimo encontrado: {
                            min_weight}")
86
87              update_weights_in_X(D_copy, arcs, min_weight,
                    A_zero, D_zero)
88
89              if boilerplate and log:
90                  if lang == "en":
91                      log(f"\nUpdated weights in arcs entering
                            X")
92                  elif lang == "pt":
93                      log(f"\nPesos atualizados nos arcos que
                            entram em X")
94
95              # If min_weight is zero, ignore
96              if min_weight == 0:
97                  continue
98              else:
99                  # Otherwise, add to the dual list the set X
                        and its min_weight
100                 Dual_list.append((X, min_weight))
101
102     return A_zero, Dual_list
```

Transição para a extração: Fase 2

Ao término da Fase 1, o algoritmo de Frank produziu um subgrafo D_0 contendo apenas arcos de custo reduzido zero, com a garantia de que cada vértice $v \neq r$ possui ao menos uma entrada nesse subgrafo. Este é o resultado concreto dos ajustes de potenciais: transformamos o problema original em um grafo "zerado" onde sabemos que existe uma arborescência ótima.

A Fase 2 explora essa estrutura preparada: em vez de continuar ajustando custos ou potenciais, simplesmente extraímos uma arborescência usando exclusivamente os arcos apertados de D_0. A separação entre as fases é clara: a Fase 1 prepara o terreno dual, e a Fase 2 colhe o resultado primal. Esta divisão não apenas simplifica a análise de corretude (todos os arcos escolhidos são apertados, satisfazendo complementaridade), mas também permite otimizações específicas em cada etapa.

Fase 2 - algoritmo de Frank a função `phase2_find_minimum_arborescence` recebe `D_original`, `r0` e `A_zero` e devolve `Arb` (um `DiGraph`) contendo a arborescência construída apenas com 0-arestas. O fluxo é:

- *Inicialização:* `Arb = nx.DiGraph(); Arb.add_node(r0); n = len(D_original.nodes())`.
- *Laço externo:* `for _ in range(n-1)` força a inclusão de exatamente $n-1$ arestas.
- *Laço interno (varredura de candidatos):* percorre `A_zero` na ordem dada; para cada (u, v), se `u in Arb.nodes()` e `v not in Arb.nodes()`, então
  - lê atributos originais com `edge_data = D_original.get_edge_data(u,v)`;
  - insere `Arb.add_edge(u, v, **edge_data)`;
  - faz `break` para reiniciar a varredura desde o início de `A_zero` na próxima iteração (crescimento em camadas a partir do conjunto já alcançado).
- *Invariantes práticos:* o teste `v not in Arb.nodes()` evita ciclos e mantém grau de entrada ≤ 1 por vértice; `u in Arb.nodes()` garante que sempre expandimos a partir de vértices já alcançados por $r0$.
- *Hipóteses sobre a entrada:* `A_zero` deve conter ao menos uma entrada para cada $v \neq r0$; a ordem em `A_zero` funciona como critério de desempate e pode alterar qual arborescência ótima é retornada, sem afetar o custo.
- *Custo:* no pior caso, $O(n \cdot |A\_zero|)$, pois a cada inclusão recomeçamos a varredura de `A_zero`.

Parâmetros opcionais `draw_fn` e `log` controlam visualização e mensagens.

Função 5.6: Fase 2: algoritmo de Frank - versão 1 (sem heap)

```
def phase2_find_minimum_arborescence(
    D_original, r0, A_zero, draw_fn=None, log=None,
       boilerplate: bool = True, lang="pt"
):
    """
    Encontra a arborescência mínima em um grafo dirigido D
       com raiz r0.
    A função retorna a arborescência mínima como um DiGraph.

    Parâmetros:
        - D_original: grafo dirigido (DiGraph)
        - r0: vértice raiz
        - A_zero: lista de arcos (u, v) que formam a arboresc
           ência mínima

    Retorno:
        - Arb: grafo dirigido (DiGraph) que representa a
           arborescência mínima
    """
    Arb = nx.DiGraph()

    # Add the root node
    Arb.add_node(r0)
    n = len(D_original.nodes())
```

```
21
22      # While there are arcs to be considered
23      for _ in range(n - 1):
24          for u, v in A_zero:
25              if u in Arb.nodes() and v not in Arb.nodes():
26                  edge_data = D_original.get_edge_data(u, v)
27                  Arb.add_edge(u, v, **edge_data)
28                  # Restart the loop after adding an edge
29                  break
30          if boilerplate and draw_fn:
31              if lang == "en":
32                  draw_fn(Arb, title=f"Partial arborescence -
                        Iteration {_+1}")
33              elif lang == "pt":
34                  draw_fn(Arb, title=f"Arborescência parcial -
                        Iteração {_+1}")
35      return Arb
```

Fase 2 - algoritmo de Frank: versão 2 (com heap) extbfImplementação com fila de prioridade. Nesta variação, mantemos uma *fronteira* de arcos candidatos em uma heap (`heapq`) e sempre extraímos o próximo arco com menor prioridade. O código constrói um grafo auxiliar apenas com as 0-arestas e usa um conjunto de visitados para garantir que cada vértice entre exatamente uma vez.

- *Pré-processamento (grafo auxiliar):* criar `Arb = nx.DiGraph()` e inserir todas as 0-arestas com um peso de prioridade indexado: `for i, (u,v) in enumerate(A_zero): Arb.add_edge(u, v, w=i)`. Aqui, `w` atua como *chave de desempate* estável baseada na ordem de `A_zero`.
- *Estados:* `V = {r0}` (vértices alcançados), `q = []` (heap de tuplas `(w, u, v)`), `A = nx.DiGraph()` (arborescência resultante).
- *Semeadura da fronteira:* para cada `(u, v, data)` em `Arb.out_edges(r0, data=True)`, empilhar `heapq.heappush(q, (data["w"], u, v))`.
- *Laço principal:* enquanto `q` não estiver vazia,
  - extrair `_, u, v = heapq.heappop(q)`;
  - se `v in V`, continuar (descartar arcos que levariam a um vértice já escolhido);
  - adicionar a aresta real com atributos originais: `edge_data = D_original.get_edge_data(u, v); A.add_edge(u, v, **edge_data)`;
  - marcar `V.add(v)`;
  - para cada `(v, w, data)` em `Arb.out_edges(v, data=True)`, se `w not in V`, empilhar `heapq.heappush(q, (data["w"], v, w))`.
- *Propriedades:* cada vértice entra uma única vez em `V` (grau de entrada ≤ 1 por vértice), só usamos arcos de `A_zero` e evitamos ciclos por construção. O processo termina com $|V| = |V(\texttt{D\_original})|$ se `A_zero` cobre uma entrada para todo $v \neq r0$.

- *Desempate/prioridade:* a chave `w=i` preserva a ordem de `A_zero`. Se desejar priorizar por custos originais, substitua `data["w"]` por $c(u, v)$ (ou por uma tupla `(c(u,v), i)` para estabilidade).
- *Complexidade:* $O(|\texttt{A\_zero}| \log |\texttt{A\_zero}|)$ para operações de heap, mais $O(|\texttt{A\_zero}|)$ para construir `Arb`.

Função 5.7: Fase 2: algoritmo de Frank - versão 2 (com heap)

```
1 def phase2_find_minimum_arborescence_v2(
2     D_original, r0, A_zero, draw_fn=None, log=None,
         boilerplate: bool = True, lang="pt"
3 ):
4     """
5     Encontra a arborescência mínima em um grafo dirigido D
         com raiz r0.
6     A função retorna a arborescência mínima como um DiGraph.
7
8     Parâmetros:
9         - D_original: grafo dirigido (DiGraph)
10        - r0: vértice raiz
11        - A_zero: lista de arcos (u, v) que formam a arboresc
             ência mínima
12
13    Retorno:
14        - Arb: grafo dirigido (DiGraph) que representa a
             arborescência mínima
15    """
16    Arb = nx.DiGraph()
17    for i, (u, v) in enumerate(A_zero):
18        Arb.add_edge(u, v, w=i)
19
20    # Set of visited vertices, starting with the root
21    V = {r0}
22
23    # Priority queue to store the edges
24    q = []
25    for u, v, data in Arb.out_edges(r0, data=True):
26
27        # Add edges to the priority queue with their weights
28        heapq.heappush(q, (data["w"], u, v))
29
30    A = nx.DiGraph()  # Arborescência resultante
31
32    if boilerplate and draw_fn:
33        if lang == "en":
34            draw_fn(Arb, title=f"Initial arborescence with
                 weights - Phase 2")
35        elif lang == "pt":
36            draw_fn(Arb, title=f"Arborescência inicial com
```

```
                pesos - Fase 2")
37
38      # While the queue is not empty
39      while q:
40          _, u, v = heapq.heappop(q)
41
42          if v in V:  # If the vertex has already been visited,
                continue
43              continue
44
45          # Add the edge to the arborescence
46          A.add_edge(u, v, w=D_original[u][v]["w"])
47
48          # Mark the vertex as visited
49          V.add(v)
50
51          # Add the outgoing edges of the visited vertex to the
                priority queue
52          for x, y, data in Arb.out_edges(v, data=True):
53              heapq.heappush(q, (data["w"], x, y))
54
55      if boilerplate and draw_fn:
56          if lang == "en":
57              draw_fn(A, title=f"Final arborescence - Phase 2")
58          elif lang == "pt":
59              draw_fn(A, title=f"Arborescência final - Fase 2")
60      # Return the resulting arborescence
61      return A
```

Checar condição de otimalidade dual: a função `check_dual_optimality_condition` verifica a condição dual: se $z(X) > 0$, então exatamente uma aresta de `Arb` entra em X. Ela recebe `Arb` (uma arborescência), `Dual_list` (lista de tuplas $(X, z(X))$), e retorna True se a condição for satisfeita, False caso contrário. O fluxo é:

- para cada (X, z) em `Dual_list`:
  - para cada (u, v) em `Arb.edges()`:
    - se $u \notin X$ e $v \in X$, incrementar contador `count`;
    - se `count > 1`, a condição falha: logar mensagem (se `boilerplate` e `log` estiverem ativos) e retornar False.
- se o laço terminar sem falhas, retornar True.
- complexidade $O(|\texttt{Dual\_list}| \cdot |\texttt{Arb.edges()}|)$ no pior caso.

Função 5.8: Checar condição de otimalidade dual

```
1 def check_dual_optimality_condition(
```

```
    Arb, Dual_list, log=None, boilerplate: bool = True, lang="pt"
):
    """
    Verifica a condição dual: z(X) > 0 implica que exatamente uma aresta de Arb entra em X.

    Parameters:
        - Arb: arborescência (DiGraph)
        - Dual_list: lista de tuplas (X, z(X)) representando as variáveis duais
        - r0: nó raiz

    Retorno:
        - bool: True se a condição dual é satisfeita, False caso contrário
    """
    for X, z in Dual_list:
        for u, v in Arb.edges():
            count = 0
            if u not in X and v in X:
                count += 1
                if count > 1:
                    if boilerplate and log:
                        if lang == "en":
                            log(
                                f"\nDual condition failed for X={X} with z(X)={z}. Incoming arcs: {count}"
                            )
                        elif lang == "pt":
                            log(
                                f"\nFalha na condição dual para X={X} com z(X)={z}. Arcos entrando: {count}"
                            )
                    return False
    return True
```

Rotina principal: a função `andras_frank_algorithm` orquestra a execução das fases 1 e 2, além da verificação da condição dual. Ela recebe o digrafo D, a raiz $r0$ (fixa), e parâmetros opcionais `draw_fn`, `log`, `boilerplate` e `lang` para controle de visualização, mensagens e idioma. O fluxo é:

- `A_zero, Dual_list = phase1_find_minimum_arborescence()` executa a Fase I;
- se `not has_arborescence(D, r0)`, loga mensagem e retorna `None, None`;
- `arborescence_frank = phase2_find_minimum_arborescence()` executa a Fase II (versão 1);

- `arborescence_frank_v2 = phase2_find_minimum_arborescence_v2()` executa a Fase II (versão 2);
- `dual_frank = check_dual_optimality_condition()` verifica a condição dual para a versão 1;
- `dual_frank_v2 = check_dual_optimality_condition()` verifica a condição dual para a versão 2;
- loga mensagens de sucesso/falha conforme `dual_frank` e `dual_frank_v2`;
- retorna `arborescence_frank, arborescence_frank_v2, dual_frank, dual_frank_v2`.

Função 5.9: Rotina Principal: chamada das funções

```
def andras_frank_algorithm(
    D, draw_fn=None, log=None, boilerplate: bool = True, lang="pt"
):
    if boilerplate and log:
        if lang == "en":
            log(f"\nExecuting András Frank algorithm...")
        elif lang == "pt":
            log(f"\nExecutando algoritmo de András Frank...")

    A_zero, Dual_list = phase1_find_minimum_arborescence(
        D, "r0", draw_fn=draw_fn, log=log, boilerplate=boilerplate, lang=lang
    )

    if boilerplate and log:
        log(f"\nA_zero: \n{A_zero}")
        log(f"\nDual_list: \n{Dual_list}")

    if not has_arborescence(D, "r0"):
        if boilerplate and log:
            if lang == "en":
                log(f"\nThe graph does not contain an arborescence with root r0.")
            elif lang == "pt":
                log(f"\nO grafo não contém uma arborescência com raiz r0.")
        return None, None

    arborescence_frank = phase2_find_minimum_arborescence(
        D, "r0", A_zero, draw_fn=draw_fn, log=log, boilerplate=boilerplate, lang=lang
    )
    arborescence_frank_v2 = phase2_find_minimum_arborescence_v2(
        D, "r0", A_zero, draw_fn=draw_fn, log=log, boilerplate=boilerplate, lang=lang
```

```
    )

    dual_frank = check_dual_optimality_condition(
        arborescence_frank, Dual_list, log=log, boilerplate=
           boilerplate, lang=lang
    )

    dual_frank_v2 = check_dual_optimality_condition(
        arborescence_frank_v2, Dual_list, log=log,
           boilerplate=boilerplate, lang=lang
    )

    if dual_frank and dual_frank_v2:
        if boilerplate and log:
            if lang == "en":
                log(f"\n✓ Dual condition satisfied for András
                    Frank.")
            elif lang == "pt":
                log(f"\n✓ Condição dual satisfeita para Andrá
                   s Frank.")
    else:
        if boilerplate and log:
            if lang == "en":
                log(f"\n× Dual condition failed for András
                   Frank.")
            elif lang == "pt":
                log(f"\n× Condição dual falhou para András
                   Frank.")

        if draw_fn:
            if boilerplate and draw_fn:
                if lang == "en":
                    draw_fn(
                        arborescence_frank,
                        title="András Frank Arborescence -
                           Method 1",
                    )
                    draw_fn(
                        arborescence_frank_v2,
                        title="András Frank Arborescence -
                           Method 2",
                    )
                elif lang == "pt":
                    draw_fn(
                        arborescence_frank,
                        title="Arborescência de András Frank
                           - Método 1",
                    )

```

```
71                      draw_fn(
72                          arborescence_frank_v2,
73                          title="Arborescência de András Frank
                                - Método 2",
74                      )
75
76      return arborescence_frank, arborescence_frank_v2,
            dual_frank, dual_frank_v2
```

Notas finais

- A implementação assume que D é conexo e contém uma raiz $r0$ de onde todos os outros vértices são alcançáveis.
- A ordem em `A_zero` pode influenciar qual arborescência ótima é retornada, mas não o custo.
- A verificação da condição dual é uma etapa de validação adicional, não necessária para a construção da arborescência.
- Parâmetros opcionais permitem controle sobre visualização e mensagens, facilitando depuração e entendimento do processo.

Das implementações à comparação sistemática

De fato, o algoritmo de Frank é uma abordagem elegante e eficiente para encontrar arborescências de custo mínimo em digrafos, combinando técnicas de programação linear, teoria dos grafos e estruturas de dados. A implementação acima captura os principais passos do algoritmo, permitindo a exploração prática dessa técnica.

Mas ter duas implementações funcionais — Chu–Liu/Edmonds e Frank — levanta questões naturais: *como elas se relacionam conceitualmente?* Ambas resolvem o mesmo problema, mas suas organizações internas diferem. Chu–Liu/Edmonds opera de forma mais direta e combinatória, enquanto Frank explicita a estrutura dual subjacente. Essas diferenças conceituais se traduzem em diferenças práticas de desempenho, estruturas de dados ou facilidade de compreensão?

Antes de responder empiricamente (via testes), vale consolidar uma visão comparativa *teórica* dessas abordagens. A próxima seção examina lado a lado as semelhanças e diferenças entre Chu–Liu/Edmonds e Frank, preparando o terreno para a análise experimental que virá em seguida.

6 Discussão e Resultados: Chu-liu/Edmonds vs. Frank

Wittgenstein, em suas investigações filosóficas, propôs a ideia de "semelhança de família" que diz que coisas que pertencem ao mesmo grupo não compartilham necessariamente uma característica definidora, mas sim uma série de características que se sobrepõem de

maneiras variadas. Assim, membros de uma "família" podem ser semelhantes em alguns aspectos, mas diferentes em outros, criando uma rede complexa de relações.

Chu–Liu/Edmonds e Frank são parentes próximos: perseguem o mesmo objetivo com blocos operacionais semelhantes, mas organizam o enredo de modos distintos — um com voz combinatória, o outro com abordagem primal–dual.

Em linhas gerais, os dois métodos são equivalentes quanto ao resultado (ambos devolvem uma r-arborescência de custo mínimo), mas organizam a mecânica de normalizar → contrair → expandir de maneiras distintas:

O algoritmo de Chu–Liu/Edmonds [3, 6] é o método clássico para encontrar uma arborescência de custo mínimo em um digrafo com pesos arbitrários. Ele funciona recursivamente, escolhendo a melhor entrada para cada vértice, detectando ciclos e contraindo-os, ajustando os custos conforme necessário. O processo se repete até que não haja mais ciclos, momento em que a arborescência é extraída.

Já o método de Frank [8, 7] adota uma abordagem primal–dual, elevando potenciais para criar um subgrafo de arcos de custo reduzido zero e, em seguida, extraindo a arborescência apenas com esses arcos. Ciclos são tratados por contração/expansão, mas sem novos ajustes de custos após a fase inicial.

As principais diferenças e semelhanças entre os dois métodos podem ser resumidas assim:

- **Paradigma e estrutura:** *Chu–Liu/Edmonds* é apresentado de forma mais *primal/combinatória*: escolhe para cada $v \neq r$ a sua melhor entrada, forma F^*, e trata ciclos imediatamente por contração com ajuste de custos, repetindo até não haver ciclos. *Frank* explicita a visão *primal–dual* em **duas fases**: (I) elevar potenciais para induzir o subgrafo de zeros D_0 com ao menos uma entrada por vértice (apenas ajustes duais e eventuais contrações de ciclos de arcos apertados), e (II) extrair uma arborescência usando *só* arcos apertados, tratando ciclos por contração/expansão, **sem** novas alterações de potenciais [8, 7, 21].
- **Papel dos potenciais:** Em *Chu–Liu/Edmonds*, a "normalização" por vértice pode ser vista como potenciais *implícitos* (subtrair mínimos de entradas), atualizados conforme a contração progride. Em *Frank*, os potenciais $\tilde{y}$ (ou o dual por cortes laminares) são entidades *explícitas* que guiam a criação de arcos apertados e mantêm $c' \geq 0$; a laminaridade é parte da prova e da organização da fase I.
- **Estratégia de extração:** *Chu–Liu/Edmonds* extrai a arborescência ao final do processo, enquanto *Frank* a obtém diretamente do subgrafo de zeros D_0 na fase II.
- **Tratamento de ciclos:** *Chu–Liu/Edmonds* intercala extração e contração: assim que F^* tem um ciclo, contrai e ajusta custos, voltando ao mesmo fluxo. *Frank* contrai apenas ciclos de arcos de custo reduzido zero durante a fase I (quando aparecem) e, na fase II, contrai ciclos surgidos da seleção *sem* mexer nos potenciais, reexpandindo ao final com a remoção de uma única aresta interna do ciclo.

- **Foco na estrutura dual:** *Frank* enfatiza a relação primal–dual, com a família laminar de cortes ativos e a condição de complementaridade primal–dual como pilares centrais. *Chu–Liu/Edmonds* é mais direto, focando na construção combinatória da arborescência.
- **Invariantes e corretude:** *Chu–Liu/Edmonds* tradicionalmente é provado por argumentos combinatórios de custo e correção sob contrações. *Frank* ancora a prova em **complementaridade primal–dual**: ao final, todas as arestas escolhidas são *apertadas* ($c' = 0$) e cada corte ativo da família laminar é atravessado exatamente uma vez, igualando valores primal e dual.
- **Extração final:** Em *Chu–Liu/Edmonds*, a seleção "uma entrada por vértice" e a resolução de ciclos acontecem iterativamente durante todo o processo. Em *Frank*, a seleção acontece de uma vez sobre o subgrafo de zeros D_0 produzido na fase I, o que simplifica a justificativa de otimalidade por manter todas as escolhas *apertadas*.
- **Complexidade e implementações:** Ambas as abordagens, em versão direta, rodam em $O(mn)$; com estruturas adequadas, alcançam $O(m \log n)$ em variantes conhecidas. A formulação dual explícita de *Frank* facilita raciocinar sobre *cortes apertados*, laminaridade e otimizações guiadas pelo dual.
- **Resumo prático:** Pense em *Chu–Liu/Edmonds* como o roteiro combinatório clássico "escolhe mínimos → contrai ciclo → ajusta e repete". O método de *Frank* reembala a mesma mecânica sob uma ótica *primal–dual*: primeiro "zeramos" as entradas com potenciais até obter D_0, depois extraímos a arborescência apenas com arcos apertados — e é exatamente essa apertude que certifica a otimalidade.

Ambos os métodos são amplamente aplicáveis, e a escolha entre eles pode depender do contexto, da familiaridade com técnicas primal–dual ou combinatórias, e das necessidades específicas de implementação.

Estabelecidos os fundamentos teóricos e as distinções conceituais, uma questão natural emerge: *como essas diferenças se manifestam na prática?* Ambas as abordagens garantem otimalidade, mas será que apresentam desempenhos similares em instâncias reais? A organização primal–dual de Frank, com suas duas fases distintas, oferece vantagens computacionais ou apenas elegância teórica?

Para responder a essas perguntas de forma empírica, realizamos uma bateria de testes sistemáticos comparando as implementações de Chu–Liu/Edmonds e Frank (em suas variantes com e sem heap) em grafos de diferentes tamanhos e densidades. Os objetivos são múltiplos: validar a corretude das implementações, caracterizar o comportamento temporal de cada método, e verificar se as previsões teóricas (como a dominância da Fase I em Frank) se confirmam na prática.

6.1 Testes e Resultados

Para validar a eficácia dos algoritmos de *Chu–Liu/Edmonds* e *Frank*, realizamos uma série de testes em diferentes instâncias de grafos direcionados. Os testes foram projetados para avaliar não apenas a correção dos algoritmos, mas também seu desempenho em termos de tempo de execução e uso de memória.

Metodologia

Geramos digrafos enraizados aleatórios com $|V| \in [100, 200]$ e $|A| \in [n, 3n]$, pesos inteiros uniformes em $[1, 20]$, e conectividade a partir da raiz $r0$ garantida por uma construção incremental. Em cada instância (arquivo `tests.py`):

- removemos arestas que entram em $r0$ (normalização compatível com as formulações);
- executamos *Chu–Liu/Edmonds* e as duas variantes da Fase II de *Frank* (versão direta e com heap) sobre o subgrafo de zeros produzido na Fase I;
- comparamos os custos retornados e verificamos a *condição dual* (complementaridade) para ambas as soluções de Frank;
- registramos resultado e tempo total por instância em CSV.

Resultados

Nas instâncias geradas, os três construtores de arborescência retornam sempre o mesmo custo, e as duas verificações da condição de complementaridade (dual) passam em 100% dos casos reportados no CSV. Isso corrobora a corretude das implementações e a equivalência entre *Chu–Liu/Edmonds* e *Frank* no valor ótimo (cf. Seções anteriores e [7, 21]).

Do ponto de vista de desempenho, a decomposição temporal por etapas evidencia três fatos principais:

- A Fase I (normalização primal–dual/elevação de potenciais) domina o tempo total para os tamanhos $|V| \in [100, 200]$ e $|A| \in [n, 3n]|$: tipicamente responde pela maior parcela do tempo por instância, ao passo que o tempo de *Chu–Liu/Edmonds* é menor e as Fases II são uma fração residual.
- Entre as duas variantes de Fase II, a versão com heap (v2) é sistematicamente mais rápida que a versão direta (v1). Observamos ganhos mediana/mediana na ordem de 3–8 $\times$ (histograma de speedup), compatíveis com a troca de uma varredura sequencial por extrações de menor prioridade em $O(\log n)$.
- As estatísticas estruturais do *Chu–Liu/Edmonds* mostram número de contrações e profundidade de recursão pequenos na maioria dos casos (tipicamente 0–3), com alguns outliers; isso é consistente com o limite $O(n)$ para o número de contrações [21]. O tamanho do subgrafo de zeros D_0 cresce aproximadamente linearmente com $|V|$, como esperado pelo critério de "uma entrada apertada por vértice". O pico de memória observado na Fase I ficou bem abaixo de 1 MB nas instâncias testadas.

As Figuras 55–60 resumem esses achados. O boxplot (Fig. 55) mostra a distribuição dos tempos de cada etapa; nota-se que a Fase I concentra a maior variabilidade e mediana mais alta. A nuvem tempo vs. $|A|$ (Fig. 56) indica crescimento aproximadamente linear nas faixas testadas. O histograma de speedup (Fig. 57) evidencia vantagem consistente da variante com heap na Fase II. Por fim, as distribuições de contrações/profundidade (Fig. 58) e de pico de memória (Fig. 59) são concentradas em valores baixos, enquanto o gráfico $|A(D_0)|$ vs. $|V|$ (Fig. 60) sugere proporcionalidade entre o tamanho de D_0 e o número de vértices.

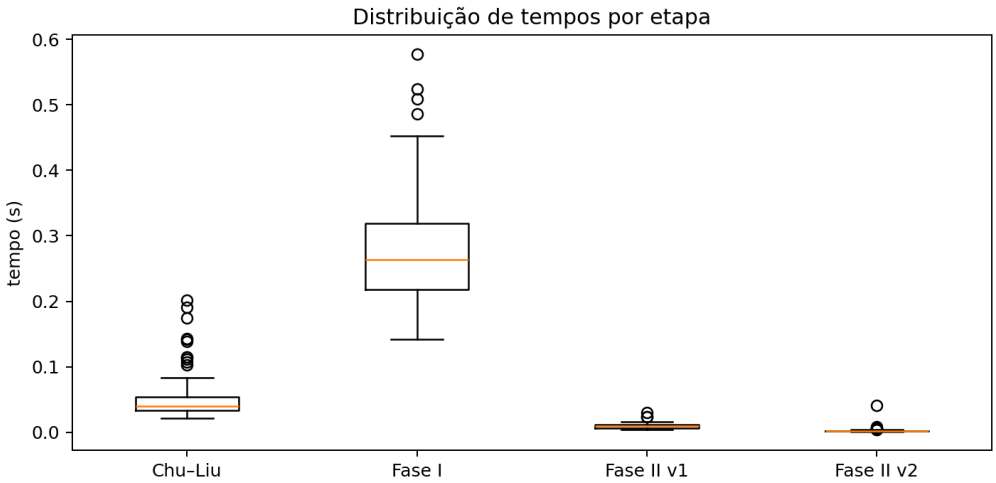


Figura 55: Distribuição de tempos por etapa (boxplot): *Chu–Liu*, Fase I, Fase II v1 (direta) e Fase II v2 (heap).

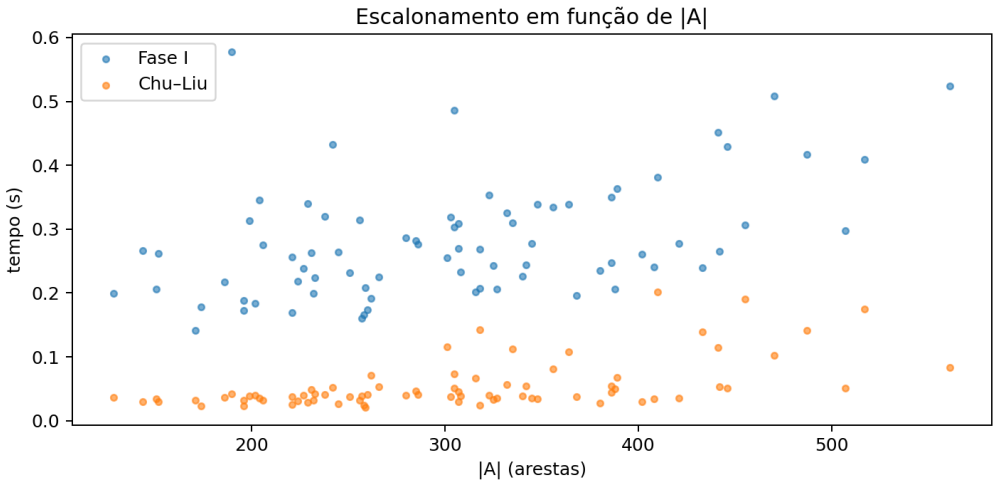


Figura 56: Escalonamento temporal em função de $|A|$: comparação entre *Chu–Liu* e Fase I.

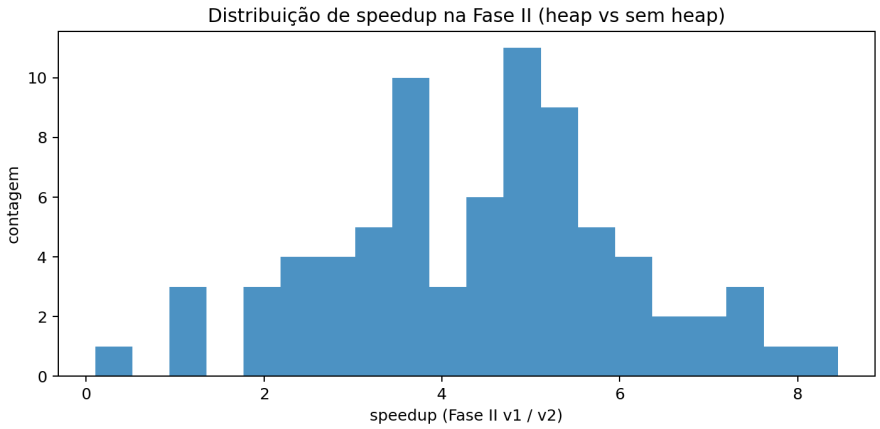


Figura 57: Histograma de speedup na Fase II (v1/v2): valores maiores que 1 indicam v2 (heap) mais rápida.

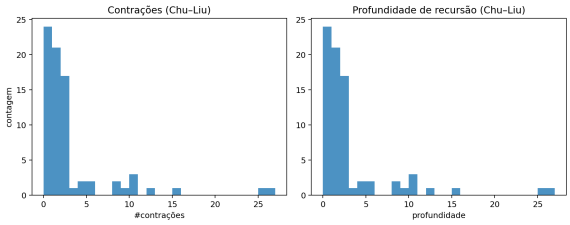


Figura 58: Distribuições do número de contrações e da profundidade de recursão em *Chu–Liu*.

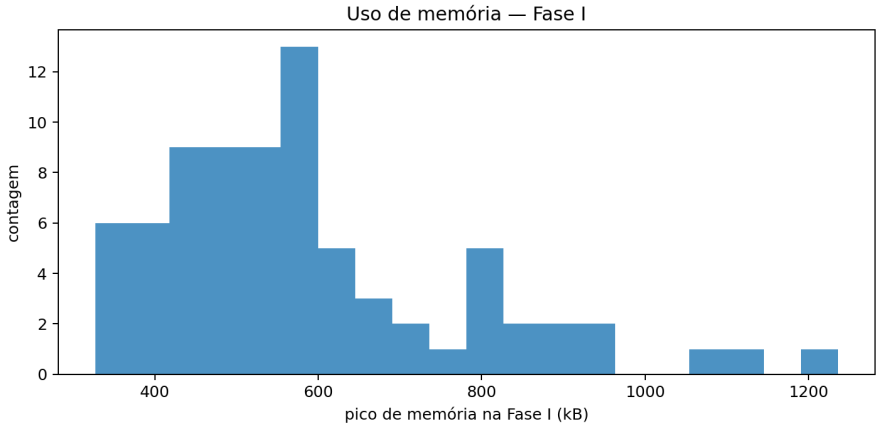


Figura 59: Pico de memória observado na Fase I (kB).

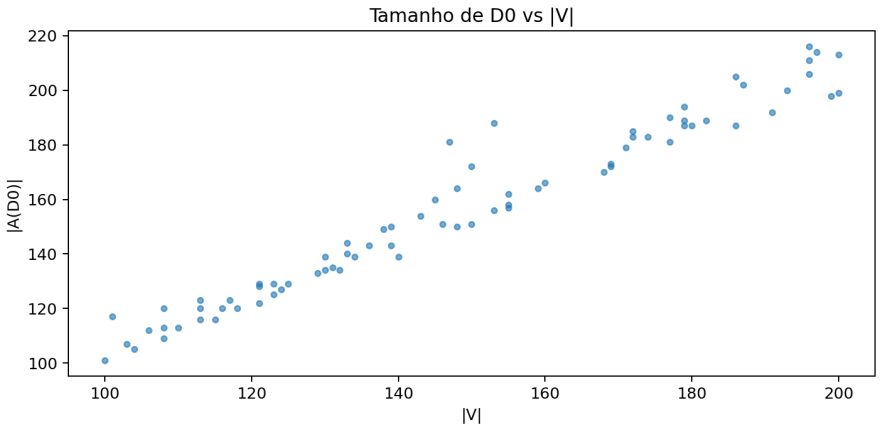


Figura 60: Tamanho de D_0 (número de arestas de custo reduzido zero) em função de $|V|$.

Em síntese, os resultados empíricos alinham-se às previsões teóricas: custos coincidem e satisfazem complementaridade; o *Chu–Liu* apresenta boa escalabilidade nas faixas testadas; a Fase I de Frank tende a dominar o tempo; e a variação com heap reduz significativamente o custo da Fase II, refletindo o uso de extrações $\log n$. Essas observações reforçam o quadro de [7, 21]: a organização primal–dual de Frank torna explícitas as estruturas (cortes ativos, apertude) que explicam tanto a corretude quanto caminhos de otimização.

Os testes e as análises apresentados oferecem uma base robusta para a compreensão prática e teórica dos algoritmos de arborescência de custo mínimo, explicitando suas forças e nuances. Compreender os algoritmos teoricamente e validá-los empiricamente, porém, é apenas parte do desafio: como transformar esse conhecimento em aprendizagem efetiva para estudantes e profissionais?

A resposta que propomos envolve tornar essa experiência concreta e interativa. Para isso, desenvolvemos uma aplicação *web* que permite acompanhar, passo a passo, o funcionamento de ambos os algoritmos, evidenciando suas semelhanças e diferenças de forma visual e manipulável.

A aprendizagem visual é especialmente útil nesse contexto: observar os algoritmos em ação, ver como normalizações transformam custos, como ciclos são contraídos e expandidos, e como potenciais duais guiam a seleção de arestas ajuda a fixar os conceitos apresentados de maneira que a leitura passiva não consegue. Antes de apresentar a aplicação em si, discutimos os fundamentos didáticos que orientaram seu design, explorando aspectos da didática de conceitos abstratos e princípios de interação humano-computador.

7 A Didática do Abstrato

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

Grafos e digrafos são simultaneamente concretos (nós e arestas) e abstratos (propriedades globais como cortes, conectividade, laminaridade). A multiplicidade de noções (caminhos, ciclos, cortes, componentes, condensação) [2, 5, 29]. Essas noções exigem transitar entre níveis de representação (intuitivo, visual, simbólico, formal) [26], o que pode ser desafiador. A abstração é poderosa, mas também pode ser uma barreira: conceitos como "corte ativo" ou "complementaridade primal–dual" são difíceis de visualizar e internalizar sem apoio didático adequado.

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

7.1 Fundamentos cognitivos e didáticos

O ensino de matemática no ensino superior exige transitar entre registros de representação (intuitivo, visual, simbólico, formal) com intencionalidade didática [26]. 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 [25]. 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.

7.1.1 Desafios centrais

Aprender conteúdos de alta abstração envolve lidar com sobrecarga cognitiva intrínseca e extrínseca [25]. Diretrizes de aprendizagem multimídia indicam que combinar representações verbais e visuais pode reduzir carga desnecessária e favorecer integração semântica [16, 19]. Em matemática avançada, a transição entre níveis de representação (intuitivo, formal, simbólico) exige mediação cuidadosa [26] e atenção a como exemplos, contraexemplos e invariantes são apresentados.

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

7.1.2 Lidando com grafos e digrafos

Na prática, o que mais dificulta o ensino de digrafos não é definir vértices e arcos, mas articular o que fazemos localmente com as estruturas globais que sustentam as provas de correção e de otimalidade em arborescências de custo mínimo — em particular, nos métodos de Chu–Liu/Edmonds e de Frank —: cortes ativos, componentes fortemente conexas (SCCs) e famílias laminares de conjuntos [2, 5, 29]. Quando essa articulação não aparece, cresce a *carga intrínseca* (múltiplas dependências simultâneas) e também a *carga extrínseca* (o esforço de alinhar texto, fórmulas e figuras). Nesse contexto específico destacamos três desafios didáticos:

- **Articular o local com o global:** escolher a melhor aresta de entrada para cada vértice não garante coerência global; isso pode criar ciclos. Ver o grafo *condensado* em SCCs (cada ciclo vira um "bloco") torna esse efeito visível e manipulável (Fig. 61). *Dificuldade típica:* estudantes tendem a projetar a heurística local para o todo e se surpreendem com ciclos "inesperados" — uma fonte comum de sobrecarga por conflito entre intuições locais e restrições globais.
- **Acompanhar efeitos de contração/expansão:** contrair um ciclo (substituílo por um supervértice) e depois reexpandir impacta os custos reduzidos c' e os cortes que ficam "ativos". A laminaridade — cortes aninhados, sem interseções conflitantes — fornece uma geometria simples para seguir essas mudanças (Fig. **??**). *Dificuldade típica:* perder o fio entre representações (grafo original, condensado, reexpansão) aumenta a carga extrínseca; sinalizar o que mudou em cada etapa reduz esse atrito.
- **Mapear o "fazer do algoritmo" para a linguagem primal–dual:** ações como elevar potenciais, escolher entradas e contrair ciclos correspondem a garantias teóricas. Em particular, *apertude* (custo reduzido zero) e *complementaridade* (exatamente uma aresta entra em cada conjunto ativo) certificam a otimalidade. Relacionar essas ideias às métricas que coletamos (tempos, número de contrações, pico de memória) ajuda a ligar prática e teoria, via custos reduzidos e reexpansão (Figs. **??** e **??**). *Dificuldade típica:* executar os passos sem ver como eles tornam certas restrições "justas" dificulta a internalização do porquê; explicitar os vínculos entre ação e certificação reduz a distância entre o operacional e o conceitual.

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

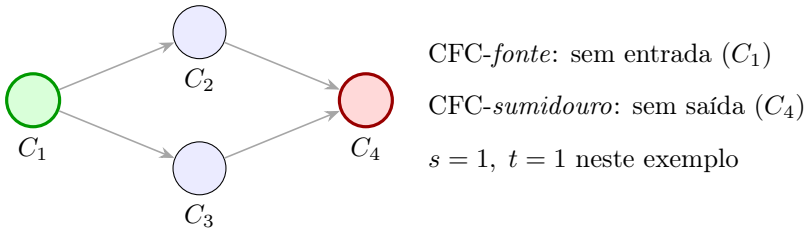


Figura 61: Condensação de D_0 e fontes do DAG: ver o grafo "como" blocos (SCCs) ajuda a articular o local (entradas por vértice) com o global (cortes e contrações).

7.2 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 [15, 28].

A teoria da carga cognitiva sugere que combinar representações verbais e visuais pode reduzir carga extrínseca e favorecer integração semântica [16, 19]. 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) [16].

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 [26]. Diagramas não são meros adornos: estruturam inferências espaciais e relacionais de modo mais eficiente que sentenças lineares [15].

Na educação em ciência da computação, visualizações de algoritmos têm efeito positivo sobretudo quando promovem *engajamento ativo* (prever, manipular, explicar) e não apenas consumo passivo [12, 17].

7.3 Disseminação de conteúdos avançados: o ecossistema de ferramentas

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

Tendo em vista esses fundamentos, ferramentas digitais podem ajudar a reduzir carga extrínseca e a integrar registros (visual, simbólico, formal) quando a interação é desenhada para promover *engajamento ativo* [16, 25, 12, 17]. A seguir, apresentamos categorias de ferramentas úteis no ensino de grafos, indicando finalidades, forças e limitações, e posicionamos a nossa aplicação nesse ecossistema.

7.3.1 Ferramentas didáticas no ensino de teoria dos grafos

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

Diagramas e matemática

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

Ambientes como Graphviz/dot e TikZ/PGF permitem especificar grafos declarativamente e gerar figuras reprodutíveis com layouts consistentes [9, 27]. Benefícios didáticos: (i) semântica visual estável (mesmo conceito, mesma forma), (ii) autoria próxima ao símbolo e ao texto (co-referência), (iii) manutenção e versionamento fáceis. Limitações: a interação costuma ser offline (figuras estáticas) e a curva de aprendizado de sintaxe pode ser um obstáculo inicial. Em contextos de prova e definição, esses recursos ancoram a narrativa formal com diagramas que obedecem às diretrizes de [15, 28].

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

Exploração e edição de grafos

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

Dentre elas destacam-se Gephi, yEd e Cytoscape, que oferecem layouts automáticos, filtros e medidas de rede [1, 30, 22]. São adequadas para: (i) reconhecer padrões estruturais (componentes, comunidades), (ii) discutir implicações de layouts para percepção de estruturas, (iii) atividades de descoberta assistida ("*overview* $\rightarrow$ *filter* $\rightarrow$ *details*") [23]. Limitações: (i) foco em análise exploratória de dados, não em algoritmos específicos; (ii) carga extrínseca ao alternar entre interface gráfica e conceitos teóricos; (iii) falta de controle fino sobre estados intermediários de algoritmos.

Visualização de algoritmos

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

Repositórios e portais como VisuAlgo e iniciativas similares apresentam animações de algoritmos clássicos com controle do ritmo e de estados [11, 12, 17]. Evidências sugerem ganhos quando o estudante prevê, manipula e explica o que vê, ao invés de consumir animações passivamente. Pontos de atenção: (i) alinhar a animação às noções teóricas subjacentes (invariantes, certificados), (ii) explicitar mapeamentos entre "o que acontece" e "o que se garante" (p.ex., custo reduzido zero $c' = 0$, complementaridade), (iii) evitar excesso de elementos visuais que aumentem carga extrínseca [16].

Ambientes programáveis e reprodutibilidade

Ferramentas como Jupyter Notebooks e bibliotecas como NetworkX permitem combinar código, texto e visualizações em um único documento interativo [14, 10]. Essas ferramentas são valiosas para criar exemplos reprodutíveis e explorar algoritmos de forma prática.

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

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

8 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, dual coding, engajamento ativo) e *como* estruturar o design (heurísticas de IHC, visão geral com detalhe sob demanda, feedback imediato).

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

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

8.1 Princípios de interação humano-computador

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

Sintetizando heurísticas de usabilidade e descoberta [18, 24] e quadros de interação e aprendizagem [20, 16, 25, 17], destacamos oito princípios orientadores: (i) usabilidade, (ii) eficiência cognitiva, (iii) feedback imediato, (iv) engajamento ativo, (v) visão geral com detalhe sob demanda (mantra *overview→filter→details* de [23]), (vi) consistência semântica, (vii) múltiplos registros de representação e (viii) prevenção/recuperação de erros. A seguir descrevemos cada um e sua materialização na ferramenta.

Usabilidade

Usabilidade 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.

Eficiência cognitiva

Eficiência cognitiva 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.

Feedback Imediato

Feedback imediato é crucial para manter 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.

Engajamento ativo

Engajamento ativo 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.

Visão geral com detalhe sob demanda

A visão geral com detalhe sob demanda 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.

Consistência semântica

A consistência semântica 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.

Múltiplos registros de representação

Múltiplos registros de representação referem-se à capacidade de apresentar informações de diferentes maneiras, atendendo às preferências e estilos de aprendizagem dos usuários. Nossa aplicação oferece várias representações do grafo (visual, textual, interativa), permitindo que os usuários escolham a forma que melhor se adapta às suas necessidades.

Prevenção de erros

A prevenção de erros envolve projetar o sistema de forma a minimizar a probabilidade de erros dos usuários. Implementamos medidas de prevenção de erros, como validação de entrada e 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 1: Síntese dos princípios de interação humano-computador aplicados e sua realização concreta na ferramenta interativa.

8.2 Descrição da aplicação

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

8.2.1 Visão geral das páginas

A aplicação *web* é composta por páginas HTML autônomas carregadas no navegador, cada uma focada em um aspecto: (i) `home.html` (apresentação / resumo), (ii) `chuliu.html` (execução passo a passo do algoritmo de Chu–Liu/Edmonds), (iii) `andrasfrank.html` (estrutura para futura visualização da abordagem primal–dual), (iv) `draw_graph.html` (editor

livre de grafos), além do componente compartilhado de navegação lateral `sidebar.html`. Cada página injeta o `sidebar` dinamicamente e carrega scripts específicos.

Bibliotecas e dependências

O desenvolvimento utilizou tecnologias *web* modernas, incluindo HTML5, Tailwind CSS para estilos responsivos, e PyScript para execução de código Python diretamente no navegador. A biblioteca NetworkX foi empregada para manipulação de grafos, enquanto Matplotlib gerou visualizações estáticas dos estados intermediários. O código foi estruturado para modularidade e clareza, facilitando futuras extensões. Detalhamos abaixo esses aspectos:

- **Frontend:** HTML5 + Tailwind CSS para composição responsiva e estilos utilitários consistentes.
- **Execução Python no cliente:** PyScript (carregado via CDN) provê execução de módulos Python (ex.: `networkx`, `matplotlib`) sem instalação local, reduzindo barreira de entrada didática.
- **Bibliotecas centrais:** NetworkX para manipulação de grafos dirigidos; `matplotlib` para renderização de snapshots; (em páginas avançadas) Cytoscape.js previsto para interação rica (estrutura já importada em `chuliu.html` / `andrasfrank.html`).
- **Empacotamento de estado:** Serialização JSON (formato `node_link`) para exportação e reuso analítico.
- **Isolamento:** Cada página carrega apenas os elementos necessários, evitando *payload* excessivo e minimizando latência de interface (princípio de eficiência cognitiva).

Componentes funcionais principais

A aplicação centraliza-se nas páginas `chuliu.html` e `andrasfrank.html`, que permitem executar os algoritmos de Chu–Liu/Edmonds e Frank, respectivamente. A seguir, detalhamos os componentes funcionais principais da interface:

1. **Editor de Grafo:** inputs para origem, destino e peso + botões de inserção e limpeza (`add-edge`, `reset-graph`).
2. **Carregamento de exemplo:** botão `load-test-graph` injeta um grafo de teste estruturado para reduzir *time-to-first-insight*.
3. **Configuração de raiz:** campo para definir / confirmar o vértice raiz (`root-node`).
4. **Execução algorítmica:** botão `run-algorithm` dispara o pipeline de filtragem e cálculo da arborescência mínima.
5. **Visualização de estados:** área cumulativa (`graph-area`) onde cada renderização mantém a coerência posicional (layout determinístico) para comparação mental entre passos.
6. **Log textual:** caixa de texto incremental (`log-output`) correlaciona ações a transformações estruturais (dual coding: texto + imagem).

7. **Exportação:** ação de exportar grafo em JSON para análise posterior ou reimportação.
8. **Incorporação de PDF:** `tese.html` exibe o `main.pdf` lado a lado à aplicação, incentivando leitura entrelaçada teoria-execução.

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. A seguir, descrevemos o fluxo típico:

1. Usuário monta ou carrega um grafo de teste.
2. Define (ou confirma) o vértice raiz r_0.
3. Executa o algoritmo (Chu–Liu/Edmonds): são aplicadas normalizações e seleção de arestas (implementação apresentada em seções anteriores / listagens).
4. Observa estados sequenciais: cada snapshot reforça invariantes (arestas escolhidas, pesos, estrutura alcançada).
5. (Opcional) Exporta o grafo resultante para replicação em notebooks ou comparação com a abordagem dual futura.

Estado e dados persistentes

O estado principal é mantido em memória (objeto `networkx.DiGraph`). 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.

Limitações atuais

Atualmente, a aplicação apresenta algumas limitações que podem impactar a experiência do usuário e a eficácia da visualização:

- Ausência de visualização explícita de contração de ciclos (marcação diferenciada por cores/aglomerados ainda não implementada).
- Não há comparação lado a lado (split view) entre algoritmos (Chu–Liu vs. Frank) ainda.
- Layout planar simples pode falhar em instâncias mais densas (sobreposição de rótulos) — futura substituição por `spring` ou `dagre`-like adaptativo.
- Falta camada de destaque cromático para custos reduzidos e arcos "apertados" ($c' = 0$).
- Exportação limita-se ao grafo final; não há pacote de estados intermediários (*replay*).

Melhorias futuras

Desse modo, entendemos que a aplicação, embora funcional, pode ser aprimorada com recursos adicionais para enriquecer a experiência didática. Algumas melhorias incluem:

- Visualização animada da contração/reexpansão de ciclos com agrupamento colapsável.
- Camada de coloração para arestas com custo reduzido zero e cortes ativos.
- Geração automática de relatório (log + estados selecionados) em PDF/ZIP.
- Módulo paralelo para a abordagem primal–dual de Frank (empacotamento de cortes e duas fases).
- Modo "comparativo" exibindo diferenças de passos e métricas agregadas.
- Instrumentação de métricas (tempo por passo, número de contrações, distribuição de pesos normalizados).

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

8.3 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.

Estrutura de arquivos

A estrutura de arquivos da aplicação é organizada da seguinte forma:

- `main.py`: script Python principal contendo funções de manipulação de grafos e lógica algorítmica.
- `index.html`: página inicial com resumo e links para outras seções.
- `chuliu.html`: página dedicada à execução do algoritmo de Chu–Liu/Edmonds.
- `andrasfrank.html`: página para futura implementação do algoritmo de Frank.
- `draw_graph.html`: editor livre de grafos.
- `sidebar.html`: componente de navegação lateral compartilhado.
- `styles.css`: arquivo de estilos customizados (além do Tailwind).
- `scripts/`: diretório contendo scripts Python e JavaScript.
- `assets/`: diretório para imagens, ícones e outros recursos estáticos.

8.3.1 Códigos principais

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

`Main.py:` o arquivo `main.py` contém funções para manipulação de grafos, execução dos algoritmos e geração de visualizações.

As funções principais incluem:

- `draw_graph(G, title, append)`: renderiza o grafo G com título e opção de anexar ou substituir a visualização.
- `add_edge()`: adiciona uma aresta ao grafo com base nos inputs do usuário.
- `reset_graph()`: limpa o grafo atual e o log.
- `export_graph()`: exporta o grafo atual em formato JSON.
- `load_test_graph()`: carrega um grafo de teste predefinido.
- `run_algorithm()`: executa o algoritmo de Chu–Liu/Edmonds, aplicando filtragem e visualizando cada passo.

A seguir, apresentamos essas funções conforme implementadas no arquivo `main.py`.

Função 8.1: Main.py: manipulação e visualização de grafos

```
import networkx as nx
from networkx.readwrite import json_graph
import matplotlib.pyplot as plt
from js import Blob, URL, document, alert
from pyscript import when, display
import json

from chuliu import find_optimum_arborescence_chuliu,
    remove_edges_to_r0

def log_in_box(msg: str):
    log_box = document.getElementById("log-output")
    log_box.value += msg + "\n"
    log_box.scrollTop = log_box.scrollHeight

def draw_graph(G: nx.DiGraph, title="Digrafo", append=True):
    plt.clf()  # Limpa a figura atual
    pos = nx.planar_layout(G)  # Layout para posicionamento
        dos nós
    plt.figure(figsize=(6, 4))  # Tamanho da figura
    # Desenha os nós e arestas
    nx.draw(
        G,
        pos,
        with_labels=True,
        node_color="lightblue",
        edge_color="gray",
        node_size=2000,
        font_size=12,
    )
```

```
29      weights = nx.get_edge_attributes(G, "w")
30      nx.draw_networkx_edge_labels(
31          G, pos, edge_labels=weights, font_color="red",
               font_size=12
32      )
33      plt.title(title)
34      display(title, target="graph-area", append=append)
35      display(plt, target="graph-area", append=append)
36      plt.close()  # Fecha a figura para liberar memória
37
38  G = nx.DiGraph()
39
40  @when("click", "#add-edge")
41  def add_edge():
42      global G
43      source = document.getElementById("source").value
44      target = document.getElementById("target").value
45      weight = document.getElementById("weight").value
46      if source and target and weight:
47          G.add_edge(source, target, w=float(weight))
48          log_in_box(f"Aresta adicionada: {source} → {target} (
               peso={weight})")
49          draw_graph(G, "Grafo com Arestas", append=False)
50
51  @when("click", "#reset-graph")
52  def reset_graph():
53      global G
54      G.clear()
55      document.getElementById("log-output").value = ""
56      draw_graph(G, "Grafo Resetado", append=False)
57      log_in_box("Grafo resetado.")
58
59  @when("click", "#export-graph")
60  def export_graph(event):
61      log_in_box("Exportando grafo...")
62      global G
63      if G.number_of_nodes() == 0:
64          log_in_box("[ERRO] O grafo está vazio.")
65          return
66
67      # Converte o grafo para JSON
68      data = json_graph.node_link_data(G, edges="links")
69      json_data = json.dumps(data, indent=4)
70
71      # Cria um link de download no navegador
72      blob = Blob.new([json_data], {"type": "application/json"
           })
73      url = URL.createObjectURL(blob)
74
```

```
75      # Configura e dispara o download
76      link = document.createElement("a")
77      link.href = url
78      link.download = "graph_teste.json"
79      link.click()
80      URL.revokeObjectURL(url)
81
82      log_in_box("Download do grafo iniciado.")
83
84  @when("click", "#load-test-graph")
85  def load_test_graph(event):
86      global G
87      G.clear()
88      G.add_edge("r0", "B", w=10)
89      G.add_edge("r0", "A", w=2)
90      G.add_edge("r0", "C", w=10)
91      G.add_edge("B", "A", w=1)
92      G.add_edge("A", "C", w=4)
93      G.add_edge("C", "D", w=2)
94      G.add_edge("D", "B", w=2)
95      G.add_edge("B", "E", w=8)
96      G.add_edge("C", "E", w=4)
97
98      log_in_box("Grafo de teste carregado.")
99      draw_graph(G, "Grafo de Teste (DG)", append=False)
100
101 @when("click", "#show-ready-arborescence")
102 def show_ready_arborescence(event):
103     T = nx.DiGraph()
104     T.add_edge("r0", "A", w=2)
105     T.add_edge("A", "C", w=4)
106     T.add_edge("C", "D", w=2)
107     T.add_edge("D", "B", w=2)
108     T.add_edge("C", "E", w=4)
109     draw_graph(T, "Arborescência Pré-definida")
110     log_in_box("Arborescência pronta exibida.")
111
112
113 @when("click", "#run-algorithm")
114 def run_algorithm(event):
115     global G
116     r0 = document.getElementById("root-node").value or "r0"
117     if r0 not in G:
118         alert(f"[ERRO] O nó raiz '{r0}' deve existir no grafo
            .")
119         return
120
121     log_in_box("Executando algoritmo de Chu-Liu...")
122     print(remove_edges_to_r0)
```

```
    G_filtered = remove_edges_to_r0(G, r0)
    T = find_optimum_arborescence_chuliu(
        G_filtered, r0, draw_fn=draw_graph, log=log_in_box
    )
    draw_graph(T, "Arborescência Ótima")
    log_in_box("Execução concluída com sucesso.")
```

`Index.html:` o arquivo `index.html` define a estrutura principal da página HTML, incluindo a integração com PyScript e Tailwind CSS para estilos responsivos. A seguir, apresentamos o código completo dessa página.

Markup 8.1: Estrutura principal da página HTML com PyScript

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width,initial
       -scale=1" />
    <title>Chu-Liu/Edmonds Visualizer</title>
    <script src="https://cdn.tailwindcss.com"></script>
    <link
      rel="stylesheet"
      href="https://pyscript.net/releases/2025.3.1/core.css"
    />
    <script
      type="module"
      src="https://pyscript.net/releases/2025.3.1/core.js"
    ></script>
  </head>
  <body class="bg-gray-100">
    <header class="bg-[#5b8f79] text-white text-center p-6">
      <h1 class="text-3xl font-bold">Chu-Liu/Edmonds
         Visualizer</h1>
      <p class="text-lg mt-2">
        Visualização passo a passo de arborescências em
           grafos direcionados
      </p>
    </header>

    <main class="flex flex-col items-center gap-8 py-8 px-8">
      <section class="text-center w-full max-w-4xl">
        <h2 class="text-2xl font-semibold text-[#5b8f79] mb-4
           ">
          Definir Di-grafo
        </h2>
```

```
30          <div class="flex flex-wrap justify-center gap-4 items
               -center mb-4">
31            <input
32              type="text"
33              id="source"
34              placeholder="Origem"
35              class="p-3 border rounded border-gray-300 w-40"
36            />
37            <input
38              type="text"
39              id="target"
40              placeholder="Destino"
41              class="p-3 border rounded border-gray-300 w-40"
42            />
43            <input
44              type="number"
45              id="weight"
46              placeholder="Peso"
47              min="0"
48              class="p-3 border rounded border-gray-300 w-40"
49            />
50            <button
51              id="add-edge"
52              class="p-3 bg-[#5b8f79] text-white font-bold
                   rounded hover:bg-[#4a7a66]"
53            >
54                          Adicionar Aresta
55            </button>
56          </div>
57          <div class="flex flex-wrap justify-center gap-4 items
               -center mb-4">
58            <button
59              id="reset-graph"
60              class="p-3 bg-[#5b8f79] text-white font-bold
                   rounded hover:bg-[#4a7a66]"
61            >
62                          Limpar Grafo
63            </button>
64            <button
65              id="export-graph"
66              class="p-3 bg-[#5b8f79] text-white font-bold
                   rounded hover:bg-[#4a7a66]"
67            >
68                          Exportar Grafo para JSON
69            </button>
70          </div>
71          <div class="flex flex-wrap justify-center gap-4 items
               -center mb-4">
72            <button
```

```
73              id="load-test-graph"
74              class="p-3 bg-[#5b8f79] text-white font-bold
                   rounded hover:bg-[#4a7a66]"
75            >
76                          Carregar Grafo de Teste
77            </button>
78            <button
79              id="show-ready-arborescence"
80              class="p-3 bg-[#5b8f79] text-white font-bold
                   rounded hover:bg-[#4a7a66]"
81            >
82                          Mostrar Arborescência do Grafo de
                               Teste
83            </button>
84          </div>
85        </section>
86
87        <section class="text-center w-full max-w-4xl">
88          <h2 class="text-2xl font-semibold text-[#5b8f79] mb-4
                ">
89            Executar Algoritmo de Chu-Liu/Edmonds
90          </h2>
91          <div class="flex flex-wrap justify-center gap-4 items
                -center">
92            <label for="root-node" class="text-gray-700 font-
                   medium"
93              >Nó Raiz:</label
94            >
95            <input
96              type="text"
97              id="root-node"
98              placeholder="Nó Raiz"
99              value="r0"
100             class="p-3 border rounded border-gray-300 w-40"
101           />
102           <button
103             id="run-algorithm"
104             class="p-3 bg-[#5b8f79] text-white font-bold
                   rounded hover:bg-[#4a7a66]"
105           >
106                         Executar Algoritmo
107           </button>
108         </div>
109       </section>
110
111       <section
112         id="graph-output"
113         class="w-full max-w-4xl bg-white p-6 rounded shadow"
114       >
```

```
115         <h2 class="text-2xl font-semibold text-[#5b8f79] mb-4
               ">Grafo Atual</h2>
116         <py-script id="graph-area"></py-script>
117       </section>
118
119       <section id="log" class="w-full max-w-4xl bg-white p-6
             rounded shadow">
120         <h2 class="text-2xl font-semibold text-[#5b8f79] mb-4
               ">
121           Log da Execução
122         </h2>
123         <textarea
124           id="log-output"
125           readonly
126           class="w-full p-3 border rounded border-gray-300 h-
                 40"
127         ></textarea>
128       </section>
129     </main>
130
131     <script type="py" src="./main.py" config="./pyscript.json
           "></script>
132   </body>
133 </html>
```

8.3.2 Demais Páginas da Aplicação *web*

Além da página principal do visualizador, a aplicação inclui um conjunto de páginas modulares que suportam a navegação e fornecem contexto adicional. A seguir, detalhamos a estrutura e função de cada uma dessas páginas.

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

Figura 62: Captura de tela de `home.html`: visão geral com resumo e integrantes.

A seguir, apresentamos o código completo dessa página.

Markup 8.2: home.html

```
<!DOCTYPE html>
<html lang="pt-BR">

<head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width,
       initial-scale=1" />
    <title>Arbograph</title>
    <link rel="icon" type="image/x-icon" href="../assets/logo
       .png"/>
    <script src="https://cdn.tailwindcss.com"></script>
    <link rel="stylesheet" href="https://pyscript.net/
       releases/2025.3.1/core.css" />
    <script type="module" src="https://pyscript.net/releases
       /2025.3.1/core.js"></script>
</head>

<body class="flex min-h-screen text-gray-800 bg-gray-100">
    <style>
        .py-error,
        .py-terminal-error {
            display: none !important;
        }
    </style>
```

```
21      <div class="absolute top-[100px] w-full h-[1px] bg-[#
           DBDBDB]"></div>
22
23      <!-- MENU LATERAL -->
24      <div id="sidebar"></div>
25
26      <!-- CONTEÚDO PRINCIPAL -->
27      <div id="main-content" class="flex-1 bg-[#E5E5E5] p-6">
28          <!-- Análise e Implementação de Algoritmo de Busca de
                uma r-Arborescência Inversa de Custo Mínimo em
               Grafos Dirigidos -->
29          <span class="flex items-center gap-4 mb-10 text-2xl
               text-[#352B67]">Análise e Implementação de
               Algoritmo de Busca de uma r-Arborescência Inversa
               de Custo Mínimo em Grafos Dirigidos</span>
30          <section class="mb-10 bg-white p-8 rounded-lg shadow-
               lg" id="resumo">
31              <scan class="text-3xl font-semibold text-center
                   mb-8 text-[#352B67] text-xl">Resumo</scan>
32              <div class="prose max-w-none text-gray-700">
33                  <scan class="text-lg leading-relaxed mb-4
                       text-sm font-light">
34                      At vero eos et accusamus et iusto odio
                           dignissimos ducimus qui
35                      blanditiis praesentium voluptatum
                           deleniti atque corrupti quos
36                      dolores et quas molestias excepturi sint
                           occaecati cupiditate non
37                      provident, similique sunt in culpa qui
                           officia deserunt mollitia a
38                      nimi, id est laborum et dolorum fuga. Et
                           harum quidem rerum facilis
39                      est et expedita distinctio. Nam libero
                           tempore, cum soluta nobis est eligendi
40                      optio cumque nihil impedit quo minus id
                           quod maxime placeat facere possimus,
                           omnis voluptas
41                      assumenda est, omnis dolor repellendus.
                           Temporibus autem quibusdam et aut
                           officiis debitis aut
42                      rerum necessitatibus saepe eveniet ut et
                           voluptates repudiandae sint et
                           molestiae non recusandae.
43                      Itaque earum rerum hic tenetur a sapiente
                            delectus, ut aut reiciendis
                           voluptatibus maiores alias .
44                  </scan>
45              </div>
46          </section>
```

```
47          <section class="bg-white p-8 rounded-lg shadow-lg" id
               ="integrantes">
48              <h2 class="text-3xl font-light text-center mb-8
                   text-[#352B67] text-xl">Integrantes do Projeto
                   </h2>
49
50              <div class="grid grid-cols-1 md:grid-cols-3 lg:
                   grid-cols-3 gap-4 justify-items-center">
51                  <div class="bg-gray-100 p-6 rounded-lg shadow
                       -md text-center flex flex-col items-center
                        w-full max-w-sm">
52                      <img src="../assets/lori.jpg" alt="Foto
                           da Lori"
53                          class="w-32 h-32 rounded-full object-
                               cover mb-4 border-2 border-[#8
                               d79e5]">
54                      <h3 class="text-xl font-semibold text-[#4
                           f4678]">Lori Sampaio</h3>
55                      <p class="text-gray-600 mt-1">Discente do
                            BCC-UFABC</p>
56                      <p class="text-sm text-gray-500 mt-2">
57                          <a href="https://linkedin.com/in/
                               integrante1" target="_blank"
58                              class="text-blue-500 hover:
                                   underline">LinkedIn</a> |
59                          <a href="mailto:lorenypsum@gmail.com"
                              class="text-blue-500 hover:
                               underline">Email</a>
60                      </p>
61                  </div>
62                  <div class="bg-gray-100 p-6 rounded-lg shadow
                       -md text-center flex flex-col items-center
                        w-full max-w-sm">
63                      <img src="https://www.ufabc.edu.br/images
                           /conteudo/img-padrao-autor.png" alt="
                           Foto do Mario"
64                          class="w-32 h-32 rounded-full object-
                               cover mb-4 border-2 border-[#8
                               d79e5]">
65                      <h3 class="text-xl font-semibold text-[#4
                           f4678]">Mario Leston Rey</h3>
66                      <p class="text-gray-600 mt-1">Docente do
                           BCC-UFABC</p>
67                      <p class="text-sm text-gray-500 mt-2">
68                          <a href="#" target="_blank"
69                              class="text-blue-500 hover:
                                   underline">LinkedIn</a> |
70                          <a href="mailto:mario.leston@ufabc.
                               edu.br" class="text-blue-500 hover
```

```
                                :underline">Email</a>
71                      </p>
72                  </div>
73                  <div class="bg-gray-100 p-6 rounded-lg shadow
                       -md text-center flex flex-col items-center
                        w-full max-w-sm">
74                      <img src="https://media.licdn.com/dms/
                           image/v2/C4D03AQHr9O4LzxUomQ/profile-
                           displayphoto-shrink_400_400/profile-
                           displayphoto-shrink_400_400
                           /0/1646342924520?e=1753920000&v=beta&t
                           =8
                           fpR8VQiVYX_2xWaQwUXK6NqxKbHVYvGKH1ProxHogg
                           " alt="Foto da Samira"
75                          class="w-32 h-32 rounded-full object-
                               cover mb-4 border-2 border-[#8
                               d79e5]">
76                      <h3 class="text-xl font-semibold text-[#4
                           f4678]">Samira Haddad</h3>
77                      <p class="text-gray-600 mt-1">Discente do
                            BCC-UFABC</p>
78                      <p class="text-sm text-gray-500 mt-2">
79                          <a href="https://www.linkedin.com/in/
                               samirahad" target="_blank"
80                              class="text-blue-500 hover:
                                   underline">LinkedIn</a> |
81                          <a href="mailto:samira.had.2000@gmail
                               .com" class="text-blue-500 hover:
                               underline">Email</a>
82                      </p>
83                  </div>
84
85
86              </div>
87          </section>
88      </div>
89
90      <!-- Scripts -->
91      <script src="../scripts/js/sidebar.js"></script>
92      <script src="../scripts/js/main.js"></script>
93  </body>
94  </html>
```

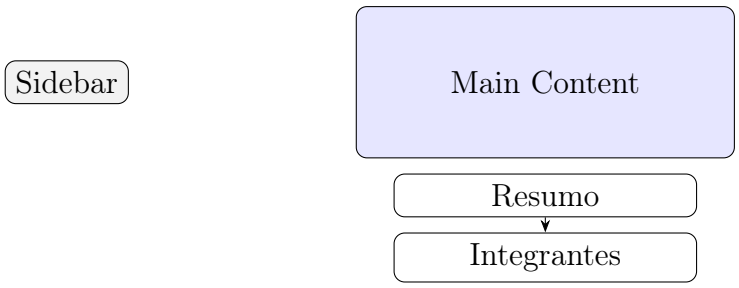


Figura 63: `home.html`: navegação lateral + conteúdo hierárquico (contexto →resumo →equipe).

Comentário: estrutura linear vertical facilita *progressive disclosure*; separação lateral reduz interferência visual com conteúdo principal.

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

Figura 64: Captura de tela de `draw_graph.html`: editor livre de grafos.

A seguir, apresentamos o código completo dessa página.

Markup 8.3: draw_graph.html

```
1 <!DOCTYPE html>
2 <html lang="pt-BR">
3
4 <head>
5     <meta charset="UTF-8" />
```

```
6       <meta name="viewport" content="width=device-width,
            initial-scale=1" />
7       <title>Desenhe seu grafo</title>
8       <link rel="icon" type="image/x-icon" href="../assets/logo
            .png"/>
9       <script src="https://cdn.tailwindcss.com"></script>
10      <script src="https://unpkg.com/cytoscape@3.26.0/dist/
            cytoscape.min.js"></script>
11
12      <link href="https://cdnjs.cloudflare.com/ajax/libs/
            flowbite/2.2.1/flowbite.min.css" rel="stylesheet" />
13      <script src="https://cdnjs.cloudflare.com/ajax/libs/
            flowbite/2.2.1/flowbite.min.js"></script>
14
15      <link rel="stylesheet" href="https://pyscript.net/
            releases/2025.3.1/core.css" />
16      <script type="module" src="https://pyscript.net/releases
            /2025.3.1/core.js"></script>
17  </head>
18
19  <body class="flex min-h-screen text-gray-800 bg-gray-100">
20      <style>
21          .py-error, .py-terminal-error {
22              display: none !important;
23          }
24      </style>
25      <div class="absolute top-[100px] w-full h-[1px] bg-[#
            DBDBDB]"></div>
26
27      <!-- MENU LATERAL -->
28      <div id="sidebar"></div>
29
30      <!-- CONTEÚDO PRINCIPAL -->
31      <div id="main-content" class="flex-1 bg-[#E5E5E5] p-6">
32          <span class="flex items-center gap-4 mb-10 text-4xl
                text-[#352B67]">Desenhe seu grafo</span>
33          <!-- Inputs -->
34          <div class="w-full flex justify-center">
35              <div class="grid grid-cols-10 gap-4 w-full mx-
                    auto">
36                  <div class="col-span-2 p-4"></div>
37                  <!-- Coluna 1 -->
38                  <div class="col-span-3 p-4">
39                      <span class="text-sm text-gray-500">
40                          1. Desenhe um grafo,
41                          <span id="load-test-graph" class="
                                cursor-pointer text-[#5A3CE5]
                                hover:underline">carregue</span>
42                          um exemplo ou
```

```
43                          <span id="import-graph" class="cursor
                               -pointer text-[#5A3CE5] hover:
                               underline">importe</span>
44                          <input type="file" id="file-input"
                               style="display: none;" />
45                          um grafo já existente.
46                      </span>
47                  </div>
48              </div>
49          </div>
50
51
52          <!-- Grafo original -->
53          <div class="grid grid-cols-10 gap-4 py-4 px-4">
54              <div class="col-span-2 p-4"></div>
55              <div class="col-span-6 p-4">
56                  <span class="mb-50 text-xl text-[#9993B7]">
                       Grafo Original</span>
57                  <div id="graph-editor" class="my-3 mt-50 py-4
                        px-50 rounded-lg bg-white h-[500px]">
58
59                  </div>
60              </div>
61              <div class="col-span-2 p-4">
62                  <div class="py-4 px-2 flex justify-left items
                       -center gap-1">
63                      <img src="../assets/download.png" alt="
                           Download" id="export-graph-original"
64                           class="my-5 cursor-pointer w-8 h-8
                               rounded-lg hover:opacity-80" />
65                      <img src="../assets/delete.png" alt="
                           Reset" id="reset-graph"
66                           class="my-5 cursor-pointer w-8 h-8
                               rounded-lg hover:opacity-80" />
67                  </div>
68              </div>
69
70          </div>
71      </div>
72
73      <div id="toast-danger" class="hidden fixed top-5 right-5
           z-50 flex items-center w-full max-w-xs p-4 mb-4 text-
           gray-500 bg-white rounded-lg shadow-sm dark:text-gray-
           400 dark:bg-gray-800 transition-opacity duration-500
           opacity-100" role="alert">
74          <div class="inline-flex items-center justify-center
               shrink-0 w-8 h-8 text-red-500 bg-red-100 rounded-
               lg dark:bg-red-800 dark:text-red-200">
75              <!-- SVG de erro -->
```

```
76              <svg class="w-5 h-5" aria-hidden="true" fill="
                   currentColor" viewBox="0 0 20 20">
77                  <path d="M10 .5a9.5 9.5 0 1 0 9.5 9.5A9.51
                       9.51 0 0 0 10 .5Zm3.707 11.793a1 1 0 1 1-
                       1.414 1.414L10 11.414l-2.293 2.293a1 1 0 0
                        1-1.414-1.414L8.586 10 6.293 7.707a1 1 0
                       0 1 1.414-1.414L10 8.586l2.293-2.293a1 1 0
                        0 1 1.414 1.414L11.414 10l2.293 2.293Z"/>
78              </svg>
79              <span class="sr-only">Error icon</span>
80          </div>
81          <div id="toast-danger-msg" class="ms-3 text-sm font-
               normal">Ocorreu um erro.</div>
82          <button type="button" class="ms-auto -mx-1.5 -my-1.5
               bg-white text-gray-400 hover:text-gray-900 rounded
               -lg focus:ring-2 focus:ring-gray-300 p-1.5 hover:
               bg-gray-100 inline-flex items-center justify-
               center h-8 w-8 dark:text-gray-500 dark:hover:text-
               white dark:bg-gray-800 dark:hover:bg-gray-700"
               data-dismiss-target="#toast-danger" aria-label="
               Close" onclick="document.getElementById('toast-
               danger').classList.add('hidden')">
83              <span class="sr-only">Close</span>
84              <svg class="w-3 h-3" aria-hidden="true" fill="
                   none" viewBox="0 0 14 14">
85                  <path stroke="currentColor" stroke-linecap="
                       round" stroke-linejoin="round" stroke-
                       width="2" d="m1 1 6 6m0 0 6 6M7 7l6-6M7 7l
                       -6 6"/>
86              </svg>
87          </button>
88      </div>
89
90      <!-- Scripts -->
91      <script src="../scripts/js/sidebar.js"></script>
92      <script src="../scripts/js/main.js"></script>
93      <script src="../scripts/js/draw_graph.js"></script>
94      <script type="py" src="../scripts/draw_page.py" config="
           ../scripts/pyscript.json"></script>
95  </body>
96  </html>
```

A figura a seguir destaca o foco primordial no componente editor de grafos.

Figura 65: `draw_graph.html`: foco primordial no componente editor.

Comentário: a estrutura modular permite fácil adaptação e reutilização de componentes.

`Sidebar.html:` componente de navegação lateral consistente em todas as páginas. Utiliza Tailwind CSS para estilo e inclui links para as principais seções do site, reforçando um modelo mental estável de navegação. A seguir, apresentamos o código completo desse componente.

Markup 8.4: sidebar.html

```
<script src="https://cdn.tailwindcss.com"></script>
<!-- <aside class="w-64 bg-[#E5E5E5] text-white p-6 flex flex-col justify-between h-full"> -->
<aside class="w-64 bg-[#E5E5E5] text-white p-6 flex flex-col justify-between h-full border-r-[1px] border-[#DBDBDB]">
    <div>
        <a href="home.html" class="flex items-center gap-4 mb-6 text-white font-bold text-xl">
            <img src="../assets/logo.png" alt="ArboGraph" class="w-16 h-auto" />
            <span class="text-[#352B67]">ArboGraph</span>
        </a>
        <!-- <div class="border-t-2 border-[#DBDBDB] w-full my-4"></div> -->
        <nav class="flex flex-col gap-3 text-left text-white font-medium">
            <a href="home.html" class="hover:bg-[#e0dede] p-2 rounded flex items-center gap-2">
                <img src="../assets/home.png" alt="Home" class="w-6 h-6" />
                <span class="text-[#787486]">Home</span>
            </a>
            <a href="chuliu.html" class="hover:bg-[#e0dede] p-2 rounded flex items-center gap-2">
                <img src="../assets/graph.png" alt="Chu-Liu" class="w-6" />
                <span class="text-[#787486]">Chu-Liu/Edmonds</span>
            </a>
```

```
19              <a href="andrasfrank.html" class="hover:bg-[#
                    e0dede] p-2 rounded flex items-center gap-2">
20                  <img src="../assets/graph.png" alt="Fulkerson
                        " class="w-6" />
21                  <span class="text-[#787486]">Andras Frank</
                        span>
22              </a>
23              <a href="draw_graph.html" class="hover:bg-[#
                    e0dede] p-2 rounded flex items-center gap-2">
24                  <img src="../assets/graph.png" alt="Draw"
                        class="w-6" />
25                  <span class="text-[#787486]">Desenhe um grafo
                        </span>
26              </a>
27              <a href="tese.html" class="hover:bg-[#e0dede] p-2
                    rounded flex items-center gap-2">
28                  <img src="../assets/book.png" alt="Tesis"
                        class="w-6 h-6" />
29                  <span class="text-[#787486]">Nossa tese</span
                        >
30              </a>
31              <div class="border-t-[2px] border-[#DBDBDB]"></
                    div>
32
33              <a href="tese.html" class="relative flex items-
                    center gap-2 mb-7 text-x">
34                  <img
35                    src="../assets/help.png"
36                    alt="help"
37                    class="w-48 mb-7 mx-auto"
38                  />
39                  <button
40                    class="absolute left-20 bottom-10 bg-[#
                          ffffff] text-[#aba9a9] font-thin py-2 px
                          -4 rounded hover:bg-[#e0dede]"
41                  > Link
42                  </button>
43                </a>
44          </nav>
45      </div>
46  </aside>
```

A figura a seguir ilustra a estrutura hierárquica linear do menu lateral.

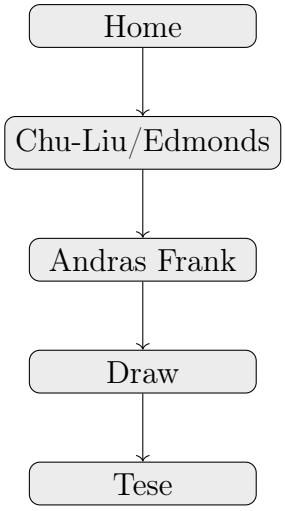


Figura 66: `sidebar.html`: lista vertical reforçando modelo mental estável de navegação.

Comentário: a consistência visual e funcional do menu lateral em todas as páginas reduz a carga cognitiva associada à navegação, permitindo que os usuários se concentrem no conteúdo principal.

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

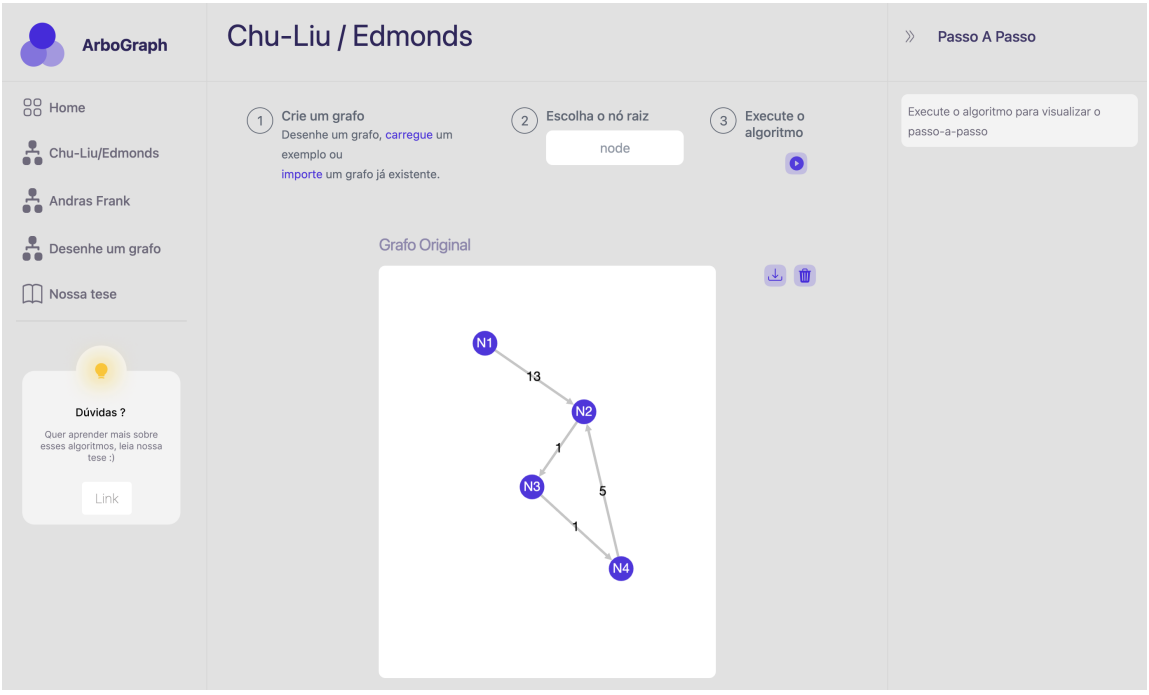


Figura 67: Captura de tela de `chuliu.html`: criação de grafo, seleção de raiz e execução do algoritmo.

A seguir, apresentamos o código completo dessa página.

Markup 8.5: chuliu.html

```
<!DOCTYPE html>
<html lang="pt-BR">

<head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width,
       initial-scale=1" />
    <title>Chu-Liu/Edmonds</title>
    <link rel="icon" type="image/x-icon" href="../assets/logo
       .png"/>
    <script src="https://cdn.tailwindcss.com"></script>
    <script src="https://unpkg.com/cytoscape@3.26.0/dist/
       cytoscape.min.js"></script>

    <link href="https://cdnjs.cloudflare.com/ajax/libs/
       flowbite/2.2.1/flowbite.min.css" rel="stylesheet" />
    <script src="https://cdnjs.cloudflare.com/ajax/libs/
       flowbite/2.2.1/flowbite.min.js"></script>

    <link rel="stylesheet" href="https://pyscript.net/
       releases/2025.3.1/core.css" />
    <script type="module" src="https://pyscript.net/releases
       /2025.3.1/core.js"></script>
</head>

<body class="flex min-h-screen text-gray-800 bg-gray-100">
    <style>
        .py-error, .py-terminal-error {
            display: none !important;
        }
    </style>
    <div class="absolute top-[100px] w-full h-[1px] bg-[#
       DBDBDB]"></div>

    <!-- MENU LATERAL -->
    <div id="sidebar"></div>

    <!-- CONTEÚDO PRINCIPAL -->
    <div id="main-content" class="flex-1 bg-[#E5E5E5] p-6">
        <span class="flex items-center gap-4 mb-10 text-4xl
           text-[#352B67]">Chu-Liu / Edmonds</span>
        <!-- Stepper -->
        <div class="grid grid-cols-50 gap-1">
            <div class="col-span-1 p-2">
            </div>
            <div class="col-span-9 p-2">
                <ol class="items-start gap-4 mx-4 space-y-2
```

```
                    sm:flex sm:space-x-4 sm:space-y-0 rtl:
                    space-x-reverse">
39                   <li class="flex text-gray-500 dark:text-
                        blue-500 space-x-2.5 rtl:space-x-
                        reverse">
40                       <span class="flex items-center
                            justify-center w-8 h-8 border
                            border-gray-500 rounded-full
                            shrink-0 dark:border-blue-500">
41                           1
42                       </span>
43                       <span>
44                           <h3 class="font-medium leading-
                                tight">Crie um grafo</h3>
45                           <span class="text-sm text-gray-
                                500">
46                               Desenhe um grafo,
47                               <span id="load-test-graph"
                                    class="cursor-pointer text
                                    -[#5A3CE5] hover:underline
                                    ">carregue</span>
48                               um exemplo ou <br>
49                               <span id="import-graph" class
                                    ="cursor-pointer text-[#5
                                    A3CE5] hover:underline">
                                    importe</span>
50                               <input type="file" id="file-
                                    input" style="display:
                                    none;" />
51                               um grafo já existente.
52                           </span>
53                       </span>
54                   </li>
55                   <li class="flex text-gray-500 dark:text-
                        gray-400 space-x-2.5 rtl:space-x-
                        reverse">
56                       <span class="flex items-center
                            justify-center w-8 h-8 border
                            border-gray-500 rounded-full
                            shrink-0 dark:border-gray-400">
57                           2
58                       </span>
59                       <span>
60                           <h3 class="font-medium leading-
                                tight">Escolha o nó raiz</h3>
61                           <!-- <p class="text-sm mb-2">
                                Defina o ponto inicial</p> -->
62                           <input type="text" id="root-node"
63                               class="w-full p-2 mt-2
```

```
                                    rounded-lg text-center
                                    focus:outline-none focus:
                                    ring-2 focus:ring-blue-500
                                     focus:border-blue-500
                                    border-transparent"
64                                placeholder="node" />
65                        </span>
66                    </li>
67                    <li class="flex text-gray-500 dark:text-
                        gray-400 space-x-2.5 rtl:space-x-
                        reverse">
68                        <span class="flex items-center
                            justify-center w-8 h-8 border
                            border-gray-500 rounded-full
                            shrink-0 dark:border-gray-400">
69                            3
70                        </span>
71                        <span>
72                            <h3 class="font-medium leading-
                                tight">Execute o algoritmo</h3
                                >
73                            <div class="mt-1 flex justify-
                                center items-center gap-1">
74                                <img id="run-algorithm" src="
                                    ../assets/run.png" alt="
                                    Executar"
75                                    class="cursor-pointer w-8
                                        h-8 rounded-lg hover:
                                        opacity-80 mt-2" />
76                            </div>
77                        </span>
78                    </li>
79                </ol>
80            </div>
81            <div class="col-span-1 p-2">
82            </div>
83        </div>
84        <!-- Grafo original -->
85        <div class="grid grid-cols-10 gap-4 py-4 px-4">
86            <div class="col-span-2 p-4"></div>
87            <div class="col-span-6 p-4">
88                <span class="mb-50 text-xl text-[#9993B7]">
                    Grafo Original</span>
89                <div id="graph-editor" class="my-3 mt-50 py-4
                     px-50 rounded-lg bg-white h-[500px]">
90                </div>
91            </div>
92            <div class="col-span-2 p-4">
93                <div class="py-4 px-2 flex justify-left items
```

```
                    -center gap-1">
94                      <img src="../assets/download.png" alt="
                            Download" id="export-graph-original"
95                          class="my-5 cursor-pointer w-8 h-8
                                rounded-lg hover:opacity-80" />
96                      <img src="../assets/delete.png" alt="
                            Reset" id="reset-graph"
97                          class="my-5 cursor-pointer w-8 h-8
                                rounded-lg hover:opacity-80" />
98                  </div>
99              </div>
100
101         </div>
102         <!-- Arborescência -->
103         <div id="arborescence-section" class="grid grid-cols-
                10 gap-4 py-4 px-4 hidden">
104             <div class="col-span-2 p-4"></div>
105             <div class="col-span-6 p-4">
106                 <span class="mb-50 text-xl text-[#9993B7]">
                        Arborescência</span>
107                 <div id="arborescence-viewer" class="my-3 mt-
                        50 py-4 px-50 rounded-lg bg-white h-[500px
                        ]">
108                 </div>
109             </div>
110             <div class="col-span-2 p-4">
111                 <div class="py-4 px-2 flex justify-left items
                        -center gap-1">
112                     <img src="../assets/download.png" alt="
                            Download" id="export-graph-
                            arborescencia"
113                         class="my-5 cursor-pointer w-8 h-8
                                rounded-lg hover:opacity-80" />
114                 </div>
115             </div>
116         </div>
117
118         <!-- Logs de execução -->
119         <div id="log-section" class="grid grid-cols-10 gap-4
                py-4 px-4 hidden">
120             <div class="col-span-2 p-4"></div>
121             <div class="col-span-6 p-4">
122                 <button id="collapser-button"
123                     class="w-full bg-[#EEEEEE] text-[#948EB3]
                            py-2 px-4 rounded-t-md text-left flex
                            justify-between items-center"
124                     onclick="toggleCollapser()">
125                     Log de Execução
126                     <img id="collapser-icon" src="../assets/
```

```
                        plus.png" alt="Contrair" class="w-5 h-
                        5 hover:opacity-80" />
127                 </button>
128
129                 <!-- Conteúdo colapsável -->
130                 <div id="log-collapser-content"
131                     class="p-4 bg-[#EEEEEE] text-sm text-gray
                        -400 rounded-b-md hidden transition-
                        all duration-500 ease-in-out">
132                     <textarea id="log-output" readonly
133                         class="w-full p-3 rounded h-40 bg-[#
                            EEEEEE] text-sm text-gray-400
                            border-transparent"></textarea>
134                 </div>
135             </div>
136         </div>
137     </div>
138
139     <!-- Passo a Passo -->
140     <div id="right-sidebar" class="w-80 transition-all
        duration-300 border-l-[1px] bg-[#E5E5E5] border-[#
        DBDBDB] overflow-hidden">
141         <div id="title_step_area" class="flex items-center
            gap-6 py-8 mx-4 top-6">
142             <button id="toggle-sidebar" class="text-right
                text-[#4B4277] hover:text-[#2d255d]">
143                 <img id="collapser-icon" src="../assets/
                    back_arrow_right.png" alt="Contrair" class
                    ="w-5 h-5 hover:opacity-80" />
144             </button>
145             <span id="title_step" class="text-lg text-[#4
                B4277] font-medium">Passo A Passo</span>
146         </div>
147         <div id="container_step_by_step" class="my-6 mx-4 gap
            -4 py-2 px-2 bg-[#F5F5F5] rounded-lg">
148             <span id="step_warning" class="text-sm font-light
                text-[#787486]">Execute o algoritmo para
                visualizar o passo-a-passo</span>
149         </div>
150     </div>
151
152     <!-- Toasts -->
153     <div id="toast-danger" class="hidden fixed top-5 right-5
        z-50 flex items-center w-full max-w-xs p-4 mb-4 text-
        gray-500 bg-white rounded-lg shadow-sm dark:text-gray-
        400 dark:bg-gray-800 transition-opacity duration-500
        opacity-100" role="alert">
154         <div class="inline-flex items-center justify-center
            shrink-0 w-8 h-8 text-red-500 bg-red-100 rounded-
```

```
            lg dark:bg-red-800 dark:text-red-200">
155             <!-- SVG de erro -->
156             <svg class="w-5 h-5" aria-hidden="true" fill="
                    currentColor" viewBox="0 0 20 20">
157                 <path d="M10 .5a9.5 9.5 0 1 0 9.5 9.5A9.51
                        9.51 0 0 0 10 .5Zm3.707 11.793a1 1 0 1 1-
                        1.414 1.414L10 11.414l-2.293 2.293a1 1 0 0
                         1-1.414-1.414L8.586 10 6.293 7.707a1 1 0
                        0 1 1.414-1.414L10 8.586l2.293-2.293a1 1 0
                         0 1 1.414 1.414L11.414 10l2.293 2.293Z"/>
158             </svg>
159             <span class="sr-only">Error icon</span>
160         </div>
161         <div id="toast-danger-msg" class="ms-3 text-sm font-
                normal">Ocorreu um erro.</div>
162         <button type="button" class="ms-auto -mx-1.5 -my-1.5
                bg-white text-gray-400 hover:text-gray-900 rounded
                -lg focus:ring-2 focus:ring-gray-300 p-1.5 hover:
                bg-gray-100 inline-flex items-center justify-
                center h-8 w-8 dark:text-gray-500 dark:hover:text-
                white dark:bg-gray-800 dark:hover:bg-gray-700"
                data-dismiss-target="#toast-danger" aria-label="
                Close" onclick="document.getElementById('toast-
                danger').classList.add('hidden')">
163             <span class="sr-only">Close</span>
164             <svg class="w-3 h-3" aria-hidden="true" fill="
                    none" viewBox="0 0 14 14">
165                 <path stroke="currentColor" stroke-linecap="
                        round" stroke-linejoin="round" stroke-
                        width="2" d="m1 1 6 6m0 0 6 6M7 7l6-6M7 7l
                        -6 6"/>
166             </svg>
167         </button>
168     </div>
169
170     <!-- Modal de Imagem -->
171     <div id="image-modal" class="fixed inset-0 z-50 flex
            items-center justify-center bg-black bg-opacity-80
            hidden">
172         <img id="image-modal-img" src="" alt="Imagem Ampliada
                " class="max-w-3xl max-h-[80vh] rounded-lg shadow-
                lg border-4 border-white">
173     </div>
174
175     <!-- Modal Loader -->
176     <div id="loader-modal" class="fixed inset-0 z-50 flex
            items-center justify-center bg-black bg-opacity-30
            hidden">
177         <div class="flex flex-col items-center">
```

```
178             <div class="w-16 h-16 border-4 border-[#5a3ce5]
                    border-t-transparent border-solid rounded-full
                     animate-spin"></div>
179             <span class="mt-4 text-white text-lg">Processando
                    ...</span>
180         </div>
181     </div>
182
183     <!-- Modal para peso da aresta -->
184     <div id="edge-weight-modal" class="fixed inset-0 z-50
           flex items-center justify-center bg-black bg-opacity-
           40 hidden">
185         <div class="bg-white rounded-lg shadow-lg p-6 w-80">
186             <h3 class="text-lg font-semibold mb-4 text-gray-
                    800">Peso da aresta</h3>
187             <input id="edge-weight-input" type="number" min="
                    1" class="w-full p-2 border rounded mb-4 focus
                    :outline-none focus:ring-2 focus:ring-blue-500
                    " placeholder="Digite o peso" />
188             <div class="flex justify-end gap-2">
189             <button id="edge-weight-cancel" class="px-4 py-2
                    rounded bg-gray-200 text-gray-700 hover:bg-
                    gray-300">Cancelar</button>
190             <button id="edge-weight-ok" class="px-4 py-2
                    rounded bg-blue-600 text-white hover:bg-blue-
                    700">OK</button>
191             </div>
192         </div>
193     </div>
194
195     <!-- Scripts -->
196     <script src="../scripts/js/sidebar.js"></script>
197     <script src="../scripts/js/main.js"></script>
198     <script src="../scripts/js/draw_graph.js"></script>
199     <script type="py" src="../scripts/chuliu_page.py" config=
           "../scripts/pyscript.json"></script>
200 </body>
201 </html>
```

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

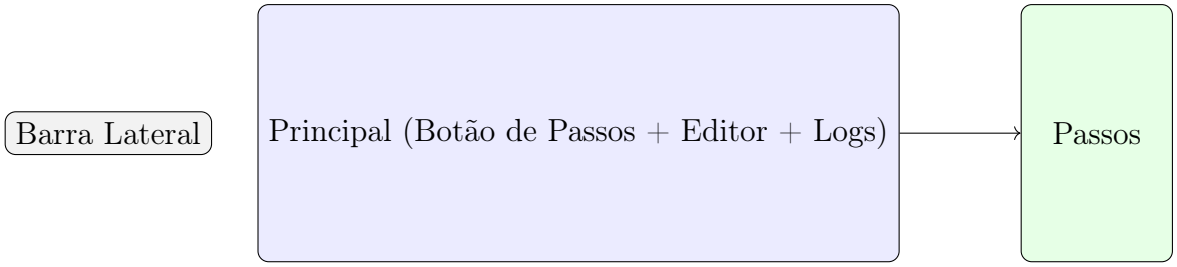


Figura 68: `chuliu.html` - tripartição funcional (navegação, conteúdo interativo, guia de passos).

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

`Andrasfrank.html:` página dedicada ao visualizador do algoritmo de Andras Frank. 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.

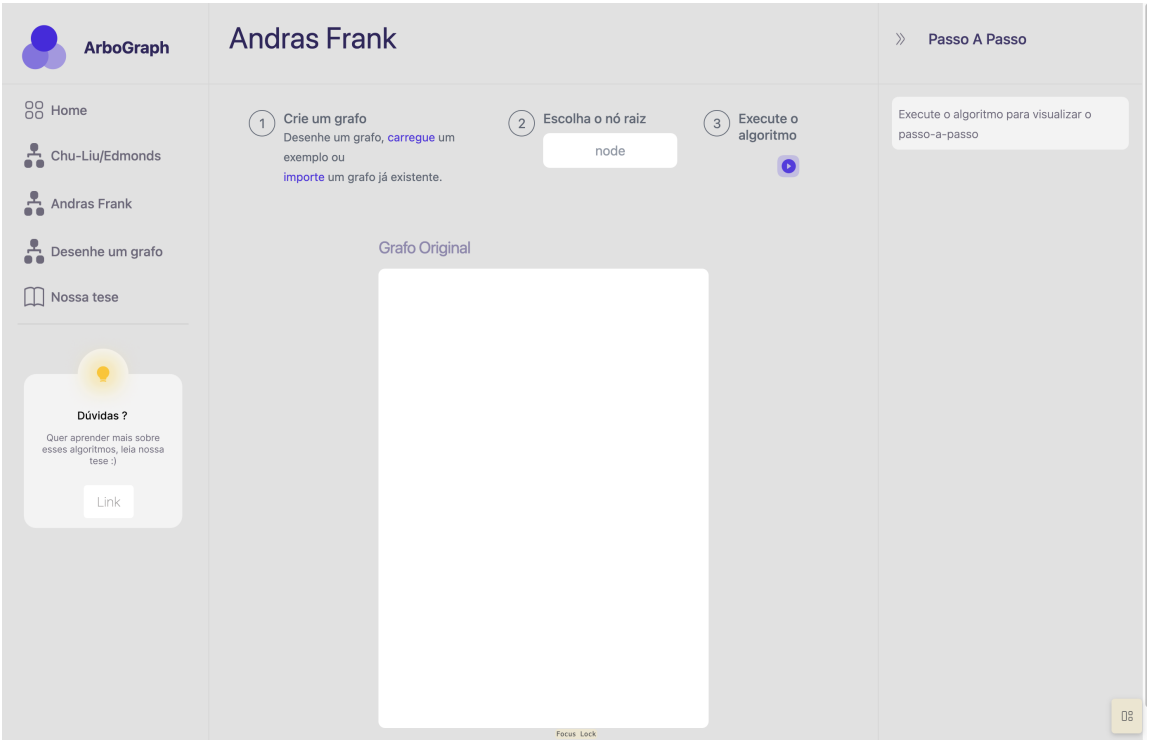


Figura 69: Captura de tela de `andrasfrank.html`: interface para o procedimento em duas fases.

A seguir, apresentamos o código completo dessa página.

Markup 8.6: andrasfrank.html

```
1 <!DOCTYPE html>
2 <html lang="pt-BR">
```

```
3
4 <head>
5     <meta charset="UTF-8" />
6     <meta name="viewport" content="width=device-width,
          initial-scale=1" />
7     <title>Andras Frank</title>
8     <link rel="icon" type="image/x-icon" href="../assets/logo
          .png"/>
9     <script src="https://cdn.tailwindcss.com"></script>
10    <script src="https://unpkg.com/cytoscape@3.26.0/dist/
          cytoscape.min.js"></script>
11
12    <link href="https://cdnjs.cloudflare.com/ajax/libs/
          flowbite/2.2.1/flowbite.min.css" rel="stylesheet" />
13    <script src="https://cdnjs.cloudflare.com/ajax/libs/
          flowbite/2.2.1/flowbite.min.js"></script>
14
15    <link rel="stylesheet" href="https://pyscript.net/
          releases/2025.3.1/core.css" />
16    <script type="module" src="https://pyscript.net/releases
          /2025.3.1/core.js"></script>
17 </head>
18
19 <body class="flex min-h-screen text-gray-800 bg-gray-100">
20    <style>
21        .py-error, .py-terminal-error {
22            display: none !important;
23        }
24    </style>
25    <div class="absolute top-[100px] w-full h-[1px] bg-[#
          DBDBDB]"></div>
26
27    <!-- MENU LATERAL -->
28    <div id="sidebar"></div>
29
30    <!-- CONTEÚDO PRINCIPAL -->
31    <div id="main-content" class="flex-1 bg-[#E5E5E5] p-6">
32        <span class="flex items-center gap-4 mb-10 text-4xl
              text-[#352B67]">Andras Frank</span>
33
34        <div class="grid grid-cols-50 gap-1">
35            <div class="col-span-1 p-2">
36            </div>
37            <div class="col-span-9 p-2">
38                <ol class="items-start gap-4 mx-4 space-y-2
                      sm:flex sm:space-x-4 sm:space-y-0 rtl:
                      space-x-reverse">
39                    <li class="flex text-gray-500 dark:text-
                          blue-500 space-x-2.5 rtl:space-x-
```

```
                        reverse">
40                       <span class="flex items-center
                            justify-center w-8 h-8 border
                            border-gray-500 rounded-full
                            shrink-0 dark:border-blue-500">
41                        1
42                       </span>
43                       <span>
44                           <h3 class="font-medium leading-
                                tight">Crie um grafo</h3>
45                           <span class="text-sm text-gray-
                                500">
46                            Desenhe um grafo,
47                            <span id="load-test-graph"
                                class="cursor-pointer text
                                -[#5A3CE5] hover:underline
                                ">carregue</span>
48                            um exemplo ou <br>
49                            <span id="import-graph" class
                                ="cursor-pointer text-[#5
                                A3CE5] hover:underline">
                                importe</span>
50                            <input type="file" id="file-
                                input" style="display:
                                none;" />
51                            um grafo já existente.
52                           </span>
53                       </span>
54                   </li>
55                   <li class="flex text-gray-500 dark:text-
                        gray-400 space-x-2.5 rtl:space-x-
                        reverse">
56                       <span class="flex items-center
                            justify-center w-8 h-8 border
                            border-gray-500 rounded-full
                            shrink-0 dark:border-gray-400">
57                        2
58                       </span>
59                       <span>
60                           <h3 class="font-medium leading-
                                tight">Escolha o nó raiz</h3>
61                           <!-- <p class="text-sm mb-2">
                                Defina o ponto inicial</p> -->
62                           <input type="text" id="root-node"
63                            class="w-full p-2 mt-2
                                rounded-lg text-center
                                focus:outline-none focus:
                                ring-2 focus:ring-blue-500
                                 focus:border-blue-500
```

```
                                    border-transparent"
64                                placeholder="node" />
65                        </span>
66                    </li>
67                    <li class="flex text-gray-500 dark:text-
                          gray-400 space-x-2.5 rtl:space-x-
                          reverse">
68                        <span class="flex items-center
                              justify-center w-8 h-8 border
                              border-gray-500 rounded-full
                              shrink-0 dark:border-gray-400">
69                            3
70                        </span>
71                        <span>
72                            <h3 class="font-medium leading-
                                  tight">Execute o algoritmo</h3
                                  >
73                            <div class="mt-1 flex justify-
                                  center items-center gap-1">
74                                <img id="run-algorithm" src="
                                      ../assets/run.png" alt="
                                      Executar"
75                                    class="cursor-pointer w-8
                                        h-8 rounded-lg hover:
                                        opacity-80 mt-2" />
76                            </div>
77                        </span>
78                    </li>
79                </ol>
80            </div>
81            <div class="col-span-1 p-2">
82            </div>
83        </div>
84
85        <!-- Grafo original -->
86        <div class="grid grid-cols-10 gap-4 py-4 px-4">
87            <div class="col-span-2 p-4"></div>
88            <div class="col-span-6 p-4">
89                <span class="mb-50 text-xl text-[#9993B7]">
                      Grafo Original</span>
90                <div class="my-3 mt-50 py-4 px-50 rounded-lg
                      bg-white">
91                    <!-- <span id="draw_warning" class="text-
                          sm font-light text-[#787486] mx-9">
                          Desenhe ou importe um grafo.</span> --
                          >
92                    <div id="graph-editor" class="my-3 mt-50
                          py-4 px-50 rounded-lg bg-white h-[500
                          px]">
```

```
93                      </div>
94                  </div>
95              </div>
96              <div class="col-span-2 p-4">
97                  <div class="py-4 px-2 flex justify-left items
                        -center gap-1">
98                      <img src="../assets/download.png" alt="
                            Download" id="export-graph-original"
99                          class="my-5 cursor-pointer w-8 h-8
                                rounded-lg hover:opacity-80 hidden
                                " />
100                 </div>
101             </div>
102
103         </div>
104         <!-- Arborescência -->
105         <div id="arborescence-section" class="grid grid-cols-
                10 gap-4 py-4 px-4 hidden">
106             <div class="col-span-2 p-4"></div>
107             <div class="col-span-6 p-4">
108                 <span class="mb-50 text-xl text-[#9993B7]">
                        Arborescência</span>
109                 <div id="arborescence-viewer" class="my-3 mt-
                        50 py-4 px-50 bg-white rounded-lg h-[500px
                        ]">
110                 </div>
111             </div>
112             <div class="col-span-2 p-4">
113                 <div class="py-4 px-2 flex justify-left items
                        -center gap-1">
114                     <img src="../assets/download.png" alt="
                            Download" id="export-graph-
                            arborescencia"
115                         class="my-5 cursor-pointer w-8 h-8
                                rounded-lg hover:opacity-80" />
116                 </div>
117             </div>
118
119         </div>
120
121         <!-- Logs de execução -->
122         <div id="log-section" class="grid grid-cols-10 gap-4
                py-4 px-4 hidden">
123             <div class="col-span-2 p-4"></div>
124             <div class="col-span-6 p-4">
125                 <button id="collapser-button"
126                     class="w-full bg-[#EEEEEE] text-[#948EB3]
                            py-2 px-4 rounded-t-md text-left flex
                            justify-between items-center"
```

```
127                     onclick="toggleCollapser()">
128                     Log de Execução
129                     <img id="collapser-icon" src="../assets/
                            plus.png" alt="Contrair" class="w-5 h-
                            5 hover:opacity-80" />
130                 </button>
131
132                 <!-- Conteúdo colapsável -->
133                 <div id="log-collapser-content"
134                     class="p-4 bg-[#EEEEEE] text-sm text-gray
                            -400 rounded-b-md hidden transition-
                            all duration-500 ease-in-out">
135                     <textarea id="log-output" readonly
136                         class="w-full p-3 rounded h-40 bg-[#
                                EEEEEE] text-sm text-gray-400
                                border-transparent"></textarea>
137                 </div>
138             </div>
139         </div>
140     </div>
141
142     <!-- Passo a Passo -->
143     <div id="right-sidebar" class="w-80 transition-all
            duration-300 border-l-[1px] bg-[#E5E5E5] border-[#
            DBDBDB] overflow-hidden">
144         <div id="title_step_area" class="flex items-center
                gap-6 py-8 mx-4 top-6">
145             <button id="toggle-sidebar" class="text-right
                    text-[#4B4277] hover:text-[#2d255d]">
146                 <img id="collapser-icon" src="../assets/
                        back_arrow_right.png" alt="Contrair" class
                        ="w-5 h-5 hover:opacity-80" />
147             </button>
148             <span id="title_step" class="text-lg text-[#4
                    B4277] font-medium">Passo A Passo</span>
149         </div>
150         <div id="container_step_by_step" class="my-6 mx-4 gap
                -4 py-2 px-2 bg-[#F5F5F5] rounded-lg">
151             <span id="step_warning" class="text-sm font-light
                    text-[#787486]">Execute o algoritmo para
                    visualizar o passo-a-passo</span>
152         </div>
153     </div>
154
155     <!-- Toast de Erro -->
156     <div id="toast-danger" class="hidden fixed top-5 right-5
            z-50 flex items-center w-full max-w-xs p-4 mb-4 text-
            gray-500 bg-white rounded-lg shadow-sm dark:text-gray-
            400 dark:bg-gray-800 transition-opacity duration-500
```

```
        opacity-100" role="alert">
157         <div class="inline-flex items-center justify-center
               shrink-0 w-8 h-8 text-red-500 bg-red-100 rounded-
               lg dark:bg-red-800 dark:text-red-200">
158             <!-- SVG de erro -->
159             <svg class="w-5 h-5" aria-hidden="true" fill="
                   currentColor" viewBox="0 0 20 20">
160                 <path d="M10 .5a9.5 9.5 0 1 0 9.5 9.5A9.51
                       9.51 0 0 0 10 .5Zm3.707 11.793a1 1 0 1 1-
                       1.414 1.414L10 11.414l-2.293 2.293a1 1 0 0
                        1-1.414-1.414L8.586 10 6.293 7.707a1 1 0
                       0 1 1.414-1.414L10 8.586l2.293-2.293a1 1 0
                        0 1 1.414 1.414L11.414 10l2.293 2.293Z"/>
161             </svg>
162             <span class="sr-only">Error icon</span>
163         </div>
164         <div id="toast-danger-msg" class="ms-3 text-sm font-
               normal">Ocorreu um erro.</div>
165         <button type="button" class="ms-auto -mx-1.5 -my-1.5
               bg-white text-gray-400 hover:text-gray-900 rounded
               -lg focus:ring-2 focus:ring-gray-300 p-1.5 hover:
               bg-gray-100 inline-flex items-center justify-
               center h-8 w-8 dark:text-gray-500 dark:hover:text-
               white dark:bg-gray-800 dark:hover:bg-gray-700"
               data-dismiss-target="#toast-danger" aria-label="
               Close" onclick="document.getElementById('toast-
               danger').classList.add('hidden')">
166             <span class="sr-only">Close</span>
167             <svg class="w-3 h-3" aria-hidden="true" fill="
                   none" viewBox="0 0 14 14">
168                 <path stroke="currentColor" stroke-linecap="
                       round" stroke-linejoin="round" stroke-
                       width="2" d="m1 1 6 6m0 0 6 6M7 7l6-6M7 7l
                       -6 6"/>
169             </svg>
170         </button>
171     </div>
172
173     <!-- Modal de Imagem -->
174     <div id="image-modal" class="fixed inset-0 z-50 flex
           items-center justify-center bg-black bg-opacity-80
           hidden">
175         <img id="image-modal-img" src="" alt="Imagem Ampliada
               " class="max-w-3xl max-h-[80vh] rounded-lg shadow-
               lg border-4 border-white">
176     </div>
177
178     <!-- Modal Loader -->
179     <div id="loader-modal" class="fixed inset-0 z-50 flex
```

```
        items-center justify-center bg-black bg-opacity-30
        hidden">
180         <div class="flex flex-col items-center">
181             <div class="w-16 h-16 border-4 border-[#5a3ce5]
                    border-t-transparent border-solid rounded-full
                     animate-spin"></div>
182             <span class="mt-4 text-white text-lg">Processando
                    ...</span>
183         </div>
184     </div>
185
186     <!-- Modal para peso da aresta -->
187     <div id="edge-weight-modal" class="fixed inset-0 z-50
        flex items-center justify-center bg-black bg-opacity-
        40 hidden">
188         <div class="bg-white rounded-lg shadow-lg p-6 w-80">
189             <h3 class="text-lg font-semibold mb-4 text-gray-
                    800">Peso da aresta</h3>
190             <input id="edge-weight-input" type="number" min="
                    1" class="w-full p-2 border rounded mb-4 focus
                    :outline-none focus:ring-2 focus:ring-blue-500
                    " placeholder="Digite o peso" />
191             <div class="flex justify-end gap-2">
192             <button id="edge-weight-cancel" class="px-4 py-2
                    rounded bg-gray-200 text-gray-700 hover:bg-
                    gray-300">Cancelar</button>
193             <button id="edge-weight-ok" class="px-4 py-2
                    rounded bg-blue-600 text-white hover:bg-blue-
                    700">OK</button>
194             </div>
195         </div>
196     </div>
197
198     <!-- Scripts -->
199     <script src="../scripts/js/sidebar.js"></script>
200     <script src="../scripts/js/main.js"></script>
201     <script src="../scripts/js/draw_graph.js"></script>
202     <script type="py" src="../scripts/andrasfrank_page.py"
        config="../scripts/pyscript.json"></script>
203 </body>
204
205 </html>
```

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

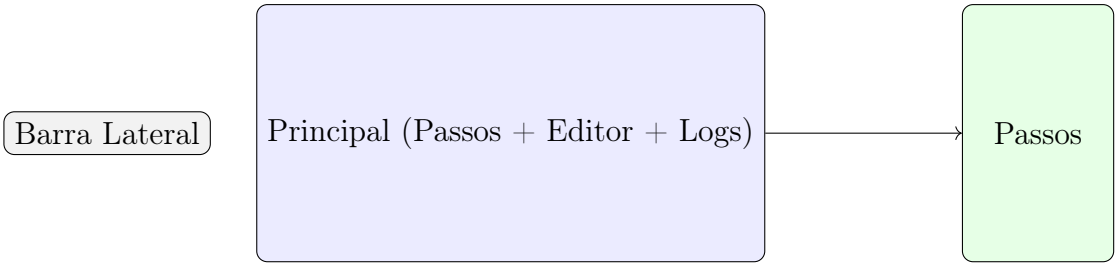


Figura 70: `andrasfrank.html` - reutilização de padrão para consistência cognitiva.

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

O ecossistema de páginas cria uma narrativa pedagógica: contextualização (`home`) → experimentação livre (`draw_graph`) → exploração guiada (`chuliu`, `andrasfrank`) → consolidação formal (`tese`). Tal sequência segue princípios de *spiral curriculum* e reduz carga intrínseca inicial.

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

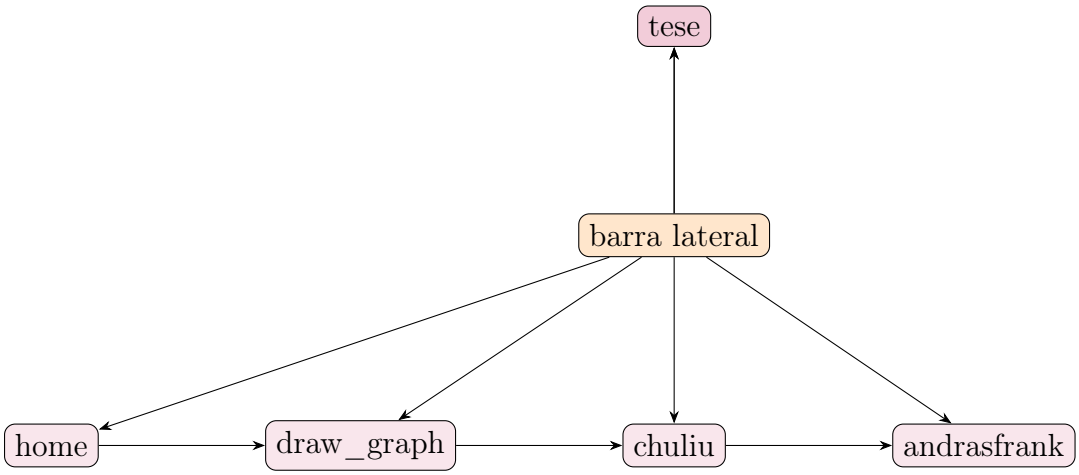


Figura 71: A `barra lateral` injeta navegação consistente; páginas de algoritmo formam trilha exploratória (instância livre →algoritmo 1 →algoritmo 2).

Síntese arquitetural das páginas

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.

8.4 Avaliação da Interface

wip A avaliação preliminar da interface *web* envolveu testes com um grupo de usuários composto por estudantes de graduação em ciência da computação e profissionais da área. Os participantes foram convidados a utilizar o visualizador interativo para criar grafos, executar os algoritmos de Chu-Liu/Edmonds e Andras Frank, e explorar os passos detalhados. Feedback qualitativo foi coletado através de questionários e entrevistas, focando na usabilidade, clareza das instruções, eficácia do passo a passo e compreensão dos algoritmos.

wip Os resultados indicaram que a maioria dos usuários considerou a interface intuitiva e fácil de navegar. O passo a passo foi particularmente elogiado por facilitar a compreensão dos algoritmos, permitindo que os usuários acompanhassem cada etapa do processo. Alguns participantes sugeriram melhorias, como a inclusão de mais exemplos de grafos e a possibilidade de salvar e carregar sessões de trabalho.

8.5 Comentários finais sobre a Interface *web*

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

9 Conclusão

Iniciamos essa tese falando sobre o princípio da Navalha de Ockham, que nos convida a buscar soluções simples, mas não simplificadas. Retomamos esse mote para refletir sobre o ponto de chegada: o visualizador interativo de arborescências dirigidas que procura apresentar de forma acessível e simples os algoritmos envolvidos no problema da r-arborescência de custo mínimo.

Nessa trajetória, os fins não justificam os meios, pois os meios pelos quais resolvemos o problema é o ponto central de discussão desse trabalho, os algoritmos de Chu-liu e Edmonds e o de Andras Frank.

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.

9.1 Contribuições

Este trabalho contribui para a interseção entre teoria dos grafos e design pedagógico criando um visualizador interativo de arborescências dirigidas que promove a compreensão de algoritmos que resolvem o problema da r-arborescência de custo-mínimo. As principais contribuições incluem:

- **Integração de Algoritmos Complexos:** Implementação detalhada dos algoritmos de Chu-Liu/Edmonds e Andras Frank, destacando suas bases teóricas e operacionais, e demonstrando como decisões locais se traduzem em garantias globais de otimalidade.
- **Design Pedagógico:** Aplicação de princípios de interação humano-computador e de aprendizagem multimídia para criar uma interface que facilita a compreensão de conceitos complexos, reduzindo a carga cognitiva e promovendo o engajamento ativo.
- **Arquitetura Modular e Reutilizável:** Desenvolvimento de uma estrutura *web* que permite fácil manutenção, extensão e consistência cognitiva entre diferentes páginas, utilizando tecnologias modernas como PyScript, NetworkX e Matplotlib.

- **Ferramenta Acessível e Reprodutível:** Criação de uma ferramenta que pode ser acessada diretamente no navegador, eliminando barreiras de adoção e permitindo que usuários experimentem e aprendam de forma autônoma.
- **Documentação e Avaliação:** Fornecimento de documentação detalhada e avaliação preliminar da interface, oferecendo insights sobre a eficácia pedagógica e identificando áreas para melhorias futuras.

Essas 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.

9.2 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. A dependência de bibliotecas externas, como Cytoscape.js e PyScript, também pode introduzir desafios de compatibilidade e manutenção a longo prazo. Finalmente, a avaliação da ferramenta foi preliminar e baseada em um número limitado de usuários; estudos mais abrangentes são necessários para validar sua eficácia pedagógica.

9.3 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 mais extensos, 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.

A Notas sobre matroides e sua interseção

Nesta breve nota, registramos definições mínimas para dar contexto às menções a interseção de matroides no corpo do texto. Para referências e desenvolvimento completo, ver, por exemplo, Schrijver [21].

Um **matroide** $M = (E, \mathcal{I})$ é dado por um conjunto finito E e uma família $\mathcal{I} \subseteq 2^E$ de conjuntos *independentes* que satisfazem: (i) $\varnothing \in \mathcal{I}$; (ii) se $I \in \mathcal{I}$ e $J \subseteq I$, então $J \in \mathcal{I}$ (hereditariedade); (iii) se $I, J \in \mathcal{I}$ e $|I| < |J|$, então existe $e \in J \setminus I$ com $I \cup \{e\} \in \mathcal{I}$ (troca).

Exemplos clássicos incluem: (a) o matroide gráfico, em que E é o conjunto de arestas de um grafo e $\mathcal{I}$ são os conjuntos acíclicos; (b) o matroide de partição, que impõe no máximo uma escolha por parte; (c) o matroide linear, em que E é um conjunto de vetores e $\mathcal{I}$ são subconjuntos linearmente independentes.

A **interseção de matroides** pergunta por um conjunto $X \subseteq E$ de maior cardinalidade (ou de menor custo no caso ponderado) que seja independente simultaneamente em dois matroides $M_1 = (E, \mathcal{I}_1)$ e $M_2 = (E, \mathcal{I}_2)$, isto é, $X \in \mathcal{I}_1 \cap \mathcal{I}_2$. O problema admite algoritmos polinomiais gerais, e muitas formulações clássicas em grafos se enquadram nesse arcabouço.

No contexto de arborescências dirigidas, estruturas do tipo "no máximo um arco entrando em cada vértice" podem ser modeladas por matroides de partição, enquanto restrições que evitam ciclos dirigidos aparecem via matroide gráfico orientado e técnicas afins. Isso motiva a citação no corpo do texto quando discutimos o empacotamento de múltiplas arborescências e condições por cortes.

B Conceitos de Álgebra Linear

Este apêndice reúne noções básicas de álgebra linear úteis como referência rápida. O foco recai sobre ideias que aparecem implicitamente ao longo do texto (por exemplo, ao tratar de potenciais, estruturas matriciais de digrafos e decomposições ortogonais).

Espaços vetoriais, combinações e base

Um **espaço vetorial** V sobre um corpo $\mathbb{K}$ (tipicamente $\mathbb{R}$) é um conjunto com duas operações, soma $+$ e multiplicação por escalar, que satisfazem os axiomas usuais (associatividade, comutatividade da soma, existência de neutro e inverso aditivo, distributividade etc.). Dado um subconjunto $S = \{v_1, \dots, v_k\} \subseteq V$, uma **combinação linear** é $\sum_i \alpha_i v_i$ com $\alpha_i \in \mathbb{K}$. Dizemos que S é **linearmente independente** se $\sum_i \alpha_i v_i = 0$ implica $\alpha_i = 0$ para todo i. Um conjunto gerador minimal (independente) chama-se **base** de V, e o número de vetores numa base é a **dimensão** $\dim V$ (bem-definida).

Transformações lineares e matrizes

Uma **transformação linear** $T : V \to W$ satisfaz $T(\alpha x + \beta y) = \alpha T(x) + \beta T(y)$. Fixadas bases de V e W, toda transformação linear é representada por uma **matriz** A e vale $[T(x)] = A[x]$. A composição de transformações corresponde ao produto de matrizes. A mudança de base é realizada por matrizes de passagem (sem alterar propriedades invariantes como posto).

Sistemas lineares, posto e nulidade

Resolver $Ax = b$ equivale a aplicar *eliminação de Gauss* para obter forma escalonada. O **posto** $\operatorname{rank}(A)$ é o número de linhas (ou colunas) linearmente independentes; o **núcleo** $\ker(A) = \{x : Ax = 0\}$ tem dimensão **nulidade** $\operatorname{null}(A)$. O **teorema posto–nulidade** afirma $\operatorname{rank}(A) + \operatorname{null}(A) = n$ (número de colunas de A). A existência e unicidade de soluções dependem de $\operatorname{rank}(A)$ e da consistência de b.

Autovalores e autovetores

Para $A \in \mathbb{R}^{n\times n}$, um **autovalor** λ e um **autovetor** não nulo v satisfazem $Av = \lambda v$. Se há n autovetores linearmente independentes, A é **diagonalizável** ($A = P\ \operatorname{diag}(\lambda_i)\, P^{-1}$). Para matrizes simétricas reais, vale o **teorema espectral**: existem autovetores ortonormais que formam base e autovalores reais, possibilitando decomposições ortogonais.

Produto interno e ortogonalidade

Um **produto interno** $\langle x, y\rangle$ induz norma $\|x\| = \sqrt{\langle x, x\rangle}$. Vetores são **ortogonais** se $\langle x, y\rangle = 0$. A projeção ortogonal de u no subespaço W minimiza a distância $\|u - w\|$ sobre $w \in W$. O processo de **Gram–Schmidt** transforma um conjunto independente em uma base ortonormal, útil para estabilidade numérica e interpretação geométrica.

Matrizes de grafos e o Laplaciano

Para um digrafo $G = (V, E)$, a **matriz de incidência** $B \in \mathbb{R}^{|V|\times|E|}$ codifica a orientação dos arcos; seu posto relaciona-se ao número de componentes fortemente conexas. O **Laplaciano** dirigido pode ser definido como $L = D_{\text{out}} - A$ (ou variações), onde A é a matriz de adjacência e D_{out} é diagonal com graus de saída. Em grafos não dirigidos, cofatores de L contam árvores geradoras (*teorema matriz–árvore*); em variantes dirigidas, cofatores apropriados contam arborescências enraizadas em r, conectando linearmente estruturas combinatórias e contagem de arborescências.

Observação sobre potenciais e custos reduzidos

Ao ajustar custos por *potenciais* nos vértices (transformações do tipo $c'(u, v) = c(u, v) + \pi(u) - \pi(v)$), preservamos diferenças ao longo de ciclos (somas telescópicas nulas), o que ecoa a ideia de adicionar uma 1-forma exata. Trata-se, no fundo, de uma transformação linear no espaço dos pesos de arestas parametrizada por funções em V, com forte paralelismo a mudanças de base que preservam invariantes essenciais (como somas em ciclos).

Referências

[1] Mathieu Bastian, Sebastien Heymann e Mathieu Jacomy. “Gephi: An Open Source Software for Exploring and Manipulating Networks”. Em: *Proceedings of the International AAAI Conference on Web and Social Media*. Vol. 3. 1. 2009, pp. 361–362.

[2] J. A. Bondy e U. S. R. Murty. *Graph Theory with Applications*. Springer, 2008.

[3] Y. J. Chu e T. H. Liu. “On the Shortest Arborescence of a Directed Graph”. Em: *Scientia Sinica* 14 (1965), pp. 1396–1400.

[4] T. H. Cormen et al. *Introduction to Algorithms*. 3rd. MIT Press, 2009.

[5] Reinhard Diestel. *Graph Theory*. 5th. Springer, 2017. ISBN: 978-3662536216.

[6] J. Edmonds. “Optimum Branchings”. Em: *Journal of Research of the National Bureau of Standards* 71B (1967), pp. 233–240.

[7] A. Frank e G. Hajdu. “A Simple Algorithm and Min–Max Formula for the Inverse Arborescence Problem”. Em: *Algorithms* 7.4 (2014), pp. 637–647. DOI: 10.3390/a7040637.

[8] Andŕas Frank. “A Weighted Matroid Intersection Approach to R-Arborescences and Related Problems”. Em: *Paths, Flows, and VLSI-Layout*. Ed. por Andŕas Frank et al. Two-phase primal–dual method for minimum-cost arborescences; placeholder citation. Springer, 1981.

[9] Emden R. Gansner e Stephen C. North. *Graphviz — Graph Visualization Software*. https://graphviz.org/. Acessado em 2025.

[10] Aric A. Hagberg, Daniel A. Schult e Pieter J. Swart. “Exploring Network Structure, Dynamics, and Function using NetworkX”. Em: *Proceedings of the 7th Python in Science Conference (SciPy)*. 2008, pp. 11–15.

[11] Steven Halim et al. *VisuAlgo*. https://visualgo.net/. Acesso didático a visualizações interativas de algoritmos, acessado em 2025.

[12] Christopher D. Hundhausen, Sarah A. Douglas e John T. Stasko. “A Meta-Study of Algorithm Visualization Effectiveness”. Em: *Journal of Visual Languages & Computing* 13.3 (2002), pp. 259–290.

[13] J. Kleinberg e É. Tardos. *Algorithm Design*. Addison-Wesley, 2006.

[14] Thomas Kluyver et al. “Jupyter Notebooks – a publishing format for reproducible computational workflows”. Em: *Positioning and Power in Academic Publishing: Players, Agents and Agendas*. IOS Press, 2016, pp. 87–90.

[15] Jill H. Larkin e Herbert A. Simon. “Why a diagram is (sometimes) worth ten thousand words”. Em: *Cognitive Science* 11.1 (1987), pp. 65–100.

[16] Richard E. Mayer. *Multimedia Learning*. 2nd. Cambridge University Press, 2009.

[17] Thomas L. Naps et al. “Exploring the Role of Visualization and Engagement in Computer Science Education”. Em: *Proceedings of the 34th SIGCSE Technical Symposium on Computer Science Education*. ACM, 2003, pp. 131–152.

[18] Jakob Nielsen. *Usability Engineering*. Fonte das heurísticas clássicas de usabilidade. Morgan Kaufmann, 1994. ISBN: 978-0125184069.

[19] Allan Paivio. *Mental Representations: A Dual Coding Approach*. Oxford University Press, 1990.

[20] Yvonne Rogers, Helen Sharp e Jenny Preece. *Interaction Design: Beyond Human-Computer Interaction*. 3rd. Wiley, 2011. ISBN: 978-0470665763.

[21] A. Schrijver. *Combinatorial Optimization: Polyhedra and Efficiency*. Springer, 2003.

[22] Paul Shannon et al. "Cytoscape: A Software Environment for Integrated Models of Biomolecular Interaction Networks". Em: *Genome Research* 13.11 (2003), pp. 2498–2504.

[23] Ben Shneiderman. "The eyes have it: A task by data type taxonomy for information visualizations". Em: *Proceedings 1996 IEEE Symposium on Visual Languages*. IEEE, 1996, pp. 336–343.

[24] Ben Shneiderman et al. *Designing the User Interface: Strategies for Effective Human-Computer Interaction*. 6th. Pearson, 2016. ISBN: 978-0134380384.

[25] John Sweller. "Cognitive load during problem solving: Effects on learning". Em: *Cognitive Science* 12.2 (1988), pp. 257–285.

[26] David Tall. *Advanced Mathematical Thinking*. Kluwer Academic Publishers, 1991.

[27] Till Tantau. *The TikZ and PGF Packages: Manual for version 3.0.0*. `https://ctan.org/pkg/pgf`. 2015.

[28] Colin Ware. *Information Visualization: Perception for Design*. 3rd. Morgan Kaufmann, 2012.

[29] Douglas B. West. *Introduction to Graph Theory*. 2nd. Prentice Hall, 2001. ISBN: 978-0130144003.

[30] *yEd Graph Editor*. `https://www.yworks.com/products/yed`. Editor de grafos com layouts automáticos, acessado em 2025.