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

7 de outubro de 2025

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 [4].

Essa busca dialoga com um princípio 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 [4, 5].

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. [4, 5]).

Nosso interesse, porém, não é apenas encontrar a arborescência mínima: o percurso até ela também importa, pois revela propriedades estruturais dos dígrafos 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 [1, 4]; 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 [5]. 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.

Assim sendo, analisamos e implementamos, em Python, essas duas abordagens, e apresentaremos detalhes das implementações, desafios enfrentados e soluções adotadas. Realizamos testes de volume com milhares de instâncias geradas aleatoriamente, registrando resultados em arquivos CSV e de log; os custos obtidos pelo Chu–Liu/Edmonds e pelas duas variantes de András Frank coincidem, corroborando a correção das implementações.

Adicionalmente, desenvolvemos uma aplicação web com fins didáticos, utilizando o framework PyScript e as bibliotecas NetworkX e Matplotlib, que permitem construir grafos dirigidos interativamente, escolher o vértice-raiz, executar o algoritmo de Chu–Liu/Edmonds e acompanhar, passo a passo, a evolução do grafo e o registro detalhado da execução (log). A interface inclui operações de adicionar arestas com pesos, carregar um grafo de teste e exportar a instância em formato JSON, facilitando a experimentação por estudantes e educadores.

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 dígrafos 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. **Fundamentação teórica**: revisão da literatura sobre arborescências em grafos dirigidos, incluindo definições, propriedades e resultados relevantes.

2. **Análise teórica**: consolidação dos conceitos de dígrafos e arborescências, compondo as formulações primal (normalização de custos e contração/reexpansão de ciclos no algoritmo de Chu–Liu/Edmonds) e dual (cortes dirigidos e função c-viável no método de András Frank), destacando resultados e intuições estruturais.

3. **Implementação computacional**: implementação em Python das rotinas de normalização dos custos de entrada, construção de F^*, detecção e contração de ciclos e reconstrução da solução (Chu–Liu/Edmonds), bem como das duas fases do método de András Frank; além de uma suíte de testes automatizados em larga escala sobre instâncias aleatórias com até centenas de vértices, verificando a coincidência dos custos entre os métodos e registrando resultados em CSV e log.

4. **Aplicação pedagógica**: desenvolvimento de uma aplicação web interativa (PyScript + NetworkX + Matplotlib) que permite montar instâncias, escolher o vértice-raiz e acompanhar, passo a passo, a execução do algoritmo com visualização do grafo e dos pesos das arestas, log textual e importação/exportação em JSON para facilitar a reprodução de experimentos.

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, dígrafos, propriedade em dígrafos, dígrafos 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

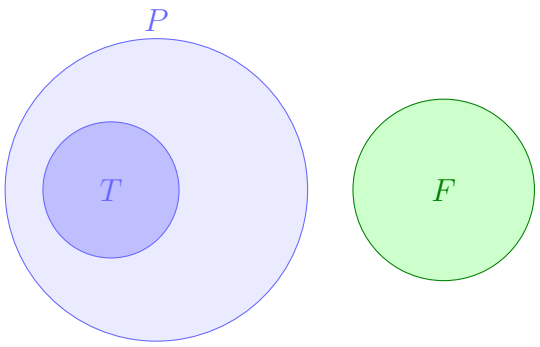
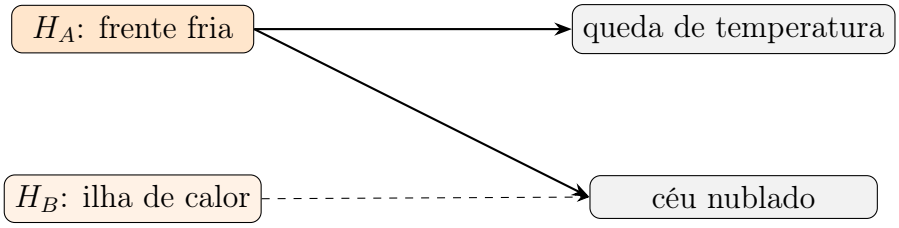


Figura 1: Relações entre os conjuntos de organismos: $T \subseteq P$ (toda árvore é planta) e F é disjunto de P.

- (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.



Ambas as opções H_A e $H\_A + H\_B$ cobrem E;
por parcimônia, preferimos apenas H_A.

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.

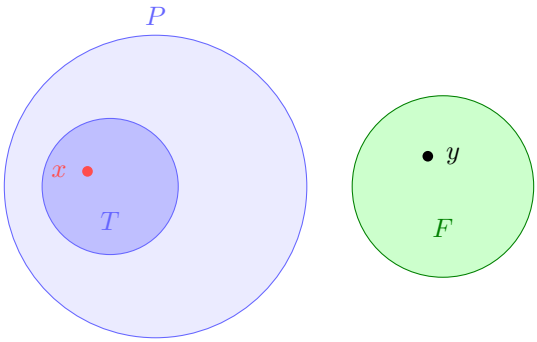


Figura 3: Pertinência: $x \in T \subseteq P$ (ponto vermelho dentro de T) e $y \in F$ (ponto preto); logo $y \notin P$ e $y \notin T$.

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).

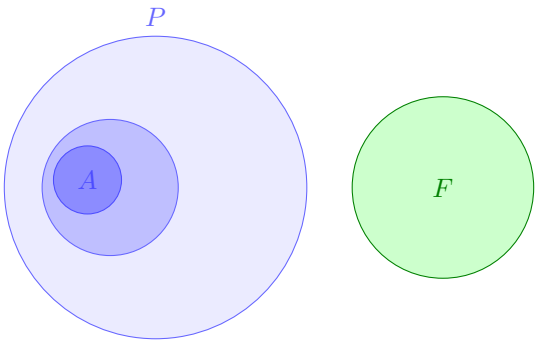


Figura 4: Inclusão: $A \subseteq T \subset P$ (círculos aninhados) e $T \cap F = \varnothing$ (círculos disjuntos).

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).

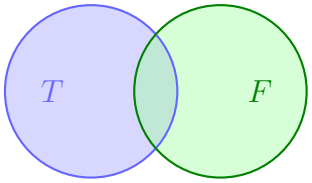


Figura 5: União: a área colorida representa T $cup F$; a sobreposição evidencia a intersecção T $cap F$ (hipotética).

- **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).

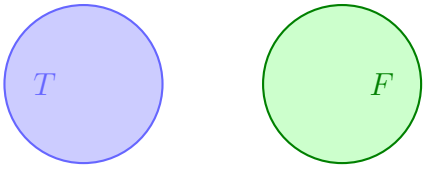


Figura 6: União disjunta: como $TcapF = varnothing$, tem-se $TuplusF = TcupF$.

- **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.

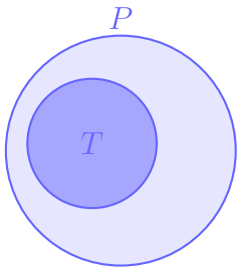


Figura 7: Interseção: como $TsubseteqP$, $TcapP = T$ (a região escura é T).

- **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.

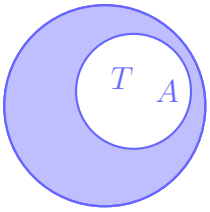


Figura 8: Diferença: região azul representa T $setminus A$ (árvores que não têm folhas verdes).

- **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.

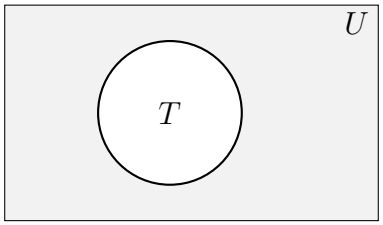


Figura 9: Complemento: a área cinza representa $T^c = U setminus T$.

- **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).

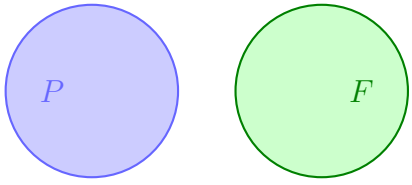


Figura 10: Diferença simétrica: como $PcapF = varnothing$, temos $PDelta\, F = PcupF$.

- **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 $T \times F = \{(t_1, f_1), (t_1, f_2), (t_2, f_1), (t_2, f_2)\}$.

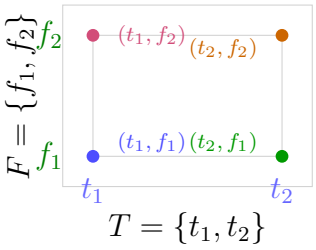


Figura 11: Produto cartesiano: pontos representam os pares de T $times F$ para $T = \{t_1, t_2\}$ e $F = \{f_1, f_2\}$.

- **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, $|2^U| = 4 = 2^{|U|}$.

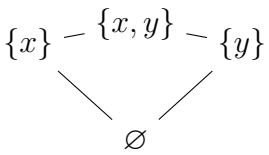


Figura 12: Conjunto das partes: diagrama de Hasse de 2^U para $U = \{x, y\}$.

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, $(P \cup F)^c = P^c \cap F^c$.

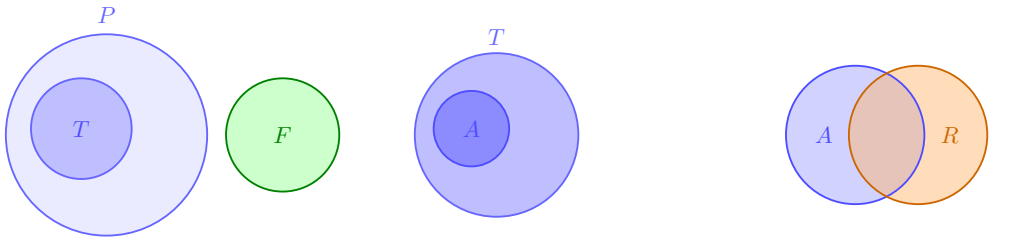
2.1.4 Coleção

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.

Uma formulação padrão usa variáveis x_a para cada arco $a \in A$ indicando sua seleção (na versão inteira $x_a \in \{0, 1\}$; se permitirmos valores fracionários $0 \leq x_a \leq 1$, obtemos a relaxação contínua, que é um programa linear (PL)): Explicando de forma mais canônica, o problema de arborescência mínima pode ser formulado como um programa linear (PL) primal–dual[1]

[1]Por "PL primal–dual" entendemos um par de formulações lineares acopladas: o primal escolhe variáveis x_a para minimizar o custo total, e o dual atribui potenciais $y(\cdot)$ a cortes/vértices impondo $c'(u, v) \geq 0$.

Na matemática, uma **relação** entre dois conjuntos A e B é uma maneira de associar elementos de A com elementos de B. Uma **função** é um tipo especial de relação que associa cada elemento de A a exatamente um elemento de B.

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.

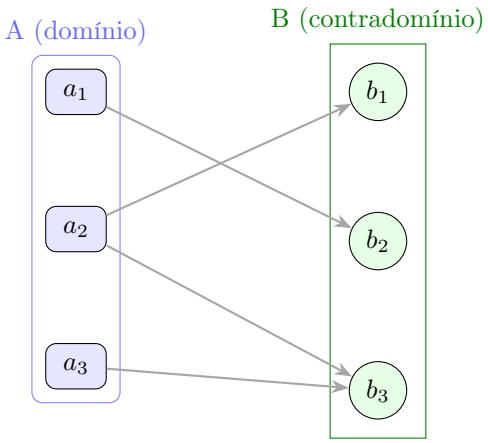


Figura 14: Relação $R \subseteq A \times B$. Cada seta representa um par $(a, b) \in R$ (isto é, $a\,R\,b$). As formas/cores distinguem domínio (A, retângulos azuis) de contradomínio (B, círculos verdes). Note que a_2 se relaciona com b_1 e b_3; logo, este R *não é função*.

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.

O dual fornece um limitante inferior (dualidade fraca); quando a solução primal usa apenas arcos de custo reduzido zero e cada corte ativo é atravessado exatamente uma vez, os valores coincidem (dualidade forte), certificando otimalidade; ver [8, 5].

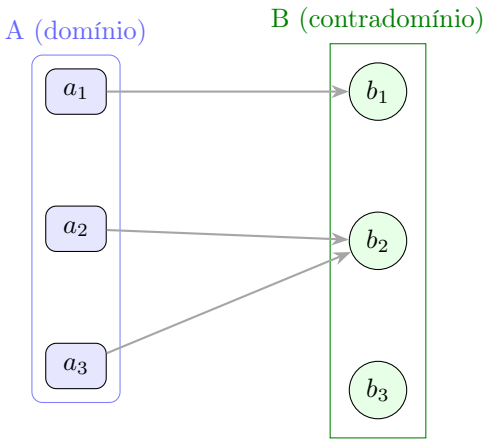


Figura 15: Função $f : A \to B$. Cada elemento de A tem *exatamente uma* imagem em B.

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 pesos 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, $f(A) = \{f(a) \mid a \in A\}$.

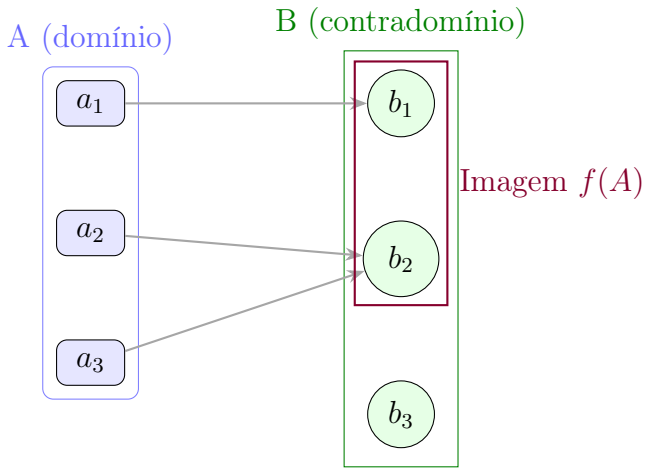


Figura 16: Domínio, contradomínio e imagem: A (retângulos azuis) mapeia via f para B (círculos verdes). A imagem $f(A)$ é o subconjunto de B efetivamente atingido (aqui, $\{b_1, b_2\}$).

- **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.

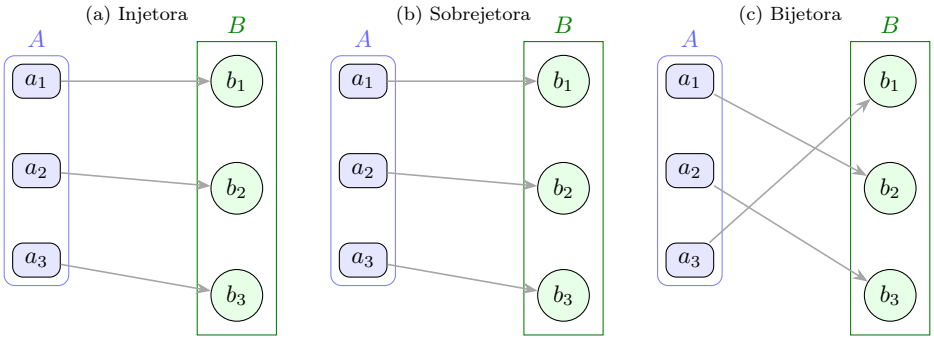


Figura 17: Funções especiais: (a) Injetora — elementos distintos em A têm imagens distintas em B; (b) Sobrejetora — todo elemento de B é imagem; (c) Bijetora — um-para-um e sobre B.

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 dígrafos, 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 dígrafos); 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.

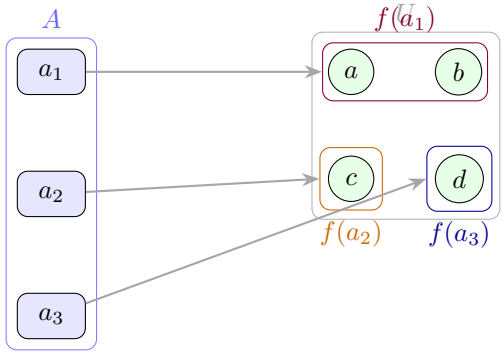


Figura 18: Função c-disjunta: cada $a \in A$ mapeia para um *subconjunto* de B, e imagens de elementos distintos são disjuntas.

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.

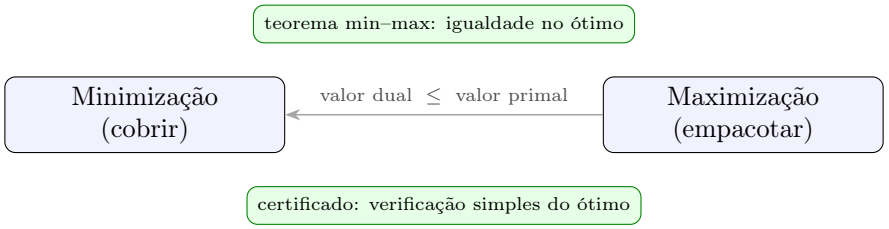


Figura 19: Intuição de min–max: um problema de cobrir (minimização) e um de empacotar (maximização) andam juntos. Sempre vale valor dual $\leq$ valor primal; quando há igualdade, temos um certificado de otimalidade.

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. ([1, 4]).

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. [5, 8])

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.

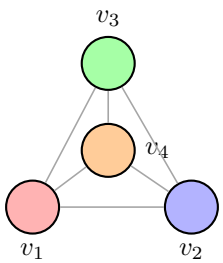


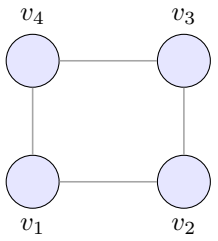
Figura 20: Coloração de grafos: exemplo de coloração própria do grafo completo K_4. Como K_4 é completo, precisamos de 4 cores para colorir seus vértices de modo que vértices adjacentes tenham cores diferentes. Uma coloração é uma função $\varphi : V \to C$ tal que, se $uv \in E$, então $\varphi(u) \neq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

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.

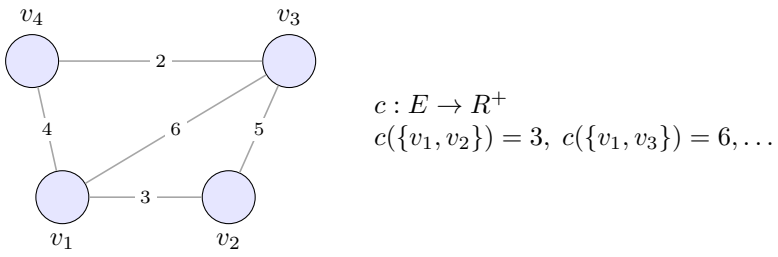


Figura 22: Grafo com custos nas arestas: a função $c : E \to \mathbb{R}^+$ atribui um custo não negativo a cada aresta.

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.

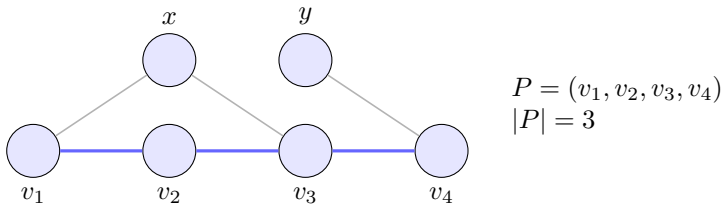


Figura 23: Caminho em grafo não dirigido: o caminho $P = (v_1, v_2, v_3, v_4)$ está destacado em azul. Seu comprimento é o número de arestas percorridas, $|P| = 3$.

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.

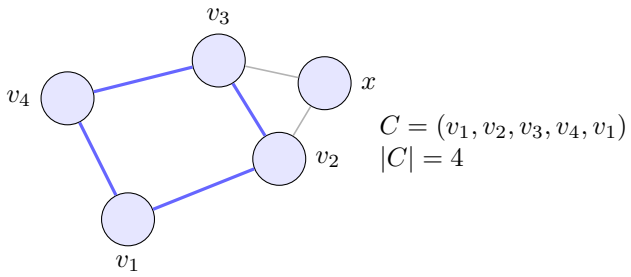


Figura 24: Ciclo em grafo não dirigido: o ciclo $C = (v_1, v_2, v_3, v_4, v_1)$ está destacado em azul. Seu comprimento é o número de arestas, $|C| = 4$.

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.

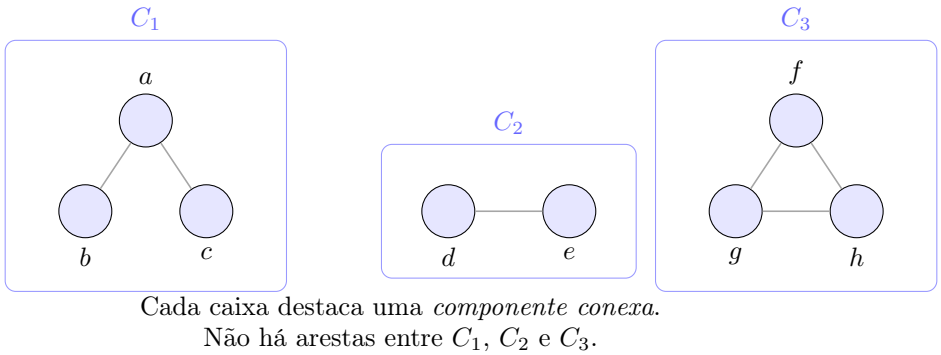


Figura 25: Componentes conexas: o grafo possui três componentes C_1, C_2 e C_3. Cada C_i é conexo e *maximal*, isto é, não pode ser estendido mantendo a conexidade.

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.

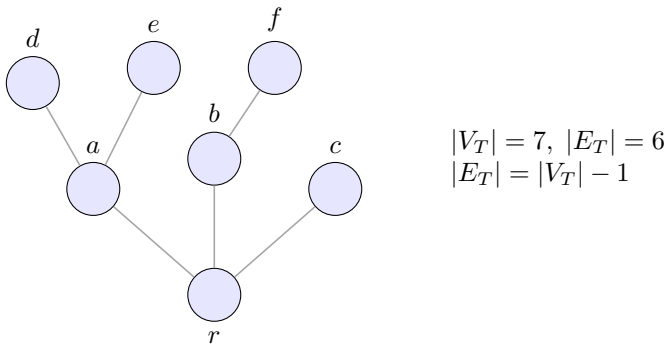


Figura 26: Árvore: grafo conexo e acíclico. No exemplo, $|V_T| = 7$ e $|E_T| = 6$, satisfazendo $|E_T| = |V_T| - 1$. Não há ciclos e existe um único caminho simples entre quaisquer dois vértices.

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

2.5 Dígrafos: 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 dígrafos), onde as arestas são pares de vértices ordenados.

Um **grafo dirigido - dígrafo** (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).

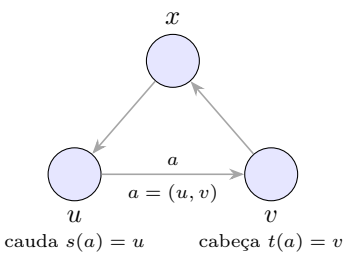


Figura 27: Dígrafos: arcos têm direção. No arco $a = (u, v)$, u é a *cauda* e v é a *cabeça*.

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.

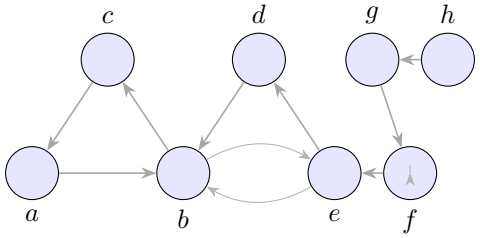
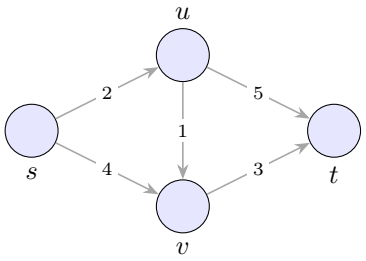


Figura 28: Dígrafo: exemplo de grafo dirigido $D = (V, A)$. Pontos representam os vértices V e setas representam os arcos A, que são pares ordenados de vértices. Laços (como (f, f)) e múltiplos arcos (como (b, e) e (e, b)) são permitidos.

Tal qual os grafos comuns, os dígrafos 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 dígrafo. 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: Dígrafos com custos nos arcos: a função $c : A \to \mathbb{R}^+$ atribui um custo não negativo a cada arco.

Um conceito importante em dígrafos 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 dígrafo é um conjunto $S \subseteq V$ tal que, no subdígrafo 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 subdígrafo 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 dígrafo é um problema NP-difícil, vamos falar sobre o que isso significa na sessão de algoritmos e complexidade.

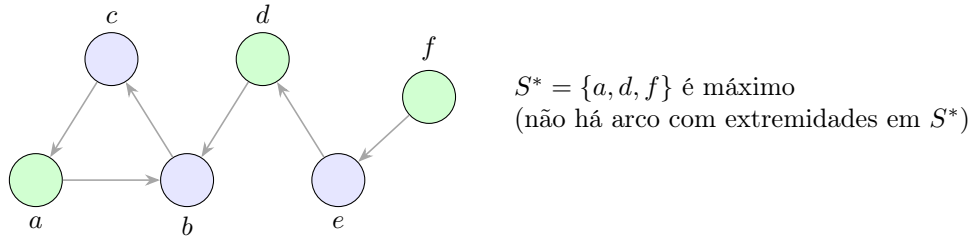


Figura 30: Empacotamento máximo de vértices: para este dígrafo, S^* é um conjunto independente de maior cardinalidade.

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

2.5.1 Subestruturas em dígrafos

Tal qual os grafos que discutimos na sessão anterior, os dígrafos 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 **subdígrafo** $D' = (V', A')$ de um dígrafo $D = (V, A)$ é um dígrafo onde $V' \subseteq V$ e $A' \subseteq A$. Ou seja, D' é formado por um subconjunto dos vértices e arcos de D.

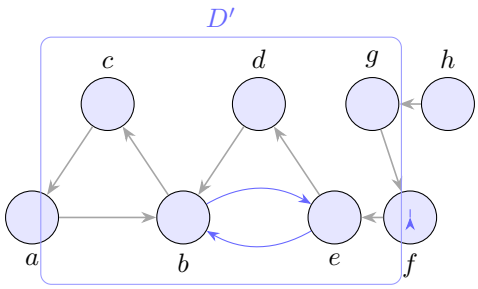


Figura 31: Subdígrafo: o subdígrafo $D' = (V', A')$ está destacado em azul. Aqui, $V' = \{b, c, d, e\}$ e A' contém apenas arcos entre esses vértices.

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, $A' = \{(u, v) \in A \mid u, v \in V'\}$.

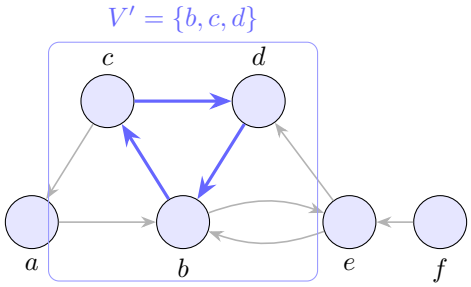


Figura 32: Subdígrafo induzido: para $V' = \{b, c, d\}$, $D[V']$ mantém todos os arcos com ambas as extremidades em V'.

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.

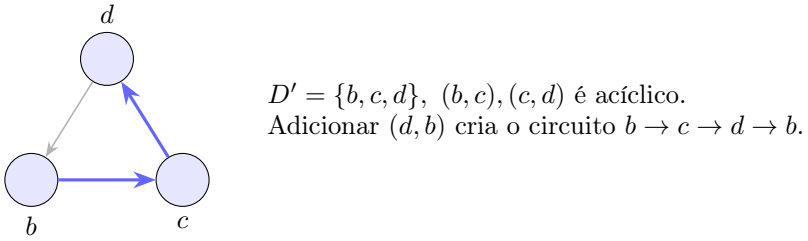


Figura 33: Subdígrafo maximal (por aciclicidade): D' é acíclico e maximal em D; adicionar o arco restante (d, b) cria um circuito.

Subdigrafo Gerador

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

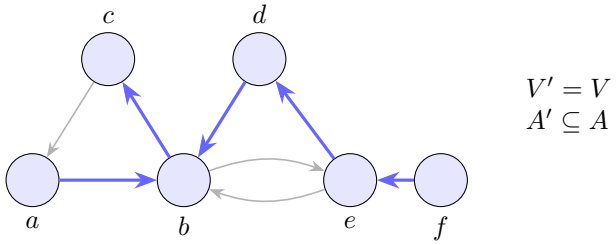


Figura 34: Subdígrafo gerador: inclui todos os vértices do dígrafo original ($V' = V$) e apenas um subconjunto dos arcos (em azul).

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 dígrafo 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.

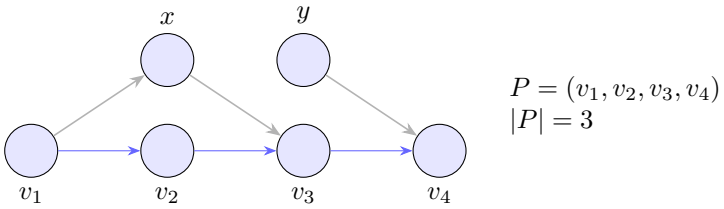


Figura 35: Trilha em dígrafo: a trilha $P = (v_1, v_2, v_3, v_4)$ está destacada em azul. Seu comprimento é o número de arcos percorridos, $|P| = 3$.

Uma conceito importante relacionado às trilhas é o de cortes.

Cortes e Min-cortes

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

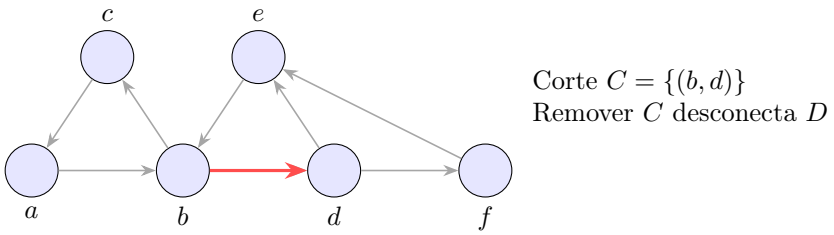


Figura 36: Corte em dígrafo: o corte $C = \{(b, d)\}$ remove conectividade de b para d.

De forma resumida, dado um dígrafo $D = (V, A)$ 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 dígrafo. Formalmente, dado um dígrafo $D = (V, A)$, 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}.

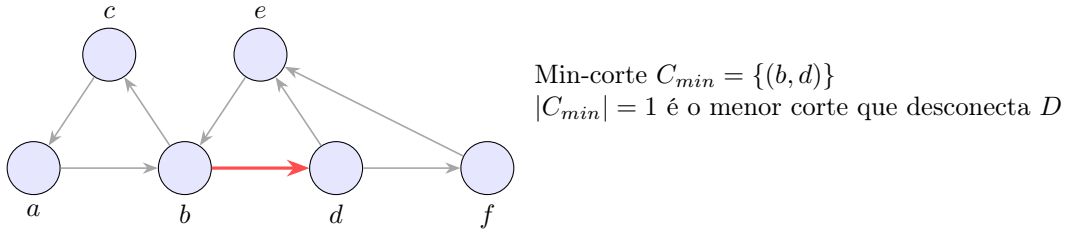


Figura 37: Min-corte em dígrafo: o min-corte C_{min} é um corte de menor cardinalidade (ou custo) que separa s de t.

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 dígrafo 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 dígrafos, relevantes para os algoritmos que discutiremos mais adiante.

Teorema 2.1: Fluxo–Corte Máximo = Mínimo Corte.

Em dígrafos 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, [8].

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.

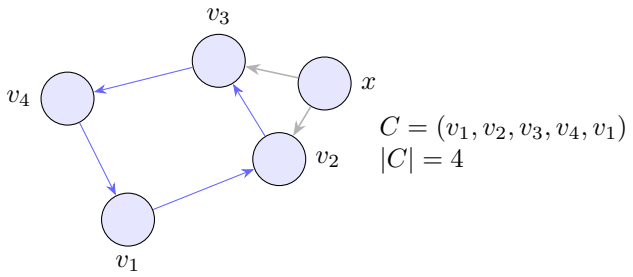


Figura 38: Ciclo direcionado em dígrafo: o ciclo $C = (v_1, v_2, v_3, v_4, v_1)$ está destacado em azul. Seu comprimento é o número de arcos, $|C| = 4$.

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 dígrafo 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 dígrafos, mas com uma nuance importante: em dígrafos, 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 dígrafo tem pelo menos um arco saindo dele (ou entrando nele), então o dígrafo 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 dígrafo 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 dígrafo 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, [8].

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.

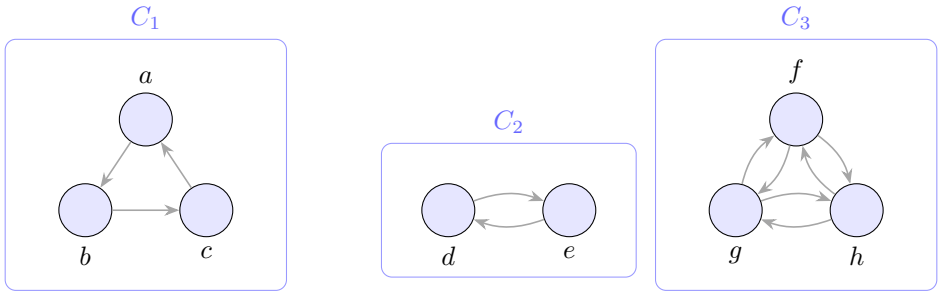
Componentes fortemente conexas

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

Formalmente, uma componente fortemente conexa C é um subdígrafo $C = (V_C, A_C)$ 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 dígrafo 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))[2]:

- **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 dígrafos fortemente conexos:** um dígrafo 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 $\mathrm{Cond}(D)$[3]. 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.

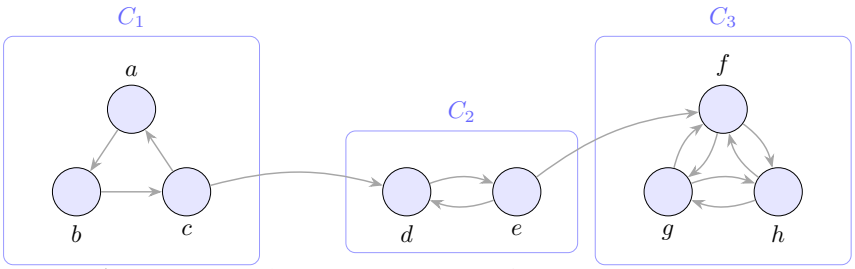
Componentes-fonte

[2]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.

[3]O grafo condensado é uma representação simplificada do dígrafo 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.



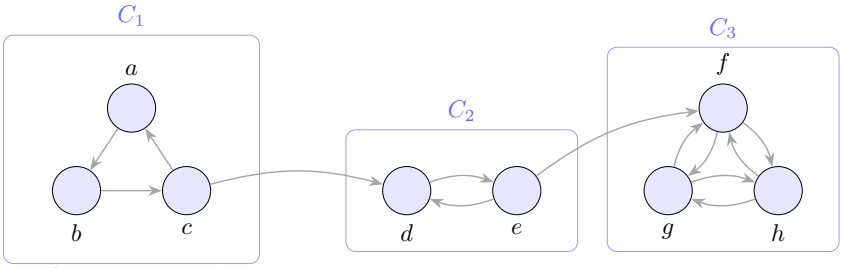
A componente C_1 é uma *componente-fonte* porque não há arcos saindo dela para outras componentes. Já C_2 e C_3 não são componentes-fonte, pois há arcos entrando nelas.

Figura 40: Componente-fonte: o dígrafo 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 dígrafo 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 dígrafo com o número mínimo de arcos necessários para torná-lo fortemente conexo.

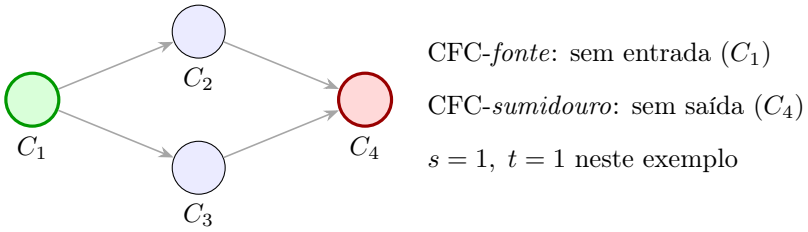


Figura 42: Grafo condensado Cond(D): cada CFC é contraída a um vértice e não há circuitos dirigidos (DAG).

Seja $D = (V, A)$ um dígrafo 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

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

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 dígrafos (e.g., [8]).

Esse lema explica por que todo dígrafo 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 dígrafo 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 dígrafo pode nos ajudar em uma busca otimizada por arborescências.

Arborescências

Uma **arborescência** é um dígrafo 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 dígrafo $T = (V_T, A_T)$ 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.

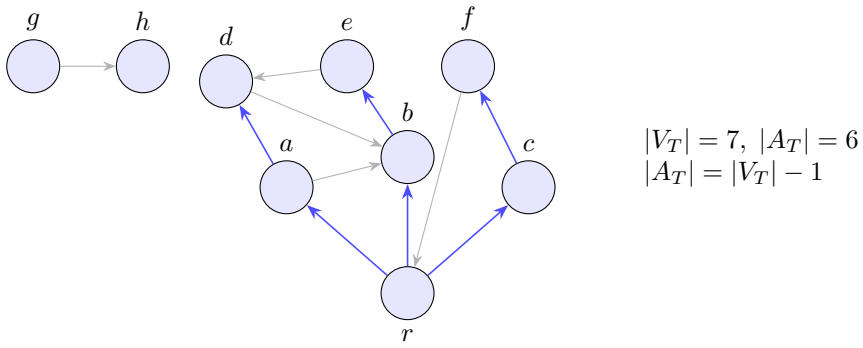


Figura 43: Arborescência: dígrafo conexo e acíclico com raiz r de onde há um caminho direcionado para todos os outros vértices em azul. No exemplo, $|V_T| = 7$ e $|A_T| = 6$, satisfazendo $|A_T| = |V_T| - 1$. Em cinza, arcos que não fazem parte da arborescência.

Definições e notação adicionais

Para facilitar a discussão sobre arborescências, introduzimos algumas definições e notações adicionais: seja $D = (V, A)$ um dígrafo 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 dígrafo original, ou seja, $V_T = V$. Nesse caso, a arborescência é formada por um subconjunto dos arcos do dígrafo 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 dígrafo.

Ramificações Geradoras

Uma **ramificação geradora** é um subdígrafo que é uma arborescência que inclui todos os vértices do dígrafo original. Formalmente, uma ramificação geradora R é um subdígrafo $R = (V_R, A_R)$ 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.

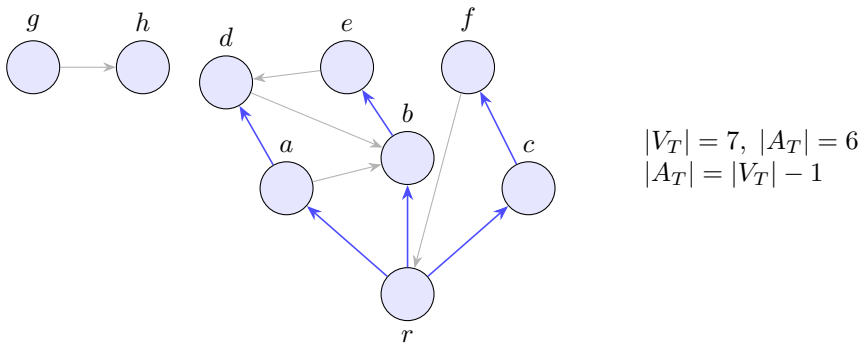


Figura 44: Ramificação geradora: arborescência que inclui todos os vértices do dígrafo original, em azul. No exemplo, $|V_T| = 7$ e $|A_T| = 6$, satisfazendo $|A_T| = |V_T| - 1$. Em cinza, arcos que não fazem parte da ramificação geradora.

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 dígrafo. 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 dígrafo 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.

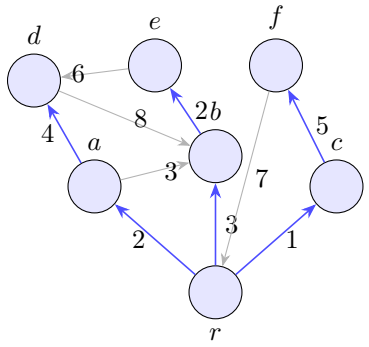
Arborescência de Custo Mínimo:

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 dígrafo $D = (V, A)$, 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.

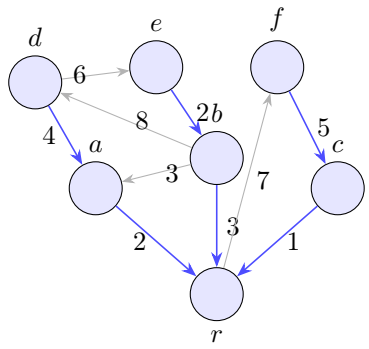


Custo total da r-arborescência: $2 + 3 + 1 + 4 + 2 + 5 = 17$

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 dígrafo $D = (V, A)$, 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 dígrafos 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 subdígrafo que satisfaça (1)–(3) é uma out-arborescência enraizada em r (e análogamente no caso inverso).

Discussões importantes sobre arborescências

Dado um Dígrafo $D = (V, A)$ 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 dígrafos, 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 dígrafo. Uma delas é via a condição de cortes, que estabelece uma relação entre a existência de arborescências e a estrutura dos cortes no dígrafo.

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 dígrafo 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 dígrafo 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, [8].

Outro resultado clássico é o teorema que caracteriza a existência de múltiplas arborescências arcodisjuntas em um dígrafo, 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 dígrafo 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.

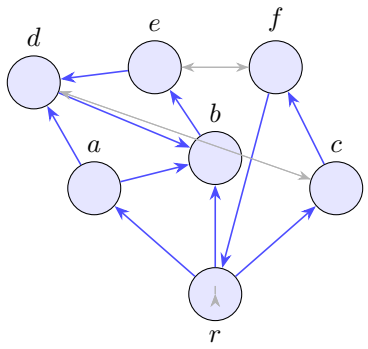


Figura 47: Dígrafo de exemplo para múltiplas arborescências arcodisjuntas.

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) [4], apresentações modernas em [8].

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 dígrafo $D = (V, A)$, 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 dígrafo 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 [5, 8].

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[4]: 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.

[4]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*[5], 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[6]. 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.

[5] 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).

[6] 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[7].

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.

[7]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). [8]

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*.

[8] 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[9] 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.

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

2. **Decisões defitivas**: 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 dígrafos (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 *dígrafos* 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 dígrafo 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**[10] (*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.

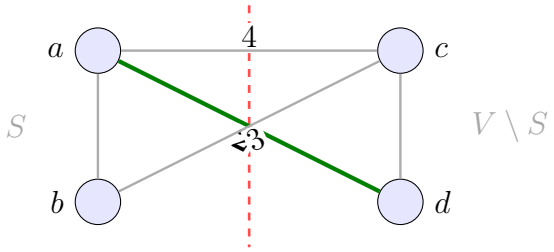


Figura 48: Princípio do corte seguro: entre as arestas que cruzam $(S, V \setminus S)$, a de menor peso (em verde) é *segura* — pode ser incluída em alguma MST sem perder optimalidade.

- **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.

[10]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 $\sum_{e \in T} w(e)$.

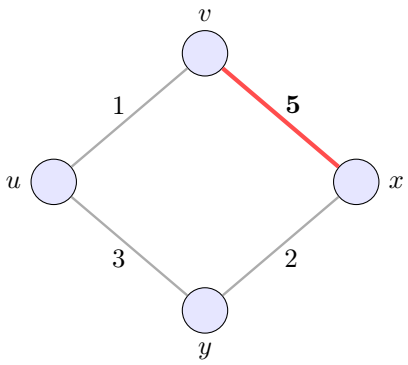


Figura 49: Princípio do ciclo: em qualquer ciclo, a aresta mais pesada (em vermelho) pode ser removida sem desconectar o grafo, reduzindo (ou não aumentando) o custo. Portanto, nenhuma MST contém a aresta mais pesada de um ciclo.

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[11], pode-se obter $O(m + n \log n)$ [2, 7, 9, 3].

Os algoritmos gulosos de MST são corretos em grafos não dirigidos, mas a passagem para dígrafos *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 [8].

3.1.1 A busca em digrafos

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

[11] *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 dígrafo. 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 dígrafos 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

Maquiável 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*[12] para capturar o desempenho agregado.

Recursividade

[12] 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[13].

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 $n = 1$.

[13]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[14] 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 dígrafo 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:

- **Dígrafo**: 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 dígrafo 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 dígrafos 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.

[14] 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, [2] (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 dígrafo.
- **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 dígrafos contraídos e expandiremos as soluções para o dígrafo original.
- **Análise de Complexidade**: avaliaremos a eficiência dos algoritmos em termos de tempo e espaço, garantindo que sejam viáveis para grandes dígrafos.
- **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 dígrafo 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.

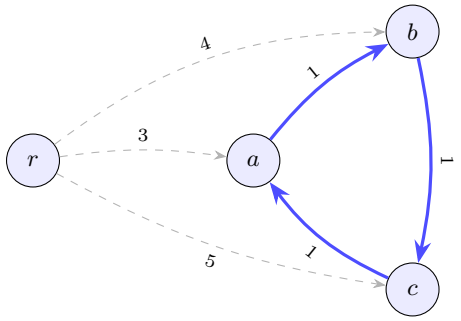


Figura 50: Ciclo gerado pelas escolhas locais "mais baratas por vértice". Os arcos grossos (custo 1) entram em a, b, c e formam $a \to b \to c \to a$. Os arcos tracejados partindo de r existem, mas são mais caros e por isso não são escolhidos pelo critério local.

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);
- 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) que entra em $w \in C$ passa a (u, x_C) com custo reduzido $c'(u, x_C) = c(u, w) - c(a_w)$, onde a_w é o arco mais barato 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.

Visão Operacional - Chu–Liu/Edmonds

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 [7, 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 após contração e no grafo original. (a) No grafo contraído, toda arborescência seleciona exatamente um arco que entra no supervértice x_C. (b) Ao expandir C, escolhe-se o arco correspondente (u, w) que entra em algum $w \in C$ e mantêm-se os arcos internos de custo reduzido zero, removendo exatamente um para quebrar o ciclo. Como $c'(u, x_C) = c(u, w) - c(a_w)$ e os arcos internos têm custo reduzido zero, o custo total é preservado na ida e na volta.

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 [8, 7].

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 pelo ciclo 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, pois 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: dígrafo $D = (V, A)$, 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 [7, 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 dígrafo 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 [7, 8, Sec. 4.9].

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

4.1.1 Corretude

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

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

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

pois há exatamente um arco de T entrando em cada $v \neq r$. O termo $\sum_{v \neq r} y(v)$ é constante (independe de T); assim, minimizar $\sum c$ equivale a minimizar $\sum c'$ [7, 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 [7, 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 [7, 8, 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.

4.1.2 Complexidade

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

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

4.2 Implementação em Python

Esta seção apresenta uma implementação em Python do algoritmo de Chu–Liu/Edmonds, baseada no código do trabalho. A arquitetura segue os passos teóricos:

- seleção dos arcos de menor custo que entram em cada vértice (formação de F^*);
- verificação de aciclicidade (se acíclico, retorno imediato);
- se houver ciclo, normalização dos custos por vértice (custos reduzidos);
- construção do grafo funcional F^* com arcos de custo reduzido zero;
- detecção de ciclo em F^*;
- contração do ciclo em um supervértice, com ajuste de custos;
- recursão no grafo contraído;
- expansão do ciclo e remoção de um arco interno para quebrar o ciclo.
- retorno da r-arborescência ótima.

Especificação da interface (entradas, saídas e hipóteses)

- **Entrada:** dígrafo 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 [8]).
  - 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[15] 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*[16], 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 $O(\deg^-(v))$.

Função 4.1: Normalização por vértice: custos reduzidos

```
1  def normalize_incoming_edge_weights(D: nx.DiGraph, node: str,
       lang="pt"):
2      """
3      Altera os pesos das arestas que entram em ‘node‘
4      subtraindo de cada uma o menor peso de entrada no grafo D
           .
5
6      Parâmetros:
7          - D: Um grafo direcionado (networkx.DiGraph)
8          - node: O vértice alvo cujas arestas de entrada serão
               ajustadas
9          - lang: Idioma das mensagens de erro ("en" para inglê
               s, "pt" para português)
10
11     Retorno:
12         - Nada (o grafo D é modificado no próprio lugar - in-
               place)
13     """
14
15     if lang == "en":
```

[15] 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.

[16] 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.

```
16          assert (
17              node in D
18          ), f"\nnormalize_incoming_edge_weights: The vertex '{
                node}' does not exist in the graph."
19      elif lang == "pt":
20          assert (
21              node in D
22          ), f"\nnormalize_incoming_edge_weights: O vértice '{
                node}' não existe no grafo."
23
24      # Get the incoming edges of the node with their weights
25      predecessors = list(D.in_edges(node, data="w"))
26
27      if not predecessors:
28          return
29
30      # Calculate the minimum weight among the incoming edges
31      yv = min((w for _, _, w in predecessors))
32
33      # Subtract Yv from each incoming edge
34      for u, _, _ in predecessors:
35          D[u][node]["w"] -= yv
```

Construção de F^*: a função constrói o subdígrafo 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 dígrafo 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 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^*.

Função 4.2: Construção de F^*

```
1 def get_Fstar(D: nx.DiGraph, r0: str, lang="pt"):
2     """
3     Constrói o subgrafo funcional F* a partir do grafo D e da
          raiz r0.
4     Retorna um grafo direcionado F_star com exatamente uma
         aresta de entrada
5     para cada v != r0 e nenhuma aresta entrando em r0.
```

```
6
7       Parâmetros:
8           - D: Um grafo direcionado (networkx.DiGraph)
9           - r0: Vértice raiz (rótulo)
10          - lang: Idioma das mensagens de erro ("en" para inglê
                s, "pt" para português)
11
12      Retorno:
13          - F_star: Um grafo direcionado (networkx.DiGraph) que
                representa F*
14      """
15
16      if lang == "en":
17          assert (
18              r0 in D
19          ), f"\nget_Fstar: The root vertex '{r0}' does not
                exist in the graph."
20      elif lang == "pt":
21          assert r0 in D, f"\nget_Fstar: O vértice raiz '{r0}'
                não existe no grafo."
22
23      # Create an empty directed graph for F_star
24      F_star = nx.DiGraph()
25
26      for v in D.nodes():
27          if v != r0:
28              in_edges = list(D.in_edges(v, data="w"))
29              if not in_edges:
30                  continue  # No edges entering v
31              u = next((u for u, _, w in in_edges if w == 0),
                    None)
32              if u:
33                  F_star.add_edge(u, v, w=0)
34      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.

Função 4.3: Detecção de ciclo dirigido em F^*

```python
def find_cycle(F_star: nx.DiGraph):
    """
    Encontra um ciclo dirigido no grafo.
    Retorna um subgrafo contendo o ciclo, ou None caso não exista.

    Parâmetros:
        - F_star: Um grafo direcionado (networkx.DiGraph)

    Retorno:
        - Um grafo direcionado (networkx.DiGraph) representando o ciclo, ou None se nenhum ciclo for encontrado.
    """

    try:
        nodes_in_cycle = set()
        # Extract nodes involved in the cycle
        for u, v, _ in nx.find_cycle(F_star, orientation="original"):
            nodes_in_cycle.update([u, v])
        # Create a subgraph containing only the cycle
        return F_star.subgraph(nodes_in_cycle).copy()

    except nx.NetworkXNoCycle:
        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, a rotina:

- **Pré-condições:**

  (i) C induz um ciclo dirigido em D;

  (ii) a raiz r_0 não pertence a C;

  (iii) o rótulo `label` não existe como vértice em D (é o nome de x_C).

- **Ajuste de custos (entradas em C):** para cada arco (u, w) com $u \notin C$, $w \in C$, cria (u, x_C) com custo $c'(u, x_C) := c(u, w) - y(w) = c(u, w) - c(a_w)$, onde a_w é

a entrada mais barata em w. Mantém-se, para cada u, apenas o mínimo entre paralelos (u, x_C).

- **Arcos que saem de C:** para cada arco (w, v) com $w \in C$, $v \notin C$, cria (x_C, v) com o *mesmo* custo $c(w, v)$; se surgirem paralelos, conserva-se o menor.
- **Laços e arcos internos:** arcos com ambas as pontas em C tornam-se laços em x_C e são descartados (não influenciam a solução).
- **Mapeamentos para reexpansão:** devolve dicionários
  - `in_to_cycle`$[u] = (w^*, c'(u, x_C))$: o vértice $w^* \in C$ que recebeu, de u, a entrada mínima ao contrair (após o ajuste), e o custo reduzido correspondente; usado para substituir (u, x_C) por (u, w^*) na expansão e remover a_{w^*} de C.
  - `out_from_cycle`$[v] = (w^*, c(w^*, v))$: o vértice $w^* \in C$ que envia a v o arco de menor custo antes da contração; útil para reconstruir paralelos e decidir empates.
- **Efeitos colaterais:** remove todos os vértices de C de D, insere x_C e adiciona os arcos redirecionados com os custos apropriados.
- **Complexidade:** $O(m)$ no tamanho do grafo atual: cada aresta incidente a C é examinada no máximo uma vez; a manutenção de mínimos entre paralelos é feita em tempo total linear.
- **Casos de borda:** (i) se não houver arco entrando em C, x_C ficará sem entradas, impossibilitando uma r-arborescência (instância inviável);

  (ii) se não houver arco saindo de C, x_C ficará sem saídas (permitido, mas pode isolar componentes);

  (iii) pesos negativos são suportados — a normalização por custos reduzidos preserva comparações relativas.

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[17] indica que o grafo D é modificado diretamente.

Função 4.4: Contração de ciclo

```
1 def contract_cycle(D: nx.DiGraph, C: nx.DiGraph, label: str,
      lang="pt"):
2     """
```

[17] "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.

```
3       Contrai um ciclo C no grafo D, substituindo-o por um
           supervértice rotulado `label`.
4       Modifica o grafo D no próprio lugar (in-place) e devolve
           dicionários auxiliares para a reexpansão.
5
6       Parâmetros:
7           - D: Um grafo direcionado (networkx.DiGraph)
8           - C: Um grafo direcionado (networkx.DiGraph) que
               representa o ciclo a ser contraído
9           - label: O rótulo do novo supervértice
10          - lang: Idioma das mensagens de erro ("en" para inglê
               s, "pt" para português)
11
12      Retorno:
13          - in_to_cycle: Dicionário que mapeia nós fora do
               ciclo para tuplas (no_do_ciclo, peso) das arestas
               que entram no ciclo após o ajuste
14          - out_from_cycle: Dicionário que mapeia nós fora do
               ciclo para tuplas (no_do_ciclo, peso) das arestas
               que saem do ciclo
15      """
16
17      if lang == "en":
18          assert (
19              label not in D
20          ), f"\ncontract_cycle: The label '{label}' already
               exists as a vertex in G."
21      elif lang == "pt":
22          assert (
23              label not in D
24          ), f"\ncontract_cycle: O rótulo '{label}' já existe
               como vértice em G."
25
26      cycle_nodes: set[str] = set(C.nodes())
27
28      # Stores the vertex u outside the cycle and the vertex v
           inside the cycle that receives the minimum weight edge
29      in_to_cycle: dict[str, tuple[str, float]] = {}
30
31      for u in D.nodes:
32          if u not in cycle_nodes:
33              # Find the minimum weight edge that u has to any
                   vertex in C
34              min_weight_edge_to_cycle = min(
35                  ((v, w) for _, v, w in D.out_edges(u, data="w
                       ") if v in cycle_nodes),
36                  key=lambda x: x[1],
37                  default=None,
38              )
```

```
39              if min_weight_edge_to_cycle:
40                  in_to_cycle[u] = min_weight_edge_to_cycle
41
42      for u, (v, w) in in_to_cycle.items():
43          D.add_edge(u, label, w=w)
44
45      # Stores the vertex v outside the cycle that receives the
            minimum weight edge from a vertex u inside the cycle
46      out_from_cycle: dict[str, tuple[str, float]] = {}
47
48      for v in D.nodes:
49          if v not in cycle_nodes:
50              # Find the minimum weight edge that v receives
                    from any vertex in C
51              min_weight_edge_from_cycle = min(
52                  ((u, w) for u, _, w in D.in_edges(v, data="w"
                        ) if u in cycle_nodes),
53                  key=lambda x: x[1],
54                  default=None,
55              )
56              if min_weight_edge_from_cycle:
57                  out_from_cycle[v] =
                        min_weight_edge_from_cycle
58
59      for v, (u, w) in out_from_cycle.items():
60          D.add_edge(label, v, w=w)
61
62      # Remove all nodes in the cycle from G
63      D.remove_nodes_from(cycle_nodes)
64
65      return in_to_cycle, out_from_cycle
```

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.

Função 4.5: Remoção de arestas que entram na raiz

```python
def remove_edges_to_r0(
    D: nx.DiGraph, r0: str, log=None, boilerplate: bool = True, lang="pt"
):
    """
    Remove todas as arestas que entram no vértice raiz r0 no grafo D.
    Retorna o grafo atualizado.

    Parâmetros:
        - D: Um grafo direcionado (networkx.DiGraph)
        - r0: O vértice raiz
        - log: Função opcional de logging para registrar informações
        - boilerplate: Se True, habilita mensagens de logging
        - lang: Idioma das mensagens de logging ("en" para inglês, "pt" para português)

    Retorno:
        - D: O grafo direcionado (networkx.DiGraph) atualizado, sem as arestas que entram em r0
    """

    # Verify that r0 exists in G
    if lang == "en":
        assert (
            r0 in D
        ), f"\nremove_edges_to_r0: The root vertex '{r0}' does not exist in the graph."
    elif lang == "pt":
        assert (
            r0 in D
        ), f"\nremove_edges_to_r0: O vértice raiz '{r0}' não existe no grafo."

    # Remove all edges entering r0
    in_edges = list(D.in_edges(r0))
    if not in_edges:
        if boilerplate and log:
            if lang == "en":
                log(f"\nremove_edges_to_r0: No edges entering '{r0}' to remove.")
            elif lang == "pt":
                log(
                    f"\nremove_edges_to_r0: Nenhuma aresta entrando em '{r0}' para remover."
                )
```

```
    else:
        D.remove_edges_from(in_edges)
        if boilerplate and log:
            if lang == "en":
                log(
                    f"\nremove_edges_to_r0: Removed {len(
                        in_edges)} edges entering '{r0}'."
                )
            elif lang == "pt":
                log(
                    f"\nremove_edges_to_r0: Removidas {len(
                        in_edges)} arestas entrando em '{r0}'.
                        "
                )
    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 $O(\deg^-(v))$.

Função 4.6: Remover arco interno na reexpansão

```
def remove_internal_edge_to_cycle_entry(C: nx.DiGraph, v):
    """
    Remove a aresta interna que entra no vértice de entrada `
       v` do ciclo C,
    pois `v` passa a receber uma aresta externa do grafo.

    Parâmetros:
        - C: subgrafo do ciclo (networkx.DiGraph)
        - v: vértice de entrada no ciclo que receberá a
           aresta externa

    Retorno:
        - Nada (o subgrafo do ciclo é modificado no próprio
           lugar - in-place)
    """

    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:

(i) prepara a instância (opcionalmente removendo entradas em r_0 e emitindo logs);

(ii) normaliza, para cada $v \neq r_0$, os custos das arestas que *entram* em v para induzir pelo menos uma entrada de custo reduzido zero;

(iii) constrói o grafo funcional F^* selecionando, para cada $v \neq r_0$, uma única entrada de menor custo reduzido (preferencialmente zero);

(iv) verifica aciclicidade de F^*; se acíclico, devolve F^* como r-arborescência;

(v) caso haja ciclo, contrai o ciclo em um supervértice, ajusta custos que *entram* no componente contraído e chama-se recursivamente na instância reduzida; ao retornar, expande o componente e remove exatamente uma aresta interna do ciclo para restaurar grau de entrada igual a 1 e aciclicidade.

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

- **Função (entradas/saídas):** Entrada: dígrafo ponderado $D = (V, A)$, raiz r_0, e, opcionalmente, funções `draw_fn` e `log` para visualização e registro. Saída: um subdígrafo 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):** Se fornecidos, `log` recebe mensagens estruturadas por nível de recursão, e `draw_fn` 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.

Função 4.7: Procedimento principal (recursivo)

```
def find_optimum_arborescence_chuliu(
    D: nx.DiGraph,
    r0: str,
    level=0,
    draw_fn=None,
    log=None,
    boilerplate: bool = True,
    lang="pt",
):
    """
    Encontra a arborescência ótima em um dígrafo D com raiz
       r0 usando o algoritmo de Chu-Liu/Edmonds.

    Parâmetros:
        - D: Um grafo direcionado (networkx.DiGraph)
        - r0: O vértice raiz
        - level: Nível atual da recursão (usado para logs e
           visualização)
        - draw_fn: Função opcional para visualizar o grafo em
            cada etapa
        - log: Função opcional de logging para registrar
           informações
        - boilerplate: Se True, habilita logging e visualizaç
           ão
        - lang: Idioma das mensagens de logging ("en" para
           inglês, "pt" para português)

    Retorno:
        - Um grafo direcionado (networkx.DiGraph) que
           representa a arborescência ótima

    Exceções:
        - AssertionError: Se o vértice raiz r0 não estiver
           presente no grafo D
        - AssertionError: Se nenhum ciclo for encontrado em
           F_star quando esperado
        - AssertionError: Se o rótulo contraído já existir no
            grafo D
        - AssertionError: Se não houver aresta de entrada
```

```
            para o vértice contraído em F_prime
30          - AssertionError: Se nenhum vértice do ciclo for
               encontrado para receber a aresta de entrada
31          - AssertionError: Se o rótulo contraído não for
               encontrado em F_prime
32          - AssertionError: Se os vértices u ou v não forem
               encontrados no grafo original D
33      """
34
35      indent = "  " * level
36
37      if boilerplate and log:
38          if lang == "en":
39              log(f"\nfind_optimum_arborescence_chuliu:{indent}
                   Starting level {level}")
40          elif lang == "pt":
41              log(f"\nfind_optimum_arborescence_chuliu:{indent}
                   Iniciando nível {level}")
42
43      if lang == "en":
44          assert (
45              r0 in D
46          ), f"\nfind_optimum_arborescence_chuliu: The root
               vertex '{r0}' is not present in the graph."
47      elif lang == "pt":
48          assert (
49              r0 in D
50          ), f"\nfind_optimum_arborescence_chuliu: O vértice
               raiz '{r0}' não está presente no grafo."
51
52      D_copy = D.copy()
53
54      if boilerplate and log:
55          if lang == "en":
56              log(
57                  f"\nfind_optimum_arborescence_chuliu:{indent}
                       Removing edges entering '{r0}'"
58              )
59          elif lang == "pt":
60              log(
61                  f"\nfind_optimum_arborescence_chuliu:{indent}
                       Removendo arestas que entram em '{r0}'"
62              )
63          if draw_fn:
64              if lang == "en":
65                  draw_fn(
66                      D_copy,
67                      f"\nfind_optimum_arborescence_chuliu:{
                           indent}After removing incoming edges",
```

```
68                 )
69             elif lang == "pt":
70                 draw_fn(
71                     D_copy,
72                     f"\nfind_optimum_arborescence_chuliu:{
                        indent}Após remoção de entradas",
73                 )
74
75     for v in D_copy.nodes:
76         if v != r0:
77             normalize_incoming_edge_weights(D_copy, v, lang=
                    lang)
78
79         if boilerplate and log:
80             if lang == "en":
81                 log(
82                     f"\nfind_optimum_arborescence_chuliu:{
                        indent}Normalizing weights of incoming
                         edges to '{v}'"
83                 )
84             elif lang == "pt":
85                 log(
86                     f"\nfind_optimum_arborescence_chuliu:{
                        indent}Normalizando pesos de arestas
                        de entrada para '{v}'"
87                 )
88             if draw_fn:
89                 if lang == "en":
90                     draw_fn(
91                         D_copy,
92                         f"\nfind_optimum_arborescence_chuliu
                            :{indent}After weight adjustment",
93                     )
94                 elif lang == "pt":
95                     draw_fn(
96                         D_copy,
97                         f"\nfind_optimum_arborescence_chuliu
                            :{indent}Após ajuste de pesos",
98                     )
99
100    # Build F_star
101    F_star = get_Fstar(D_copy, r0, lang=lang)
102
103    if boilerplate and log:
104        if lang == "en":
105            log(f"\nfind_optimum_arborescence_chuliu:{indent}
                    Building F_star")
106        elif lang == "pt":
107            log(f"\nfind_optimum_arborescence_chuliu:{indent}
```

```
                Construindo F_star")
108         if draw_fn:
109             if lang == "en":
110                 draw_fn(F_star, f"\
                       nfind_optimum_arborescence_chuliu:{indent}
                       F_star")
111             elif lang == "pt":
112                 draw_fn(F_star, f"\
                       nfind_optimum_arborescence_chuliu:{indent}
                       F_star")
113
114     if nx.is_arborescence(F_star):
115         for u, v in F_star.edges:
116             F_star[u][v]["w"] = D[u][v]["w"]
117         return F_star
118
119     else:
120         if boilerplate and log:
121             if lang == "en":
122                 log(
123                     f"\nfind_optimum_arborescence_chuliu:{
                           indent}F_star is not an arborescence.
                           Continuing..."
124                 )
125             elif lang == "pt":
126                 log(
127                     f"\nfind_optimum_arborescence_chuliu:{
                           indent}F_star não é uma arborescência.
                            Continuando..."
128                 )
129
130         C: nx.DiGraph = find_cycle(F_star)
131
132         if lang == "en":
133             assert C, f"\nfind_optimum_arborescence_chuliu:
                   No cycle found in F_star."
134         elif lang == "pt":
135             assert (
136                 C
137             ), f"\nfind_optimum_arborescence_chuliu: Nenhum
                   ciclo encontrado em F_star."
138
139         contracted_label = f"\n n*{level}"
140         in_to_cycle, out_from_cycle = contract_cycle(
141             D_copy, C, contracted_label, lang=lang
142         )
143
144         # Recursive call
145         F_prime = find_optimum_arborescence_chuliu(
```

```
146             D_copy,
147             r0,
148             level + 1,
149             draw_fn=None,
150             log=None,
151             boilerplate=boilerplate,
152             lang=lang,
153         )
154
155         # Identify the vertex in the cycle that received the
               only incoming edge from the arborescence
156         in_edge = next(iter(F_prime.in_edges(contracted_label
               , data="w")), None)
157
158         if lang == "en":
159             assert (
160                 in_edge
161             ), f"\nfind_optimum_arborescence_chuliu: No
                   incoming edge found for vertex '{
                   contracted_label}'."
162         elif lang == "pt":
163             assert (
164                 in_edge
165             ), f"\nfind_optimum_arborescence_chuliu: Nenhuma
                   aresta encontrada entrando no vértice '{
                   contracted_label}'."
166
167         u, _, _ = in_edge
168
169         v, _ = in_to_cycle[u]
170
171         if lang == "en":
172             assert (
173                 v is not None
174             ), f"\nfind_optimum_arborescence_chuliu: No
                   vertex in the cycle found to receive the
                   incoming edge from '{u}'."
175         elif lang == "pt":
176             assert (
177                 v is not None
178             ), f"\nfind_optimum_arborescence_chuliu: Nenhum v
                   értice do ciclo encontrado que recebeu a
                   aresta de entrada de '{u}'."
179
180         # Remove the internal edge entering vertex `v` from
               cycle C
181         remove_internal_edge_to_cycle_entry(
182             C, v
183         )  # Note: w is coming from F_prime, not from G
```

```
184
185         # Add the external edge entering the cycle (
                identified by in_edge), the weight will be
                corrected at the end using G
186         F_prime.add_edge(u, v)
187         if boilerplate and log:
188             if lang == "en":
189                 log(
190                     f"\nfind_optimum_arborescence_chuliu:{
                            indent}Adding incoming edge to cycle:
                            ({u}, {v})"
191                 )
192             elif lang == "pt":
193                 log(
194                     f"\nfind_optimum_arborescence_chuliu:{
                            indent}Adicionando aresta de entrada
                            ao ciclo: ({u}, {v})"
195                 )
196
197         # Add the remaining edges of the modified cycle C
198         for u_c, v_c in C.edges:
199             F_prime.add_edge(u_c, v_c)
200             if boilerplate and log:
201                 if lang == "en":
202                     log(
203                         f"\nfind_optimum_arborescence_chuliu
                                :{indent}Adding cycle edge: ({u_c
                                }, {v_c})"
204                     )
205                 elif lang == "pt":
206                     log(
207                         f"\nfind_optimum_arborescence_chuliu
                                :{indent}Adicionando aresta do
                                ciclo: ({u_c}, {v_c})"
208                     )
209
210         # Add the external edges leaving the cycle
211         for _, z, _ in F_prime.out_edges(contracted_label,
                data=True):
212
213             if lang == "en":
214                 assert (
215                     z in out_from_cycle
216                 ), f"\nfind_optimum_arborescence_chuliu: No
                        outgoing edge found for vertex '{z}'."
217             elif lang == "pt":
218                 assert (
219                     z in out_from_cycle
220                 ), f"\nfind_optimum_arborescence_chuliu:
```

```
                    Nenhuma aresta de saída encontrada para o
                    vértice '{z}'."
221
222             u_cycle, _ = out_from_cycle[z]
223             F_prime.add_edge(u_cycle, z)
224
225             if boilerplate and log:
226                 if lang == "en":
227                     log(
228                         f"\nfind_optimum_arborescence_chuliu
                            :{indent}Adding outgoing edge from
                             cycle: ({u_cycle}, {z})"
229                     )
230                 elif lang == "pt":
231                     log(
232                         f"\nfind_optimum_arborescence_chuliu
                            :{indent}Adicionando aresta
                            externa de saída: ({u_cycle}, {z})
                            "
233                     )
234
235         # Remove the contracted node
236         if lang == "en":
237             assert (
238                 contracted_label in F_prime
239             ), f"\nfind_optimum_arborescence_chuliu: Vertex
                   '{contracted_label}' not found in the graph."
240         elif lang == "pt":
241             assert (
242                 contracted_label in F_prime
243             ), f"\nfind_optimum_arborescence_chuliu: Vértice
                   '{contracted_label}' não encontrado no grafo."
244         F_prime.remove_node(contracted_label)
245
246         if boilerplate and log:
247             if lang == "en":
248                 log(
249                     f"\nfind_optimum_arborescence_chuliu:{
                        indent}Contracted vertex '{
                        contracted_label}' removed."
250                 )
251             elif lang == "pt":
252                 log(
253                     f"\nfind_optimum_arborescence_chuliu:{
                        indent}Vértice contraído '{
                        contracted_label}' removido."
254                 )
255
256         # Update the edge weights with the original weights
```

```
            from G
257         for u, v in F_prime.edges:
258             if lang == "en":
259                 assert (
260                     u in D and v in D
261                 ), f"\nfind_optimum_arborescence_chuliu:
                       Vertex '{u}' or '{v}' not found in the
                       original graph."
262             elif lang == "pt":
263                 assert (
264                     u in D and v in D
265                 ), f"\nfind_optimum_arborescence_chuliu: Vé
                       rtice '{u}' ou '{v}' não encontrado no
                       grafo original."
266             F_prime[u][v]["w"] = D[u][v]["w"]
267
268         if boilerplate and log:
269             if lang == "en":
270                 log(
271                     f"\n✓{indent}Final arborescence: {list(
                           F_prime.edges)}"
272                 )
273             elif lang == "pt":
274                 log(
275                     f"\n✓{indent}Arborescência final: {list(
                           F_prime.edges)}"
276                 )
277             if draw_fn:
278                 if lang == "en":
279                     draw_fn(
280                         F_prime,
281                         f"\n{indent}Final Arborescence.",
282                     )
283                 elif lang == "pt":
284                     draw_fn(
285                         F_prime,
286                         f"\n{indent}Arborescência final.",
287                     )
288         return F_prime
```

Notas finais

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.

Para fechar este capítulo, vale destacar algumas decisões de projeto e implicações práticas:

- **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.

No capítulo seguinte revisitaremos o método sob uma *ótica primal–dual* em duas fases, proposta por András Frank. Essa perspectiva organiza a normalização via *potenciais*[18] $y(\cdot)$, 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

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*.

[18] 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 [6]. 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 [6, 5, 8].

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 $c^\top x$[19] 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.

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.

[19] 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.

5.1.2 Laminaridade, potenciais e contrações

Com variáveis duais $y(X)$ associadas a cortes, podemos, por um argumento de *uncrossing*[20], 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 [5, 8].

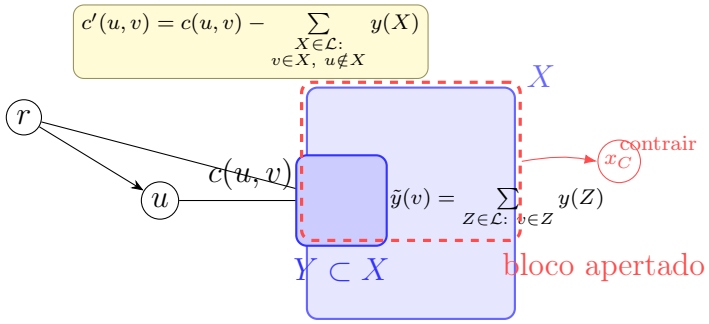


Figura 54: Família laminar de cortes ($Y \subset X$), potenciais por vértice ($\tilde{y}$), custo reduzido em uma aresta e a ideia de contrair um bloco *apertado*.

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.

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

[20] "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 [8, 5].

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 [5, 8].

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 [5].

Algoritmo 5.1: Frank: fase I

Entrada: dígrafo $D = (V, A)$, 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 [5, 8].

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 [8, 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 [5, 8]. □

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 $O(n)$ [5, 8].

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 dígrafo 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 dí
          grafo 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: dígrafo (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 dígrafo D. Ela subtrai um valor mínimo dos pesos dessas arestas e registra aquelas que atingem peso zero em uma lista e em um novo dígrafo. Essa função tem efeito colateral, pois modifica o dígrafo original.

Função 5.3: Atualizar pesos em X

```
1  def update_weights_in_X(D, arcs, min_weight, A_zero, D_zero):
2      """
3      Update the weights of the arcs in a directed graph D for
           the nodes in set X.
```

```
4       ATTENTION: The function produces collateral effect in the
            provided directed graph by updating its arcs weights.
5
6       Parameters:
7           - D: directed graph (DiGraph)
8           - arcs: list of tuples (u, v, data) where u not in X
                and v in X
9           - min_weight: minimum weight to be subtracted from
                the arcs weights
10          - A_zero: list to store the arcs that reach weight
                zero
11          - D_zero: directed graph (DiGraph) to store the arcs
                that reach weight zero
12
13      Returns:
14          - None
15      """
16
17      for u, v, _ in arcs:
18          D[u][v]["w"] -= min_weight
19          if D[u][v]["w"] == 0:
20              A_zero.append((u, v))
21              D_zero.add_edge(u, v)
```

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

Função 5.4: Identificar Arborescência

```
1  def has_arborescence(D, r0):
2      """
3      Check if a directed graph D has an arborescence with root
           r0.
4      The function returns True if an arborescence exists,
          otherwise False.
5
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 $C = \text{Cond}(D\_0)$ via `nx.condensation(D_zero)` e coleta as *fontes* de C (componentes com grau de entrada zero)[21];
- *Fontes:* se restar apenas uma fonte (a que contém $r0$), termina;
- *Processamento das fontes:* para cada fonte u com $X = \text{members}(u)$ 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.

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.
```

[21] 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.

```
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
```

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)

```python
def phase2_find_minimum_arborescence(
    D_original, r0, A_zero, draw_fn=None, log=None,
       boilerplate: bool = True, lang="pt"
):
    """
    Find the minimum arborescence in a directed graph D with
       root r0.
    The function returns the minimum arborescence as a
       DiGraph.

    Parameters:
        - D_original: directed graph (DiGraph)
        - r0: root node
        - A_zero: list of arcs (u, v) that form the minimum
           arborescence

    Returns:
        - Arb: directed graph (DiGraph) representing the
           minimum arborescence
    """
    Arb = nx.DiGraph()

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

    # While there are arcs to be considered
    for _ in range(n - 1):
        for u, v in A_zero:
            if u in Arb.nodes() and v not in Arb.nodes():
                edge_data = D_original.get_edge_data(u, v)
                Arb.add_edge(u, v, **edge_data)
                # Restart the loop after adding an edge
                break
        if boilerplate and draw_fn:
            if lang == "en":
                draw_fn(Arb, title=f"Partial arborescence -
                   Iteration {_+1}")
            elif lang == "pt":
                draw_fn(Arb, title=f"Arborescência parcial -
                   Iteração {_+1}")
    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 $|V| = |V(D\_original)|$ 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:* $O(|A\_zero| \log |A\_zero|)$ para operações de heap, mais $O(|A\_zero|)$ 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     Find the minimum arborescence in a directed graph D with
        root r0.
6     The function returns the minimum arborescence as a
        DiGraph.
7
8     Parameters:
9         - D_original: directed graph (DiGraph)
10        - r0: root node
11        - A_zero: list of arcs (u, v) that form the minimum
```

```
            arborescence
12
13      Returns:
14          - Arb: directed graph (DiGraph) representing the
                minimum arborescence
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(
2      Arb, Dual_list, log=None, boilerplate: bool = True, lang=
           "pt"
3  ):
4      """
5      Verifica a condição dual: z(X) > 0 implica que exatamente
           uma aresta de Arb entra em X.
6
7      Parameters:
8          - Arb: arborescência (DiGraph)
9          - Dual_list: lista de tuplas (X, z(X)) representando
               as variáveis duais
10         - r0: nó raiz
11
12     Returns:
13         - bool: True se a condição dual é satisfeita, False
               caso contrário
14     """
15     for X, z in Dual_list:
16         for u, v in Arb.edges():
17             count = 0
18             if u not in X and v in X:
```

```
19                  count += 1
20                  if count > 1:
21                      if boilerplate and log:
22                          if lang == "en":
23                              log(
24                                  f"\nDual condition failed for
                                        X={X} with z(X)={z}.
                                      Incoming arcs: {count}"
25                              )
26                          elif lang == "pt":
27                              log(
28                                  f"\nFalha na condição dual
                                      para X={X} com z(X)={z}.
                                      Arcos entrando: {count}"
29                              )
30                      return False
31      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 dígrafo 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

```
1 def andras_frank_algorithm(
2     D, draw_fn=None, log=None, boilerplate: bool = True, lang
          ="pt"
3 ):
4     if boilerplate and log:
5         if lang == "en":
```

```
6               log(f"\nExecuting András Frank algorithm...")
7           elif lang == "pt":
8               log(f"\nExecutando algoritmo de András Frank...")
9
10      A_zero, Dual_list = phase1_find_minimum_arborescence(
11          D, "r0", draw_fn=draw_fn, log=log, boilerplate=
                boilerplate, lang=lang
12      )
13
14      if boilerplate and log:
15          log(f"\nA_zero: \n{A_zero}")
16          log(f"\nDual_list: \n{Dual_list}")
17
18      if not has_arborescence(D, "r0"):
19          if boilerplate and log:
20              if lang == "en":
21                  log(f"\nThe graph does not contain an
                        arborescence with root r0.")
22              elif lang == "pt":
23                  log(f"\nO grafo não contém uma arborescência
                        com raiz r0.")
24          return None, None
25
26      arborescence_frank = phase2_find_minimum_arborescence(
27          D, "r0", A_zero, draw_fn=draw_fn, log=log,
                boilerplate=boilerplate, lang=lang
28      )
29      arborescence_frank_v2 =
            phase2_find_minimum_arborescence_v2(
30          D, "r0", A_zero, draw_fn=draw_fn, log=log,
                boilerplate=boilerplate, lang=lang
31      )
32
33      dual_frank = check_dual_optimality_condition(
34          arborescence_frank, Dual_list, log=log, boilerplate=
                boilerplate, lang=lang
35      )
36
37      dual_frank_v2 = check_dual_optimality_condition(
38          arborescence_frank_v2, Dual_list, log=log,
                boilerplate=boilerplate, lang=lang
39      )
40
41      if dual_frank and dual_frank_v2:
42          if boilerplate and log:
43              if lang == "en":
44                  log(f"\n✓ Dual condition satisfied for András
                        Frank.")
45              elif lang == "pt":
```

```
46                 log(f"\n✓ Condição dual satisfeita para Andrá
                      s Frank.")
47     else:
48         if boilerplate and log:
49             if lang == "en":
50                 log(f"\n× Dual condition failed for András
                      Frank.")
51             elif lang == "pt":
52                 log(f"\n× Condição dual falhou para András
                      Frank.")
53
54         if draw_fn:
55             if boilerplate and draw_fn:
56                 if lang == "en":
57                     draw_fn(
58                         arborescence_frank,
59                         title="András Frank Arborescence -
                            Method 1",
60                     )
61                     draw_fn(
62                         arborescence_frank_v2,
63                         title="András Frank Arborescence -
                            Method 2",
64                     )
65                 elif lang == "pt":
66                     draw_fn(
67                         arborescence_frank,
68                         title="Arborescência de András Frank
                            - Método 1",
69                     )
70
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.

De fato, o algoritmo de Frank é uma abordagem elegante e eficiente para encontrar arborescências de custo mínimo em dígrafos, 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, depois de toda essa exploração do método de Frank, vale a pena revisitar o clássico algoritmo de Chu–Liu/Edmonds para entender as diferenças e semelhanças entre essas duas abordagens fundamentais para o problema da arborescência de custo mínimo. A seguir, discutiremos as diferenças e semelhanças entre os métodos de Chu–Liu/Edmonds e Frank.

6 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 [**edmonds1967**, 1] é o método clássico para encontrar uma arborescência de custo mínimo em um dígrafo 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 [6, 5] 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 [6, 5, 8].

- **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 poderosos e 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. Entender as nuances de cada abordagem enriquece a compreensão do problema da arborescência de custo mínimo e das ferramentas disponíveis para resolvê-lo.

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 observados No recorte de 224 instâncias registrado em `test_results.csv` (amostra ilustrativa), obtivemos:

- **Correção e equivalência de custo.** Todos os testes finalizaram com sucesso (coluna *Sucesso* = OK) e ambas as checagens de *condição dual* retornaram `True` (colunas *Condicao_ dual_ Frank1/2*). Embora as colunas de custo no CSV estejam como "-" por limitação de *logging*, o código contém asserções que exigem $\text{custo}_{\text{Chu–Liu}} = \text{custo}_{\text{Frank v1}} = \text{custo}_{\text{Frank v2}}$. Logo, a igualdade de custos foi verificada em todas as instâncias desta amostra. Esse comportamento é consistente com a teoria: arcos *apertados* ($c' = 0$) somados à complementaridade garantem otimalidade [6, 5, 8], e o método de Edmonds é conhecido por devolver uma r-arborescência ótima [**edmonds1967**, 1].
- **Certificados duais.** A validação *a posteriori* via "exatamente uma entrada por corte ativo" (condição dual) passou em 100
- **Desempenho (tempo).** Os tempos totais por instância ficaram tipicamente entre $\approx$ 0,026 s e 0,121 s nesta faixa de n e m. Como a medida agrega todas as etapas (Chu–Liu, Fase I e Fase II), não separa o impacto relativo da Fase II v1 vs v2. Ainda assim, em linhas com $|A|$ até cerca de 600, o tempo permaneceu na casa de centésimos, coerente com implementações diretas eficientes e sobrecarga de Python/NetworkX.

Leitura à luz das referências Os achados alinham-se ao arcabouço teórico:

- **Complementaridade e laminaridade.** A Fase I de Frank produz arcos apertados e uma família de cortes ativa que pode ser tomada laminar por *uncrossing*; a Fase II escolhe apenas arcos com $c' = 0$. A combinação garante igualdade primal–dual [5, 8]. As checagens "True" no CSV corroboram essa leitura.

- **Equivalência com Edmonds.** A coincidência de custos entre Chu–Liu/Edmonds e Frank é esperada: apesar de roteiros distintos (combinatório vs primal–dual), ambos implementam a mesma mecânica de contrair/expandir orientada por mínimos e cortes [**edmonds1967**, 1, 6].
- **Complexidade.** A versão didática atinge $O(mn)$ em pior caso; variantes com estruturas adequadas atingem $O(m \log n)$ para operações de mínimos [5, 8]. A nossa versão "com heap" (Fase II v2) reduz o custo de expansão da arborescência sobre D_0, mas o gargalo prático pode permanecer na Fase I conforme a densidade e o padrão de pesos.

Limitações do experimento e melhorias

- **Registro de custos.** O CSV atual não persiste os custos (aparecem "-"), pois as variáveis locais de custo não são gravadas nas colunas correspondentes. Recomenda-se registrar explicitamente os valores para permitir análises estatísticas (média, desvio, dispersão). Ainda assim, as asserções no código garantem igualdade de custos quando o teste marca sucesso.
- **Granularidade temporal.** O tempo reportado é total por instância. Para comparar *v1* vs *v2* em Fase II e isolar a Fase I, sugere-se medir tempos por etapa (Chu–Liu, Fase I, Fase II v1, Fase II v2).
- **Variedade de instâncias.** Os grafos são aleatórios com pesos uniformes. Seria útil adicionar famílias adversárias/estruturadas (p.ex., quase completos, quase árvores, camadas com D_0 extenso/raso), bem como escalas maiores de n e m, para observar transições de gargalo e os ganhos da heap na prática.
- **Métricas adicionais.** Além de tempo e custo, incluir contagem de contrações, tamanho médio de D_0, profundidade de reexpansão e memória ajudaria a relacionar empiricamente com as previsões de [5, 8].

Conclusão Nesta amostra, ambos os métodos devolveram soluções ótimas e consistentes, com certificados duais válidos em todas as instâncias observadas e tempos compatíveis com a escala dos grafos testados. Esses resultados reforçam a equivalência prática entre as abordagens, ao mesmo tempo em que destacam o papel pedagógico do prisma primal–dual de Frank para explicar *por que* a seleção sobre D_0 é suficiente e como a laminaridade organiza a prova de otimalidade [6, 5, 8].

Para facilitar a compreensão desenvolemos uma aplicação web interativa que permita visualizar passo a passo o funcionamento dos dois algoritmos, destacando suas diferenças e semelhanças.

O aprendizado visual é uma ferramenta poderosa, e ver os algoritmos em ação pode ajudar a solidificar a compreensão dos conceitos discutidos. Na seção seguinte discutiremos aspectos didáticos relacionados ao aprendizado de digrafos, especialmente em estruturas mais complexas, traremos uma breve discussão sobre princípios de interação humano-computacional e como todos essas considerações culminaram na implementação da aplicação web.

7 Considerações Finais

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 [8].

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.

Referências

[1] Y. J. Chu e T. H. Liu. “On the Shortest Arborescence of a Directed Graph”. Em: *Scientia Sinica* 14 (1965), pp. 1396–1400.

[2] T. H. Cormen et al. *Introduction to Algorithms*. 3rd. MIT Press, 2009.

[3] Reinhard Diestel. *Graph Theory*. 5th. Springer, 2017. ISBN: 978-3662536216.

[4] J. Edmonds. “Optimum Branchings”. Em: *Journal of Research of the National Bureau of Standards* 71B (1967), pp. 233–240.

[5] 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.

[6] 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.

[7] J. Kleinberg e É. Tardos. *Algorithm Design*. Addison-Wesley, 2006.

[8] A. Schrijver. *Combinatorial Optimization: Polyhedra and Efficiency*. Springer, 2003.

[9] Douglas B. West. *Introduction to Graph Theory*. 2nd. Prentice Hall, 2001. ISBN: 978-0130144003.