

Algoritmos para r -Arborescências Geradoras Mínimas em Digrafos: Uma Aplicação Web Interativa

Lorena Sampaio, Samira Haddad
Orientador: Prof. Dr. Mário Leston Rey

Universidade Federal do ABC
Centro de Matemática, Computação e Cognição

27 de novembro de 2025

- 1 Introdução
- 2 Algoritmo de Chu-Liu-Edmonds
- 3 Algoritmo de András Frank
- 4 Resultados Experimentais
- 5 Didática do Abstrato
- 6 Aplicação Web
- 7 Conclusões

O Problema



Encontrar uma r -Arborescência Geradora de Custo Mínimo

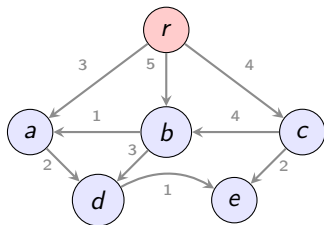
Dado um r -digrafo ponderado (D, w, r) :

- Encontrar uma r -arborescência geradora de custo mínimo de D

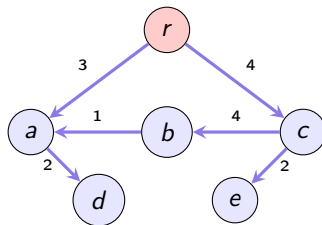
Algoritmos estudados:

- 1 Chu-Liu-Edmonds (1965-67)
- 2 András Frank (1981-2014)

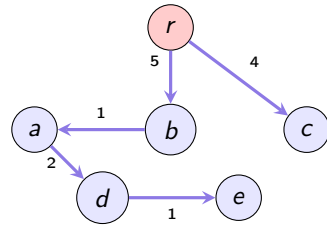
Exemplo: r -Arborescência Geradora Mínima



Digrafo Original

 r -Arborescência Geradora

Custo: 16



Geradora Mínima

Custo: 13

Chu-Liu-Edmonds



Algoritmo Recursivo: dado um r -digrafo ponderado (D, w, r)

$\text{chu-liu-edmonds}((D, w, r))$:

- 1 **Reduzir custos**: para cada vértice $v \neq r$, subtrair $\lambda(v) = \min\{w(a) : a \in \delta^-(v)\}$
- 2 **Construir D_0** : escolhendo um arco a_v de custo reduzido zero para cada $v \neq r$
- 3 **Verificar**: se D_0 é uma r -arborescência \Rightarrow **devolver** D_0
Caso contrário:
- 4 **Contração**: encontrar ciclo C em D_0 e contrair
- 5 **Chamada recursiva**: Seja $D' = D/C$ e $w' = w_\lambda/C$. Calcular $T' = \text{chu-liu-edmonds}(D', w', r)$
- 6 **Devolver**: expandir(T')

Escolha Gulosa

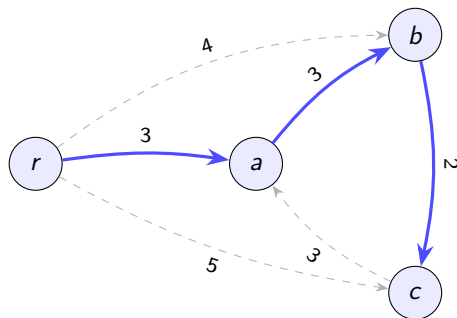
Definição:

Para cada $v \neq r$, escolher um arco a_v de custo mínimo que entra em v :

$$T := \{a_v : v \in V \setminus \{r\}\}$$

Propriedade:

Se T é uma r -arborescência, então T tem custo mínimo.



Resultado

$T = \{(r, a), (a, b), (b, c)\}$ é uma r -arborescência de custo mínimo!

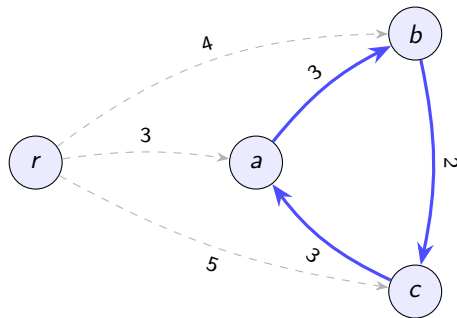
E quando a escolha gulosa falha?

Problema:

A escolha gulosa pode produzir um conjunto T que *não* é uma r -arborescência.

Exemplo:

Os arcs de custo mínimo formam um ciclo (a, b, c, a) sem alcançar r .



Arcos azuis formam um **ciclo**!

Passo 1: Redução de Custos

Definição:

Para cada $v \in V \setminus \{r\}$:

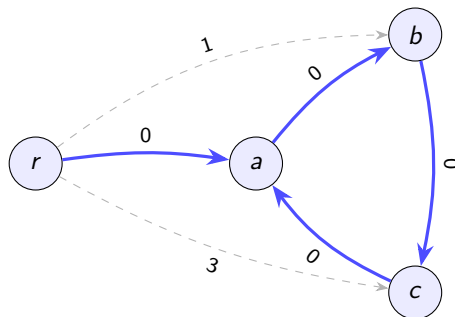
$$\lambda(v) := \min\{w(a) : a \in \delta^-(v)\}$$

Custo λ -reduzido:

$$w_\lambda(uv) := w(uv) - \lambda(v)$$

Valores de λ :

- $\lambda(a) = 3$, $\lambda(b) = 3$, $\lambda(c) = 2$



Arcos do ciclo têm custo zero!

Arcos com custo zero são candidatos para D_0

Implementação: Redução de Custos



Função reduce_weights:

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

Descrição:

- Calcula $\lambda(v) = \min\{w(a) : a \in \delta^-(v)\}$
- Reduz o custo de cada arco que entra em v
- Complexidade: $O(k)$ onde k é o número de arcos entrando em v

Resultado

Após executar `reduce_weights(D, v)` para cada $v \neq r$, todos os vértices têm ao menos um arco de entrada com custo zero.

Passo 2: Construção de D_0

Formação de D_0 : Para cada $v \neq r$, escolher um arco $a_v \in \delta^-(v)$ com $w_\lambda(a_v) = 0$ formar:

$$D_0 := (V, \{a_v : v \in V \setminus \{r\}\})$$

Arcos escolhidos:

- (r, a)
- $(a, b), (c, a)$



Passo 2: Implementação da Construção de D_0 em Python



Função `get_Dzero`:

```
def get_Dzero(D: nx.DiGraph, r: int):
    D_zero = nx.DiGraph()
    for v in D.nodes():
        if v != r:
            in_edges = D.in_edges(v,
                                   data=True)
            u = next((u for u, _, data
                      in in_edges
                      if data["w"] == 0))
            D_zero.add_edge(u, v)
    return D_zero
```

Descrição:

- Para cada vértice $v \neq r$, seleciona um arco com custo zero
- Constrói subdigrafo gerador D_0
- Garantido existir arco de custo zero após redução

Observação

Se D_0 for uma arborescência, então D_0 é necessariamente uma r -arborescência ótima.

Passo 3: Verificação de D_0

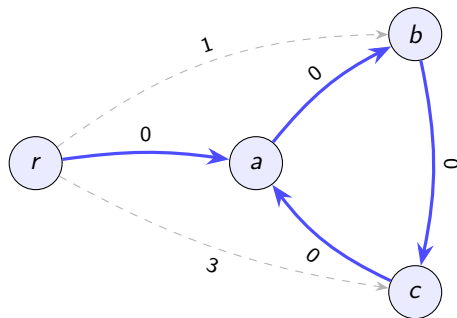
Verificar:

Se D_0 é uma r -arborescência \Rightarrow **devolver** D_0

Caso contrário:

D_0 contém algum ciclo C .

\Rightarrow **prosseguir** para os passos 4 e 5.



D_0 não é uma r -arborescência!

Neste exemplo, $D_0 = \{(r, a), (a, b), (c, a)\}$ não forma uma r -arborescência pois contém o ciclo (a, b, c, a) .

Passo 3: Implementação da Verificação de D_0 (1/2)



Verificação se é arborescência:

```
# Verificar se D_zero eh arborescencia
if nx.is_arborescence(D_zero):
    # Restaurar pesos originais
    for u, v in D_zero.edges:
        D_zero[u][v]["w"] = D[u][v]["w"]
    return D_zero
```

Esse trecho faz parte do caso base da função recursiva principal.

Caso Base

Se D_0 é uma arborescência, então ela é a r -arborescência de custo mínimo. Restauramos os pesos originais e devolvemos.

Passo 3: Implementação da Verificação de D0 (2/2)

Detecção de ciclo:

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

Descrição

- Usa `nx.find_cycle` para encontrar arcos do ciclo
- Coleta todos os vértices envolvidos
- Retorna subdigrafo induzido pelos vértices do ciclo

Passo 4: Contração de Ciclos

Operação:

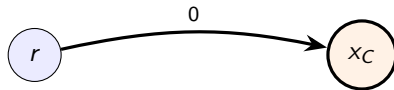
Contrair ciclo C em supervértice x_C .

Novo problema: (D', w', r) onde:

- $D' := D/C \mapsto x_C$
- $w' := w_\lambda/C \mapsto x_C$

O arco de D' que entra em x_C deve corresponder ao arco de D que entra em algum vértice de C

Podem ter arcos saindo de x_C em D' .



Digrafo contraído D'

Propriedade

Uma solução ótima em D' pode ser expandida para uma solução ótima em D .

Passo 4: Implementação da Contração (1/2)



Arcos essenciais que entram no ciclo:

```
def contract_cycle(D: nx.DiGraph, C: nx.DiGraph,
                  label: int):
    cycle_nodes: set[int] = set(C.nodes())

    # Arcos essenciais ENTRANDO no ciclo
    in_to_cycle: dict[int, tuple[int, float]] = {}
    for u in D.nodes:
        if u not in cycle_nodes:
            min_edge = min(
                ((v, data["w"])
                 for _, v, data in D.out_edges(u, data=True)
                 if v in cycle_nodes),
                key=lambda x: x[1], default=None)
            if min_edge:
                in_to_cycle[u] = min_edge

    for u, (v, w) in in_to_cycle.items():
        D.add_edge(u, label, w=w)
```

Arcos Essenciais de Entrada

Para cada vértice externo, encontra o arco de menor custo que entra no ciclo

Passo 4: Implementação da Contração (2/2)



Arcos essenciais que saem do ciclo:

```
# Arcos essenciais SAINDO do ciclo
out_from_cycle: dict[int, tuple[int, float]] = {}
for v in D.nodes:
    if v not in cycle_nodes:
        min_edge = min(
            ((u, data["w"])
             for u, _, data in D.in_edges(v, data=True)
             if u in cycle_nodes),
            key=lambda x: x[1], default=None)
        if min_edge:
            out_from_cycle[v] = min_edge

for v, (u, w) in out_from_cycle.items():
    D.add_edge(label, v, w=w)

D.remove_nodes_from(cycle_nodes)
return in_to_cycle, out_from_cycle
```

Arcos Essenciais de Saída

Os dicionários retornados são usados na expansão. Complexidade: $O(m)$

Passo 5: Chamada Recursiva

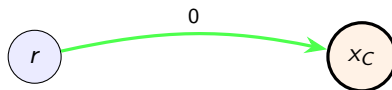


Novo problema: (D', w', r)

Chamada recursiva:

$$T' := \text{chu-liu-edmonds}(D', w', r)$$

Resultado: T' é uma r -arborescência de custo mínimo em (D', w')



r -arborescência ótima em D'

Passo 5: Implementação da Chamada Recursiva



Estrutura recursiva:

```
def chuliu_edmonds(D: nx.DiGraph, r: int, label: int):
    D_copy = D.copy()

    # Reducao de custos
    for v in D_copy.nodes:
        if v != r:
            reduce_weights(D_copy, v)

    D_zero = get_Dzero(D_copy, r)

    if nx.is_arborescence(D_zero):
        # Restaurar pesos e devolver
        for u, v in D_zero.edges:
            D_zero[u][v]["w"] = D[u][v]["w"]
        return D_zero

    # Contrair ciclo e recursao
    C = find_cycle(D_zero)
    in_to_cycle, out_from_cycle = \
        contract_cycle(D_copy, C, label)
    F_prime = chuliu_edmonds(D_copy, r, label + 1)
    # ... expansao ...
```

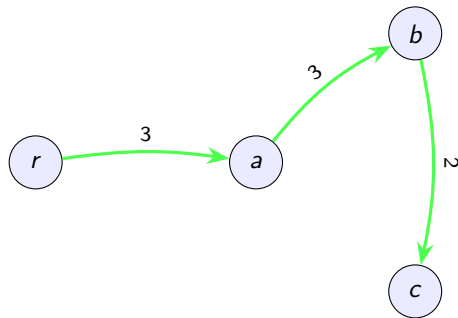
Passo 6: Reexpansão da Solução

Dado: T' ótima em (D', w')

Construir: T ótima em (D, w)

Procedimento:

- 1 Seja uv o arco de D correspondente ao arco ux_C de T'
- 2 Incluir uv em T
- 3 Incluir todos os arcos de C exceto aquele que entra em v



Resultado: T é uma r -arborescência de custo mínimo

r -arborescência final no digrafo original

Passo 6: Implementação da Reexpansão (1/2)



Encontrar e adicionar arco correspondente:

```
# F_prime: solucao em D' (com supervertice)
# Encontrar arco que entra em label
in_edge = next(iter(
    F_prime.in_edges(label, data=True)))
u, _, _ = in_edge

# Arco correspondente em D original
v, _ = in_to_cycle[u]
F_prime.add_edge(u, v)

# Adicionar arcos do ciclo (exceto o que entra em v)
for u_c, v_c in C.edges:
    if v != v_c:
        F_prime.add_edge(u_c, v_c)
```

Observação: Identifica qual arco do ciclo não será incluído na solução final.

Passo 6: Implementação da Reexpansão (2/2)



Transferir arcos externos e restaurar pesos:

```
# Arcos que saem do supervertice
for _, z, _ in list(
    F_prime.out_edges(label, data=True)):
    u_cycle, _ = out_from_cycle[z]
    F_prime.add_edge(u_cycle, z)

# Remover supervertice
F_prime.remove_node(label)

# Restaurar pesos originais
for u, v in F_prime.edges:
    F_prime[u][v]["w"] = D[u][v]["w"]

return F_prime
```

Correção: A expansão garante que T é uma r -arborescência ótima no digrafo original.

Algoritmo de András Frank



Abordagem em Duas Fases

Fase I (Fulkerson): Construir cobertura de r -conjuntos via redução de custos

Fase II (Frank): Extrair r -arborescência geradora da cobertura

Fase I: dado um r -digrafo ponderado (D, w, r)

- 1 **Inicializar:** $c := w$, $\sigma := \epsilon$ (sequência vazia)
- 2 **Enquanto** F não cobre todos r -conjuntos:
 - Encontrar r -conjunto minimal R_k não coberto por F
 - Calcular $\lambda_k := \min\{c(a) : a \in \delta^-(R_k)\}$
 - Selecionar $f_k \in \delta^-(R_k)$ com $c(f_k) = \lambda_k$
 - $\sigma := \sigma \cdot (f_k, R_k, \lambda_k)$
 - $c := c - \lambda_k 1_{\delta^-(R_k)}$
- 3 **Devolver:** $\sigma = ((f_i, R_i, \lambda_i))_{i \in [k]}$

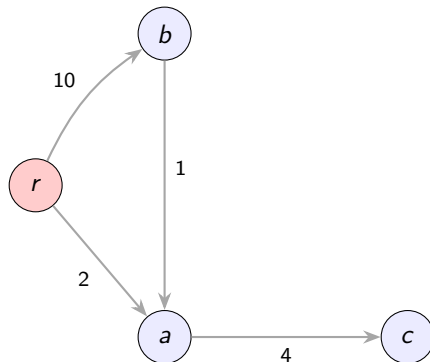
Exemplo: Digrafo Inicial

Digrafo ponderado (D, w, r) :

- Vértices: $\{r, a, b, c\}$
- Raiz: r
- Objetivo: encontrar r -arborescência geradora de custo mínimo

Pesos dos arcos:

- $(r, a) : 2$, $(r, b) : 10$
- $(b, a) : 1$, $(a, c) : 4$



Problema

Aplicar o algoritmo de András Frank para encontrar a r -arborescência geradora de custo mínimo

Fase I - Iteração 1: Encontrar r -conjunto Minimal

Estado inicial:

- $F = \emptyset$ (nenhum arco selecionado)
- $D_0 = (V, \emptyset)$
- $\mathcal{C}(D_0)$ tem 4 fontes

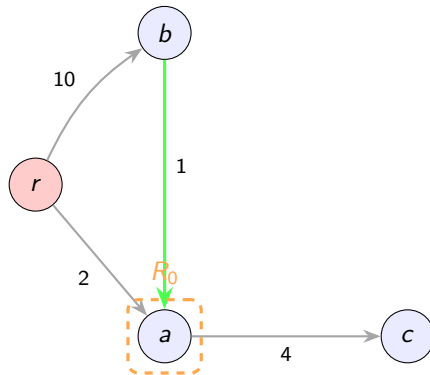
Fontes que são r -conjuntos:

- $\{a\}, \{b\}, \{c\}$

Escolha: $R_0 = \{a\}$ (minimal)

Arcos que entram em $\{a\}$:

- $(r, a) : 2$
- $(b, a) : 1 \leftarrow$ **mínimo**



$$\lambda_0 = 1, f_0 = (b, a)$$

Fase I - Iteração 1: Redução de Custos

Atualização:

- $\sigma = [(f_0, R_0, \lambda_0)]$
- $F = \{(b, a)\}$
- $D_0 = (V, \{(b, a)\})$

Redução de custos:

- $c(r, a) = 2 - 1 = 1$
- $c(b, a) = 1 - 1 = 0 \checkmark$

Custos reduzidos:

- $(r, a) : 1, (r, b) : 10$
- $(b, a) : 0, (a, c) : 4$



Arcos azuis têm custo zero

Fase I - Iteração 2: Próximo r -conjunto

Estado: $F = \{(b, a)\}$

$\mathcal{C}(D_0)$ agora tem:

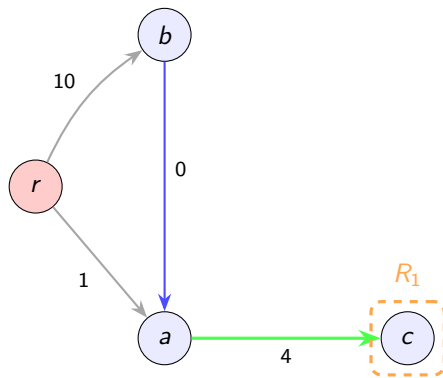
- CFC: $\{b, a\}$ ($b \rightarrow a$)
- Fontes: $\{r\}$, $\{c\}$

Escolha: $R_1 = \{c\}$

Arcos que entram em $\{c\}$:

- $(a, c) : 4 \leftarrow$ único

$\lambda_1 = 4$, $f_1 = (a, c)$



Fase I - Iteração 2: Redução e Estado Final

Atualização:

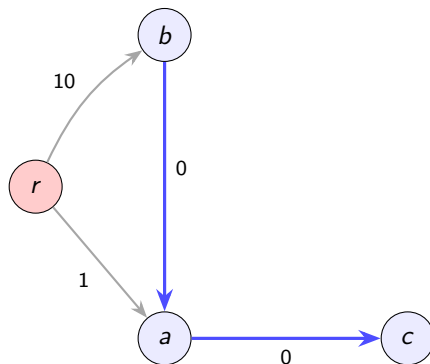
- $F = \{(b, a), (a, c)\}$
- $D_0 = (V, F)$

Redução:

- $c(a, c) = 4 - 4 = 0 \checkmark$

 $\mathcal{C}(D_0)$ agora:

- CFC: $\{r\}, \{b, a, c\}$
- Apenas 1 fonte (contém r)



Fim da Fase II!

Cobertura de r -conjuntos construída!

Sequência devolvida

Fase II: Extração da Arborescência

Entrada da Fase II:

- $F = [f_0, f_1] = [(b, a), (a, c)]$

Algoritmo guloso:

- 1 $U = \{r\}, J = \emptyset$
- 2 Iterar sobre F em ordem:
 - (b, a) : $b \notin U \times$ pular
 - (a, c) : $a \notin U \times$ pular
- 3 Precisamos incluir arcos de D não em F !

Verde = vértices em U

Observação

Fase II: Construção Passo a Passo

Iteração 1:

- $U = \{r\}$
- Procurar arco saindo de U
- (r, a) : $r \in U$, $a \notin U$ ✓
- $U := U \cup \{a\}$
- $J := \{(r, a)\}$

Iteração 2:

- (b, a) : $a \in U$ ✗
- (a, c) : $a \in U$, $c \notin U$ ✓
- $U := \{r, a, c\}$
- $J := \{(r, a), (a, c)\}$



$J =$ arborescência geradora

Fase I: Sequência w -disjunta



Conceitos fundamentais:

Uma sequência $((R_i, \lambda_i))_{i \in [k]}$ é **w -disjunta** se:

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

para cada arco $a \in A(D)$.

Interpretação:

- λ_i é a multiplicidade do r -conjunto R_i
- O peso total usado por todos os R_i não excede $w(a)$

Coleção Laminar

$\{R_i : i \in [k]\}$ é laminar: para quaisquer R_i, R_j :

Fase I: Encontrando r -conjuntos Minimais



Como encontrar um r -conjunto minimal não coberto?

Seja $D_0 := (V, F)$ onde $F = \{f_i : i \in [k]\}$.

- 1 Calcular a condensação $\mathcal{C}(D_0)$
- 2 Identificar componentes fortemente conexas (CFCs)
- 3 Encontrar uma fonte S em $\mathcal{C}(D_0)$ tal que $r \notin S$

Proposição

Toda fonte S de $\mathcal{C}(D_0)$ com $r \notin S$ é um r -conjunto minimal não coberto por F .

Complexidade: Identificação de CFCs em $O(|A|)$ usando Kosaraju

Fase I: Implementação - Estrutura Principal



Função phase1:

```
def phase1(D: nx.DiGraph, r: int):  
    D_copy = D.copy()  
    sigma = []  
    D_zero = nx.DiGraph()  
    D_zero.add_nodes_from(D_copy.nodes())  
  
    while True:  
        C = nx.condensation(D_zero)  
        sources = [x for x in C.nodes()  
                   if C.in_degree(x) == 0]  
        if len(sources) == 1:  
            break  
  
        for s in sources:  
            X = C.nodes[s]["members"]  
            if r in X:  
                continue  
            # ... (continua no proximo slide)
```

Observação: Loop principal até todos r -conjuntos estarem cobertos

Fase I: Implementação - Seleção de Arcos



Continuação da função phase1:

```
# ... (continuacao do loop)
arcs = [(u, v, data)
        for u, v, data in D_copy.edges(data=True)
        if u not in X and v in X]

min_weight = min(data["w"] for _, _, data in arcs)
a = update_weights(D_copy, arcs, min_weight)

D_zero.add_edge(a[0], a[1])
sigma.append((a, X, min_weight))

return sigma
```

Complexidade: $O(|V||A|)$ - limitado por $2|V| - 1$ iterações

Fase I: Implementação - Atualização de Pesos



Função update_weights:

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

Descrição

- Reduz peso de todos arcos que entram no r -conjunto
- Devolve arco com peso zero (custo reduzido mínimo)
- Atualiza o digrafo in-place

Fase II: Construção da Arborescência



Entrada: Sequência $(f_i)_{i \in [k]}$ da Fase I

Objetivo: Extrair $J \subseteq \{f_i : i \in [k]\}$ que é uma r -arborescência geradora

Algoritmo guloso:

- 1 Iniciar com $U := \{r\}$ e $J := \emptyset$
- 2 Para $t = 1$ até $|V| - 1$:
 - Para cada $f_i = (u_i, v_i)$ na sequência:
 - Se $u_i \in U$ e $v_i \notin U$:
 - $U := U \cup \{v_i\}$
 - $J := J \cup \{f_i\}$
 - Passar para próxima iteração

Invariante

Em cada iteração, $\varrho_J(R_i) \leq 1$ para todo $i \in [k]$

Fase II: Implementação - Versão Lista



Versão 1: Iteração sobre lista

```
def phase2(D: nx.DiGraph, r: int,
          F: list[tuple[int, int]]):
    Arb = nx.DiGraph()
    Arb.add_node(r)
    n = len(D.nodes())

    for _ in range(n - 1):
        for u, v in F:
            if u in Arb.nodes() and v not in Arb.nodes():
                edge_data = D.get_edge_data(u, v)
                Arb.add_edge(u, v, **edge_data)
                break

    return Arb
```

Complexidade: $O(|V||F|) = O(|V|^2)$ pois $|F| \leq 2|V| - 1$

Fase II: Implementação - Versão Heap



Versão 2: Usando fila de prioridade (estilo Dijkstra)

```
def phase2_v2(D, r, F):
    Arb = nx.DiGraph()
    for i, (u, v) in enumerate(F):
        Arb.add_edge(u, v, w=i) # prioridade = indice

    V = {r}
    q = []
    for u, v, data in Arb.out_edges(r, data=True):
        heapq.heappush(q, (data["w"], u, v))

    J = nx.DiGraph()
    while q:
        _, u, v = heapq.heappop(q)
        if v in V: continue
        J.add_edge(u, v, w=D[u][v]["w"])
        V.add(v)
        for x, y, data in Arb.out_edges(v, data=True):
            heapq.heappush(q, (data["w"], x, y))
    return J
```

Complexidade: $O(|V| \log |V|)$ usando heap binário

Algoritmo Completo de Frank

Composição das duas fases:

```
def andras_frank(D: nx.DiGraph, r: int):
    # Fase I: construir cobertura
    sigma = phase1(D, r)
    F = [f for f, _, _ in sigma]

    # Fase II: extrair arborescencia
    J = phase2_v2(D, r, F)

    return J
```

Complexidade Total

- Fase I: $O(|V||A|)$
- Fase II (heap): $O(|V| \log |V|)$
- **Total:** $O(|V|(|A| + \log |V|))$

Comparação de Desempenho



Experimentos: 2000 digrafos aleatórios, $|V| \in [101, 4996]$

Algoritmo	Tempo Mediano	Tempo Médio
Chu-Liu-Edmonds	0,25 s	0,58 s
Frank Fase I	8,93 s	12,40 s
Frank Fase II (lista)	0,98 s	1,34 s
Frank Fase II (heap)	0,016 s	0,020 s

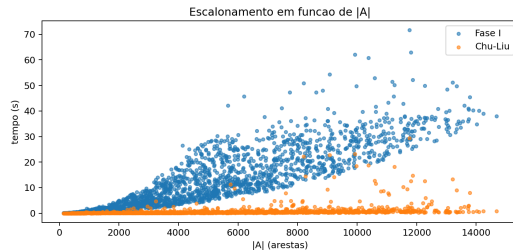
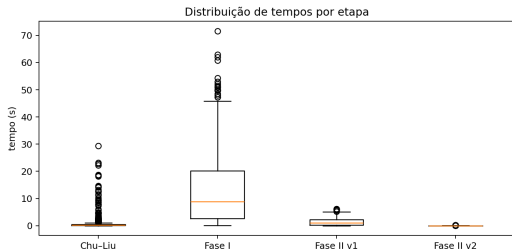
Speedup Fase II

Heap vs Lista: aceleração de **58,12 vezes** (mediana)

Escalonamento e Consumo de Memória

Escalonamento temporal:

- Tempo cresce linearmente com o número de arestas
- Fase I de Frank domina o tempo total
- Fase II (heap) é residual e muito rápida



Principais Resultados



- **Corretude validada:** custos idênticos em todas as instâncias
- **Chu-Liu-Edmonds** mais rápido para construção direta
 - Mediana: 0,25 s vs 8,93 s (Fase I Frank)
- **Otimização heap** fundamental na Fase II
 - Speedup: $58\times$ (mediana), $61\times$ (média)
- **Comportamento prático** muito melhor que limites teóricos
 - Contrações: mediana 2 (limite $O(n)$)
 - Memória modesta: 11,5 MB

Conclusões dos Experimentos



- Equivalência teórica e prática dos algoritmos confirmada
- Chu-Liu/Edmonds é mais eficiente
- Fase I de Frank é o gargalo computacional
- Heap na Fase II traz ganhos práticos expressivos
- Algoritmos são escaláveis e viáveis para grandes digrafos

Desafios na Ensino de Algoritmos de Grafos



Abstração Excessiva

- Dificuldade em visualizar estruturas
- Compreensão limitada dos passos do algoritmo

Solução Proposta

Ferramenta interativa para visualização e execução passo a passo

Objetivos da Ferramenta Didática



- Facilitar a compreensão dos algoritmos Chu-Liu-Edmonds e András Frank
- Permitir aos usuários interagir com grafos e observar o funcionamento dos algoritmos
- Fornecer feedback imediato sobre as operações realizadas
- Ser acessível via navegador web, sem necessidade de instalação

Motivação Didática



Desafio

Algoritmos de grafos são **abstratos** e **difíceis de visualizar**

Solução Proposta:

- Visualização interativa
- Execução passo a passo
- Feedback imediato
- Acessível via navegador

Tecnologias:

- PyScript (Python no browser)
- JavaScript
- HTML5/CSS3
- NetworkX

Aplicação web

Ferramenta web interativa para ensino de algoritmos de arborescências dirigidas, permitindo visualização passo a passo, edição livre de grafos e exportação de resultados, com arquitetura modular e foco didático.

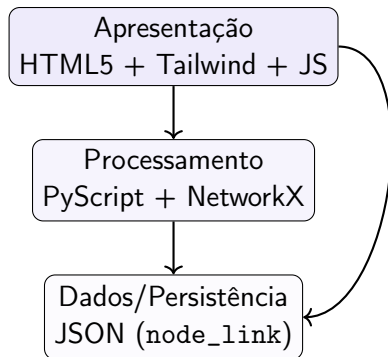
Módulos principais:

- **Visualização Algorítmica:** Páginas para execução passo a passo dos algoritmos Chu-Liu-Edmonds e András Frank.
- **Modelagem Livre:** Editor sandbox para desenhar, testar e exportar grafos arbitrários.
- **Disseminação Científica:** Página informativa sobre o projeto e a dissertação.

Arquitetura da Aplicação

Estrutura em três camadas:

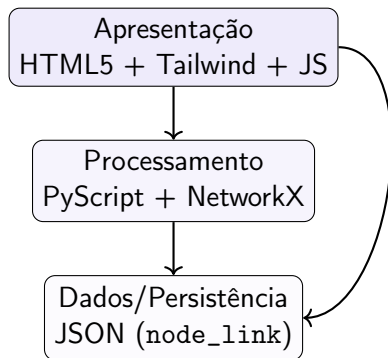
- **Apresentação:** Interface construída em HTML5, estilizada com Tailwind CSS e dinamizada por JavaScript.
- **Processamento (PyScript):** Executa algoritmos em Python (NetworkX) diretamente no navegador, gerando visualizações estáticas com Matplotlib.
- **Dados e Persistência:** Utiliza JSON (node_link) para serializar grafos, armazenar pesos e permitir exportação/importação entre módulos.



Arquitetura da Aplicação

Benefícios:

- Processamento local e rápido
- Facilidade de uso e reprodutibilidade
- Modularidade e extensibilidade



Princípios de IHC Aplicados



Fundamentos para o Design da Ferramenta

O desenvolvimento da aplicação web foi guiado por oito princípios de Interação Humano-Computador (IHC), integrando heurísticas de usabilidade e teorias de aprendizagem:

- **Usabilidade:** Interface limpa, controles claros e navegação intuitiva.
- **Eficiência cognitiva:** Redução da carga mental, destaque para informações relevantes.
- **Feedback imediato:** Atualização visual e textual em tempo real a cada ação.
- **Engajamento ativo:** Usuário explora, manipula e prediz resultados.
- **Visão geral com detalhe sob demanda**
- **Consistência semântica:** Terminologia e estilos padronizados em toda a interface.
- **Múltiplos registros de representação:** Grafo visual, log textual e parâmetros simbólicos.
- **Prevenção e recuperação de erros**

Interface: Página Principal


ArboGraph

 Home
 Chu-Liu-Edmonds
 András Frank (V1)
 András Frank (V2)
 Desenhe um digrafo
 Nossa dissertação



Dúvidas ?

Quer aprender mais sobre esses algoritmos, leia nossa tese :)

[Link](#)

Idioma:  PT  EN

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

Resumo

Este trabalho investiga e implementa algoritmos de busca de uma r -arborescência geradora mínima em digrafos. A partir da formulação clássica e da literatura de Chu–Liu–Edmonds e também da formulação de András Frank, desenvolvemos uma aplicação web que permite: (i) desenhar ou importar um digrafo ponderado, (ii) escolher o nó raiz r , (iii) executar o algoritmo passo a passo com visualização das contrações, seleção de arcos de custo mínimo e reconstrução da arborescência, e (iv) exportar resultados e logs. A solução combina PyScript e NetworkX para a lógica algorítmica, Cytoscape para edição e visualização interativa, e Tailwind/Flowbite na interface. Como contribuição, o sistema oferece um ambiente didático que torna transparentes as decisões do algoritmo e facilita a análise e comparação de soluções em diferentes instâncias, apoiando ensino, experimentação e validação.

Integrantes do Projeto



Lori Sampaio
Discente do BCC-UFABC
[LinkedIn](#) | [Email](#)




Mario Leston Rey
Docente do BCC-UFABC
[LinkedIn](#) | [Email](#)




Samira Haddad
Discente do BCC-UFABC
[LinkedIn](#) | [Email](#)

Interface: Desenho de Grafos

 **ArboGraph**

- Home
- Chu-Liu/Edmonds
- Andras Frank (V1)
- Andras Frank (V2)
- Desenhe um grafo
- Nossa tese

**Dúvidas ?**

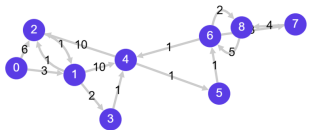
Quer aprender mais sobre esses algoritmos, leia nossa tese :)

[Link](#)


Desenhe seu grafo

1. Desenhe um grafo, [carregue](#) um exemplo ou [importe](#) um grafo já existente.

Grafo Original



Interface: Chu-Liu-Edmonds

 **ArboGraph**

Chu-Liu / Edmonds

» Passo A Passo

🏠 Home

👤 Chu-Liu/Edmonds

👤 Andras Frank (V1)

👤 Andras Frank (V2)

👤 Desenhe um grafo

📖 Nossa tese

💡


Dúvidas ?

Quer aprender mais sobre esses algoritmos, veja nossa tese :)

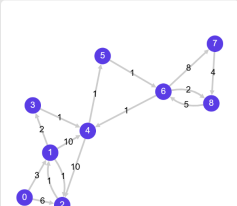
Link



1 Crie um grafo
Desenhe um grafo, carregue um exemplo ou importe um grafo já existente.

2 Escolha o nó raiz

3 Execute o algoritmo


Grafo Original




Execute o algoritmo para visualizar o passo-a-passo


Interface: András Frank





 **ArboGraph**


Andras Frank v1


[» Passo A Passo](#)


 Home


 Chu-Liu/Edmonds

 Andras Frank (V1)

 Andras Frank (V2)

 Desenhe um grafo

 Nossa tese




Dúvidas ?
Quer aprender mais sobre
esses algoritmos, leia nossa
tese :)


[Link](#)

1 Crie um grafo
Desenhe um grafo, **carregue** um exemplo ou
importe um grafo já existente.

2 Escolha o nó raiz

3 Execute o algoritmo


Grafo Original



Execute o algoritmo para visualizar o passo-a-
passo

Contribuições do Trabalho



1 Implementação completa de dois algoritmos clássicos

- Chu-Liu-Edmonds: recursivo com contração
- András Frank: duas fases com otimização heap

2 Análise experimental detalhada

- 2000 instâncias aleatórias
- Comparação de desempenho e características estruturais

3 Aplicação web interativa

- Ferramenta didática para visualização
- Execução passo a passo dos algoritmos
- Design centrado no usuário

Trabalhos Futuros



Extensões Possíveis

- Implementar outras variantes (Tarjan, Gabow)
- Análise em grafos com estruturas especiais
- Paralelização dos algoritmos
- Extensão para grafos dinâmicos

Melhorias na Aplicação

- Modo de edição visual de grafos
- Geração automática de casos de teste
- Exercícios interativos com correção automática
- Integração com plataformas de ensino (Moodle, Jupyter)

Obrigado!

Perguntas?

<https://github.com/lorenypsum/graph-visualizer>