

# 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

26 de novembro de 2025

# Sumário

- 1 Introdução
- 2 Algoritmo de Chu-Liu-Edmonds
- 3 Algoritmo de András Frank
- 4 Resultados Experimentais
- 5 Aplicação Web
- 6 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  $D$

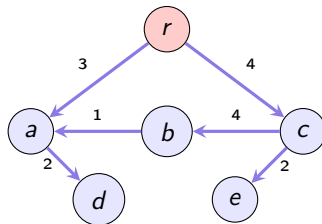
**Algoritmos estudados:**

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

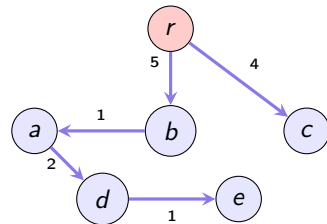
# Exemplo: $r$ -Arborescência Geradora Mínima



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

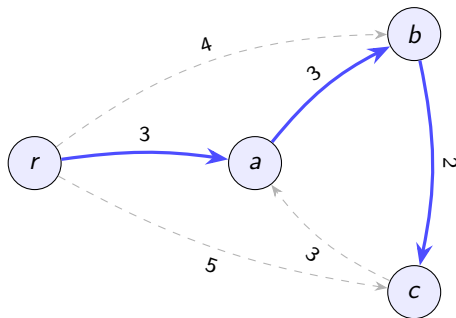
## Definição:

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!

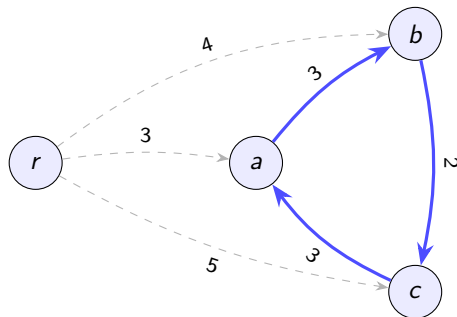
# E quando a escolha gulosa falha?

## Problema:

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

## Exemplo:

Os arcos 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  $v \in V \setminus \{r\}$ :

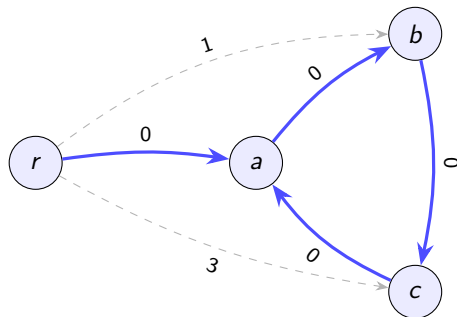
$$\lambda(v) := \min\{w(a) : a \in \delta^-(v)\}$$

Custo  $\lambda$ -reduzido:

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

## Valores de $\lambda$ :

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



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.

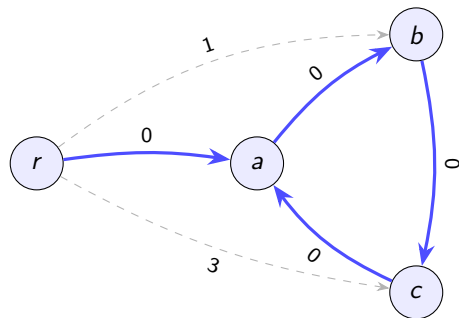
## Passo 2: Construção de $D_0$

**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 $D_0$

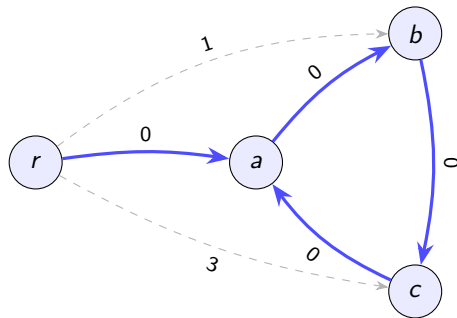
**Verificar:**

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 D0 em Python

## Verificação e detecção de ciclo:

```
# 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

# Encontrar 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)
```

## Descrição:

- Usa função de biblioteca `is_arborescence`
- Se não for arborescência, encontra um ciclo
- `find_cycle` retorna os arcos do ciclo
- Constrói subdigrafo induzido pelos vértices do ciclo

## Passo 4: Contração de Ciclos

### Operação:

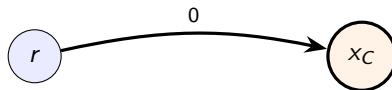
Contrair ciclo  $C$  em supervértice  $x_C$ .

**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 de Ciclos em Python

## Função `contract_cycle` (parte 1):

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

    # Encontrar arcos essenciais que ENTRAM 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

    # Adicionar arcos de u para label (supervertice)
    for u, (v, w) in in_to_cycle.items():
        D.add_edge(u, label, w=w)
```

Arcos **essenciais**: menor custo de cada vértice externo para o ciclo

# Passo 4: Implementação da Contração de Ciclos em Python (cont.)

## Função contract\_cycle (parte 2):

```
# Encontrar arcos essenciais que SAEM 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

# Adicionar arcos de label para v
for v, (u, w) in out_from_cycle.items():
    D.add_edge(label, v, w=w)

# Remover vertices do ciclo
D.remove_nodes_from(cycle_nodes)
return in_to_cycle, out_from_cycle
```

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



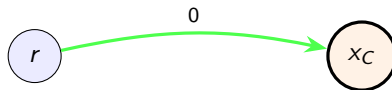
## Passo 5: Chamada Recursiva

**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 em Python

## Estrutura recursiva do algoritmo:

```
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
        ...
        return D_zero

    # Contrair ciclo e chamar recursao
```

## Observações:

- `label` identifica o supervértice
- Cada chamada recursiva incrementa `label`
- Caso base:  $D_0$  é arborescência
- Caso recursivo: contrai ciclo e resolve subproblema

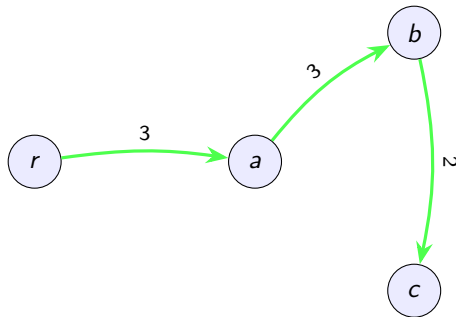
## Passo 6: Reexpansão da Solução

**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  $v$



$r$ -arborescência final no digrafo original

**Resultado:**  $T$  é uma  $r$ -arborescência de custo mínimo

# Passo 6: Implementação da Reexpansão da Solução em Python

## Expansão após chamada recursiva:

```
# F_prime eh 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)

# 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
```

## Passos da expansão:

- ① Identificar arco que entra em label
- ② Encontrar arco correspondente em  $D$
- ③ Adicionar arcos do ciclo  $C$  (exceto um)
- ④ Transferir arcos que saem de label
- ⑤ Remover label
- ⑥ Restaurar pesos originais

# András Frank: Visão Geral

## Abordagem em Duas Fases

**Fase I:** Construir cobertura de subconjuntos minimais via redução de custos

**Fase II:** Extrair arborescência da cobertura

### Diferencial:

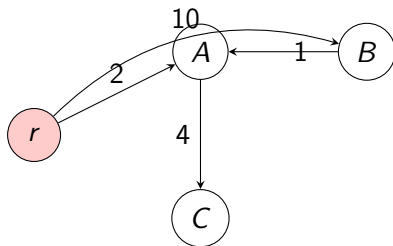
- Trabalha com múltiplos vértices simultaneamente
- Usa componentes fortemente conexas
- Redução sistemática de custos

### Complexidade:

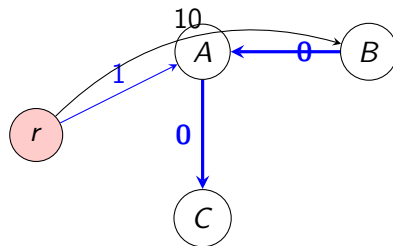
- Fase I:  $O(nm)$
- Fase II v1 (lista):  $O(n^2)$
- Fase II v2 (heap):  $O(n \log n)$

# Fase I: Redução de Custos

Para cada vértice  $v \neq r$ : subtrair o mínimo de entrada



Original



Após Redução

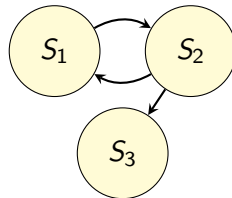
Arcos com custo **zero** formam o digrafo  $D_0$

# Fase I: Componentes Fortemente Conexas

Identificar componentes fortemente conexas (CFCs) em  $D_0$

Cada CFC forma um **subconjunto minimal**

Construir sequência laminar de subconjuntos



## Condição de Otimalidade

Sequência  $\lambda$  satisfaz:  $|\delta^-(X)| = 1$  para cada  $X$  em  $\lambda$

## Fase II: Construção da Arborescência

**Objetivo:** Extrair arborescência de  $D_0$  respeitando  $\lambda$

- 1 Iniciar com conjunto  $R = \{r\}$
- 2 Para cada  $v$  fora de  $R$ :
  - Selecionar arco  $(u, v)$  com  $u \in R$  e custo reduzido zero
  - Adicionar  $v$  a  $R$
- 3 Repetir até incluir todos os vértices

### Resultado

Arborescência ótima com mesma solução: custo 14



# 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)	<b>0,016 s</b>	<b>0,020 s</b>

## Speedup Fase II

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

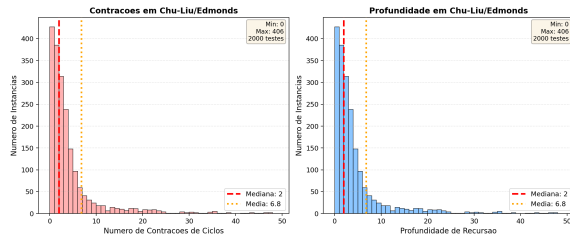
# Características Estruturais

## Contrações (Chu-Liu):

- Mediana: 2 contrações
- Média: 6,82
- Máximo: 406
- 93,8% com  $< 20$

Muito abaixo do limite teórico  $O(n)$

**Consumo de memória:** mediana 11,5 MB (Fase I)



# Motivação Didática

## Desafio

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

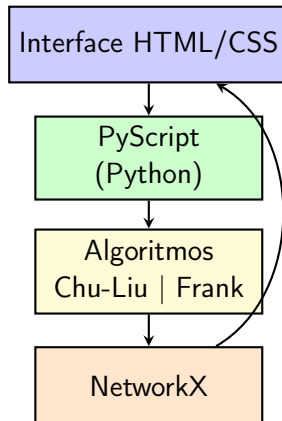
## Solução Proposta:

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

## Tecnologias:

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

# Arquitetura da Aplicação



# 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](#)

## 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







# Interface: Desenho de Grafos

The screenshot shows the ArboGraph web interface. On the left is a sidebar with a navigation menu containing: Home, Chu-Liu/Edmonds, Andras Frank (V1), Andras Frank (V2), Desenhe um grafo (selected), and Nossa tese. Below the menu is a 'Dúvidas ?' (Doubts?) section with a lightbulb icon and a 'Link' button. The main area is titled 'Desenhe seu grafo' (Draw your graph) and contains instructions: '1. Desenhe um grafo, carregue um exemplo ou importe um grafo já existente.' Below this is a section labeled 'Grafo Original' showing a directed graph with 9 nodes (0-8) and weighted edges. The graph structure is as follows: Node 0 points to 2 (weight 6) and 1 (weight 3). Node 2 points to 1 (weight 1) and 4 (weight 10). Node 1 points to 4 (weight 10) and 3 (weight 2). Node 3 points to 4 (weight 1). Node 4 points to 6 (weight 1) and 5 (weight 1). Node 6 points to 8 (weight 2) and 5 (weight 1). Node 8 points to 7 (weight 4) and 5 (weight 5). There are also self-loops on nodes 2 and 8. To the right of the graph are download and delete icons.

## Funcionalidades:

- Adicionar vértices e arestas
- Definir pesos

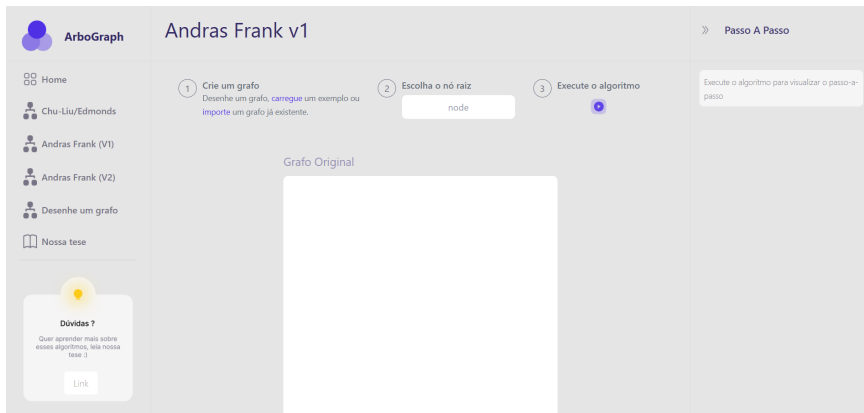
# Interface: Chu-Liu-Edmonds

The screenshot shows the ArboGraph web application interface for the Chu-Liu / Edmonds algorithm. The interface is divided into three main sections:

- Left Sidebar:** Contains navigation links: Home, Chu-Liu/Edmonds, Andras Frank (V1), Andras Frank (V2), Desenhe um grafo, and Nossa tese. There is also a "Dúvidas?" (Doubts?) section with a link to learn more about the algorithms.
- Top Section:** Displays the title "Chu-Liu / Edmonds" and a "Passo A Passo" (Step by Step) button.
- Main Content Area:**
  - Step 1: Crie um grafo** (Create a graph): Includes instructions to draw a graph, load an example, or import an existing graph.
  - Step 2: Escolha o nó raiz** (Choose the root node): A dropdown menu showing "0".
  - Step 3: Execute o algoritmo** (Execute the algorithm): A button to execute the algorithm.
  - Grafo Original:** A visual representation of the original graph with 9 nodes (0-8) and weighted edges. The edges and their weights are: (0,1):3, (0,2):6, (1,2):1, (1,3):2, (1,4):10, (2,4):1, (3,4):1, (4,5):1, (4,6):1, (5,6):1, (6,7):8, (6,8):2, (7,8):4, and (8,8):5 (self-loop).

- Visualização passo a passo
- Destacamento de ciclos detectados
- Log detalhado das operações

# Interface: András Frank



- Exibição das duas fases
- Visualização de CFCs
- Comparação entre versões (lista vs heap)



# Princípios de Design

## Teoria dos Registros de Representação (Duval)

Transitar entre diferentes representações:

- **Visual:** diagramas do grafo
- **Simbólico:** código Python
- **Textual:** log das operações

## Feedback Imediato

Validação em tempo real das operações do usuário

# 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

# 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

# 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>