Grafos – Árvore geradora mínima

Prof. Renato Pimentel

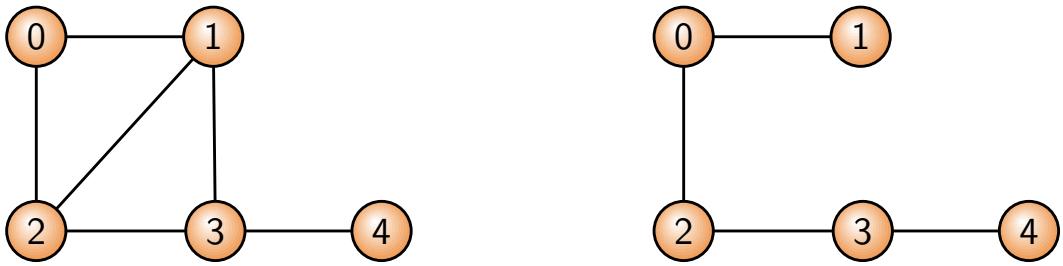
2025/2

Sumário

1. Árvore geradora mínima em grafos
   - Introdução
   - Algoritmo de Prim
   - Algoritmo de Kruskal
   - Exercícios

Árvore geradora mínima

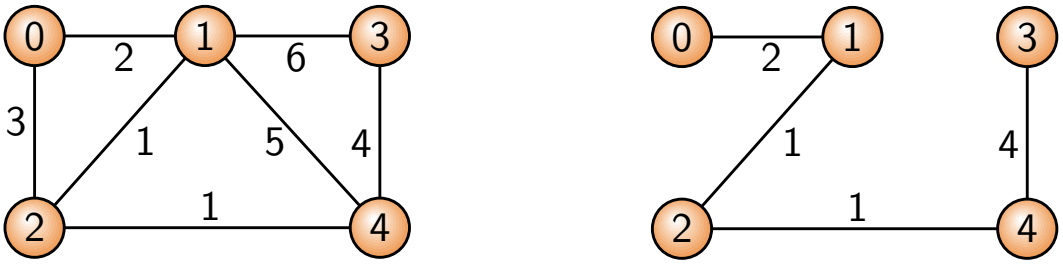
- Dado um grafo $G(V, A)$, com $n \equiv |V|$ vértices, uma **árvore geradora** (em inglês, *spanning tree*) é um subgrafo $T(V, A_T)$ contendo todos os n vértices do grafo original, além de um subconjunto $A_T \subseteq A$ com $|A_T| = n - 1$ arestas que permitam a conexão entre tais vértices.
  - O grafo G deve ser **conexo**;
  - Tal subgrafo T é **acíclico** – daí o termo **árvore**.



Um grafo (esq.) e uma possível árvore geradora (dir.)

Se o grafo, além de conexo, é ponderado, sua **árvore geradora mínima (AGM)** – em inglês, *minimum spanning tree* – é a árvore geradora cujo **custo** (em termos da soma dos pesos das arestas) é o menor possível.

- Além do grafo ser conexo e ponderado, deve ser não-direcionado (i.e. não pode ser dígrafo)
  - Para quaisquer dois vértices na AG, existe um caminho que os une – situação em geral possível somente sem direcionamento.
- Como todos os vértices estão conectados entre si, o cálculo da árvore geradora independe do vértice inicial.



Um grafo ponderado (esq.) e sua árvore geradora mínima (dir.)

Exemplos de uso

- Minimizar custos em redes de comunicação, eletricidade, transporte, etc.
  - Garante a interligação de todos os pontos com o menor custo possível em termos de circuitos integrados.
- Otimização de Infraestrutura em redes de computadores, rodovias, tubulações e distribuição de energia;
- Análise de *clusters* (agrupamentos) em aprendizado de máquina;
- Segmentação de imagens;
- Aplicações em finanças (correlações entre diferentes ativos); etc.

Resolução

- O problema da árvore geradora pode ser resolvido usando uma estratégia que constrói a árvore incrementalmente;
  1. A partir de um grafo $G(V, A)$, começa-se com uma árvore T, com um vértice e nenhuma aresta (**grafo trivial**);
  2. Define-se a **franja** de T como o conjunto de todas as arestas (u, v) de A tais que $u \in T$ mas $v \notin T$ (i.e. arestas conectando um vértice na árvore com outro fora).

Passo geral

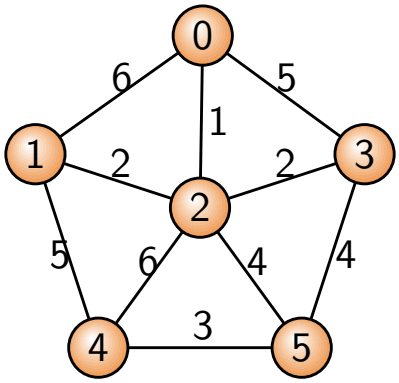
enquanto a franja não estiver vazia:
    tome uma aresta (u, v) da franja e troque T por $T + (u, v)$

  3. Ao final, processo termina com franja vazia.
     - ★ Se $|V_T| = |V|$, o grafo é conexo. Caso contrário, V_T é isolado e encontrou-se uma componente conexa.

- Existem dois algoritmos clássicos para obter soluções ótimas para a **árvore geradora mínima** – quando a aresta da franja é a de menor peso:
  - Algoritmo de Prim
  - Algoritmo de Kruskal
- Ambos são **gulosos**: a cada etapa, fazem a escolha melhor possível no momento, dentre as várias possibilidades.
- Diferença entre eles está na regra usada para encontrar as arestas que farão parte da árvore.

Grafo ponderado para teste I

A fim de demonstrar os algoritmos de árvore geradora mínima, utilizamos o seguinte grafo ponderado conexo:



Fonte: adaptado de BACKES, A. R. Algoritmos e Estruturas de Dados em Linguagem C. https://www.sistemas.ufu.br/biblioteca-gateway/minhabiblioteca/9788521638315

A criação do mesmo usa a seguinte implementação, considerando-se um máximo de 5 conexões por vértice:

Grafo ponderado para teste II

main.c – grafo de teste

```c
#include <stdio.h>
#include <stdlib.h>
#include "Grafo.h"
int main(){
  int eh_digrafo = 0;
  Grafo* gr = cria_Grafo(6, 6, 1);

  insereAresta(gr, 0, 1, eh_digrafo, 6);
  insereAresta(gr, 0, 2, eh_digrafo, 1);
  insereAresta(gr, 0, 3, eh_digrafo, 5);
  insereAresta(gr, 1, 2, eh_digrafo, 2);
  insereAresta(gr, 1, 4, eh_digrafo, 5);
  insereAresta(gr, 2, 3, eh_digrafo, 2);
  insereAresta(gr, 2, 4, eh_digrafo, 6);
  insereAresta(gr, 2, 5, eh_digrafo, 4);
  insereAresta(gr, 3, 5, eh_digrafo, 4);
  insereAresta(gr, 4, 5, eh_digrafo, 3);
```


Grafo ponderado para teste III

```c

  // buscar pela AGM aqui

  libera_Grafo(gr);
  system("pause");
  return 0;
}
```

Sumário

1 Árvore geradora mínima em grafos
- Introdução
- Algoritmo de Prim
- Algoritmo de Kruskal
- Exercícios

Algoritmo de Prim

- Parte de um vértice inicial;
- A cada iteração, busca a aresta de **menor peso** que conecte um vértice da árvore a outro que ainda não esteja na mesma;
- O novo vértice é adicionado a árvore, e repete-se o processo;
- O algoritmo prossegue até que:
  - Todos os vértices façam parte da árvore;
  - Não se pode encontrar uma aresta que satisfaça essa condição (**grafo desconexo**).

- A função que obtém a árvore recebe três parâmetros:
  1. o grafo
  2. o vértice inicial da busca
  3. um *array* que indicará, para cada vértice, qual o seu **pai**, ou seja, o vértice anterior na AGM – correspondendo ao **resultado** da busca.
     - ★ o vértice do *array* cujo pai é seu próprio índice representa a **raiz** da árvore.

main.c – algoritmo de Prim

```
17    .
18    .
19    insereAresta(gr, 4, 5, eh_digrafo, 3);
20
21    int pai[6];
22    arvoreGeradoraMinimaPrim_Grafo(gr, 0, pai);
23    .
24    .
```


grafo.c – algoritmo de Prim

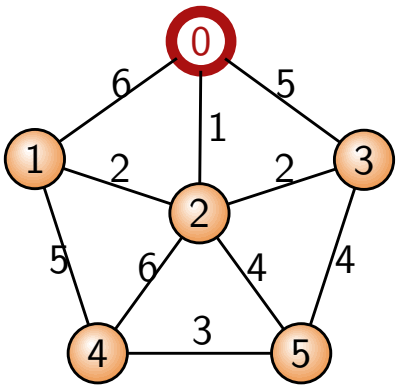
```
103 void arvoreGeradoraMinimaPrim_Grafo (Grafo *gr, int
      orig, int *pai) {
104   int i, j, dest, primeiro, NV = gr->nro_vertices;
105   double menorPeso;
106
107   // vértices inicialmente não têm pai (exceto orig)
108   for(i=0; i < NV; i++)
109     pai[i] = -1; // sem pai
110   pai[orig] = orig; // vértice inicial: raiz da AGM
111
112   while (1){
113     primeiro = 1;
114     // percorre todos os vértices, e
115     // procura aresta de menor peso ligando um vértice
116     // já na árvore a outro ainda não inserido
117     for(i=0; i < NV; i++) {
118       if (pai [i] != -1){ // vértice já inserido:
119         // percorre-se seus vizinhos
```

```
120         for(j=0; j<gr->grau[i]; j++) {
121           int vizinho = gr->arestas[i][j];
122           if (pai[vizinho] == -1){ // não inserido
123             // busca aresta com menor peso
124             if (primeiro) {
125               menorPeso = gr->pesos[i][j];
126               orig = i;
127               dest = vizinho;
128               primeiro = 0;
129             } else {
130               if (menorPeso > gr->pesos[i][j]){
131                 menorPeso = gr->pesos[i][j];
132                 orig = i;
133                 dest = vizinho;
134               }
135             }
136           }
137         }
138       }
139     }
```



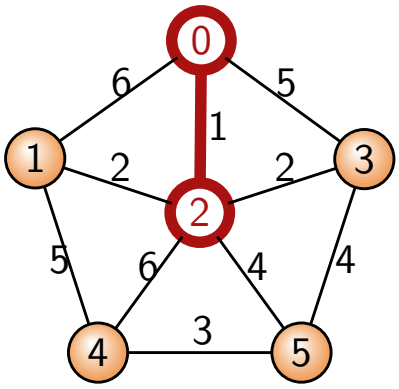
```
140     if (primeiro == 1)
141       break;
142
143     pai[dest] = orig;
144   }
145 }
```

Passo a passo



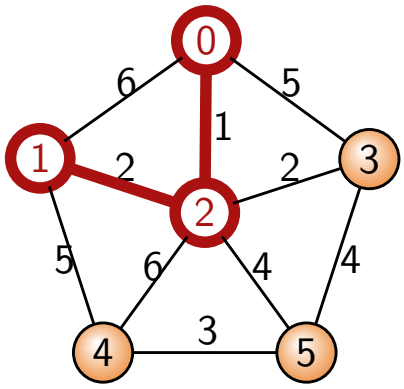
	pai
0	0
1	-1
2	-1
3	-1
4	-1
5	-1

Inicia o cálculo com o vértice 0.
Atribui seu próprio índice como pai.
O restante dos vértices recebem pai igual a -1 (sem pai)



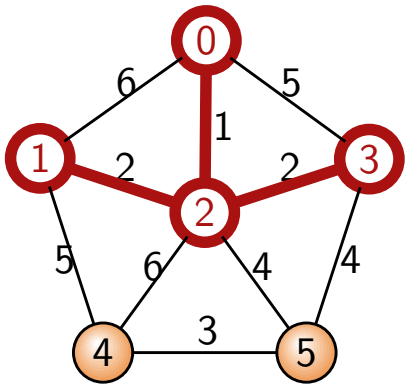
	pai
0	0
1	-1
2	0
3	-1
4	-1
5	-1

Procura nos vértices com pai por um vizinho sem pai e com menor peso: vértice 2.
Atribui vértice 0 como pai do vértice 2.



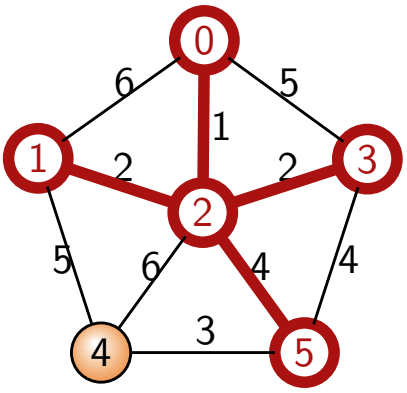
	pai
0	0
1	2
2	0
3	-1
4	-1
5	-1

Procura nos vértices com pai por um vizinho sem pai e com menor peso: vértice 1.
Atribui vértice 2 como pai do vértice 1.



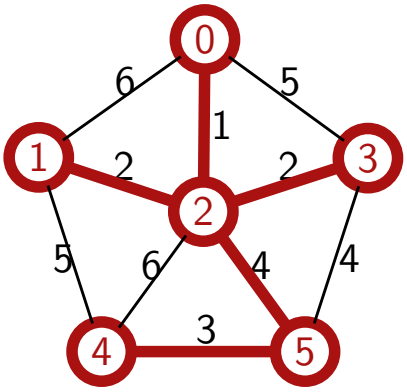
	pai
0	0
1	2
2	0
3	2
4	-1
5	-1

Procura nos vértices com pai por um vizinho sem pai e com menor peso: vértice 3.
Atribui vértice 2 como pai do vértice 3.



	pai
0	0
1	2
2	0
3	2
4	-1
5	2

Procura nos vértices com pai por um vizinho sem pai e com menor peso: vértice 5.
Atribui vértice 2 como pai do vértice 5.



	pai
0	0
1	2
2	0
3	2
4	5
5	2

Procura nos vértices com pai por um vizinho sem pai e com menor peso: vértice 4.
Atribui vértice 5 como pai do vértice 4. **Fim do cálculo.**

Complexidade

- Considerando um grafo $G(V, A)$, onde $|V|$ é o número de vértices e $|A|$ é o número de arestas, a complexidade no pior caso é $O(|V| \times |A|)$.
  - Como $|A|$ é proporcional a $|V|^2$, o custo é $O(|V|^3)$.
- A eficiência depende da forma usada para procurar a aresta de menor peso.
  - Usando uma **fila de prioridades** para as arestas, o custo pode ser reduzido para $O(|A| log |V|)$. **Sugestão de leitura:** https://www.ime.usp.br/~pf/analise_de_algoritmos/aulas/MST-prim.html#sec:prim3

Sumário

1. Árvore geradora mínima em grafos
   - Introdução
   - Algoritmo de Prim
   - Algoritmo de Kruskal
   - Exercícios

Algoritmo de Kruskal

- Considera cada vértice como uma árvore independente;
- A cada iteração, o algoritmo procura a aresta de **menor peso** que conecta duas árvores diferentes; Os vértices das árvores selecionadas passam a fazer parte de uma mesma árvore;
- O algoritmo prossegue até que:
  - Todos os vértices façam parte da árvore;
  - Não se pode encontrar uma aresta que satisfaça essa condição (**grafo desconexo**).

Diferença entre ambos os algoritmos

- O algoritmo de Prim se inicia com um vértice e cresce uma única árvore a partir dele;
- O algoritmo de Kruskal constrói uma **floresta** (várias árvores) ao longo do tempo, e são unidas ao final do processo.

- A função que obtém a árvore recebe três parâmetros:
  1. o grafo
  2. o vértice inicial da busca
  3. um *array* que indicará, para cada vértice, qual o seu **pai**, ou seja, o vértice anterior na AGM – correspondendo ao **resultado** da busca.
     - ★ o vértice do *array* cujo pai é seu próprio índice representa a **raiz** da árvore.

main.c – algoritmo de Kruskal

```
.
.
insereAresta(gr, 4, 5, eh_digrafo, 3);

int pai[6];
arvoreGeradoraMinimaKruskal_Grafo(gr, 0, pai);
.
.
```


grafo.c – algoritmo de Kruskal

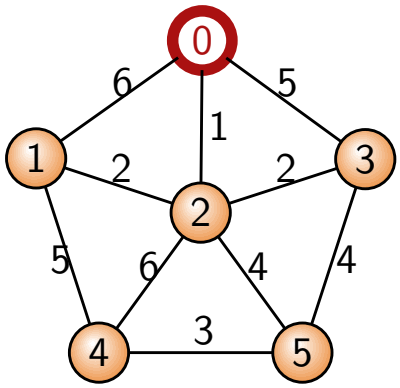
```
void arvoreGeradoraMinimaKruskal_Grafo (Grafo *gr, int
    orig, int *pai) {
  int i, j, dest, primeiro, NV = gr->nro_vertices;
  double menorPeso;
  int *arv = (int*) malloc(NV*sizeof(int));

  // cada vértice inicialmente é uma árvore sem pai
  for(i=0; i < NV; i++) {
    pai[i] = -1; // sem pai
    arv[i] = i;
  }
  pai[orig] = orig; // vértice inicial: raiz da AGM

  while (1){
    primeiro = 1;
    // percorre todos os vértices, e
    // Procura aresta de menor peso ligando
    // árvores diferentes
```

```
164     for(i=0; i < NV; i++) {
165       for(j=0; j<gr->grau[i]; j++) {
166         int vizinho = gr->arestas[i][j];
167         if (arv[i] != arv[vizinho]){ // não inserido
168           // busca aresta com menor peso
169           if (primeiro) {
170             menorPeso = gr->pesos[i][j];
171             orig = i;
172             dest = vizinho;
173             primeiro = 0;
174           } else {
175             if (menorPeso > gr->pesos[i][j]){
176               menorPeso = gr->pesos[i][j];
177               orig = i;
178               dest = vizinho;
179             }
180           }
181         }
182       }
183     }
```



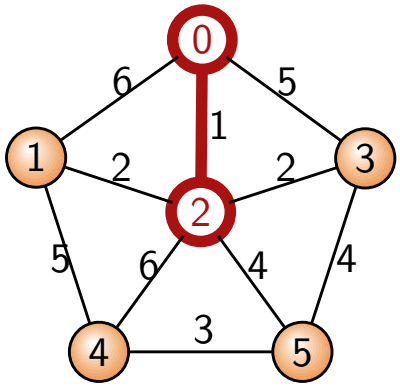
```
184     if (primeiro == 1)
185       break;
186
187     if (pai[orig] == -1)
188       pai[orig] = dest;
189     else
190       pai[dest] = orig;
191
192     // une as duas árvores da floresta selecionada
193     for(i=0; i < NV; i++)
194       if (arv[i] == arv[dest])
195         arv[i] = arv[orig];
196   }
197   free(arv);
198 }
```

Passo a passo



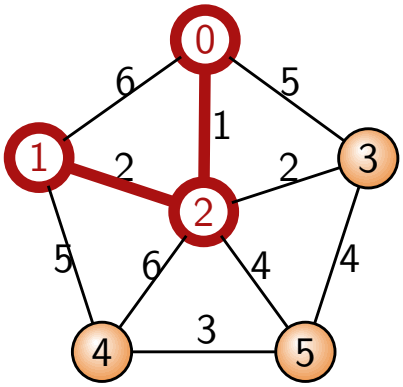
	pai	arv
0	0	0
1	-1	1
2	-1	2
3	-1	3
4	-1	4
5	-1	5

Inicia o cálculo com o vértice 0.
Atribui seu próprio índice como pai.
O restante dos vértices recebem pai igual a -1 (sem pai)
Inicializa a árvore com o índice do vértice.



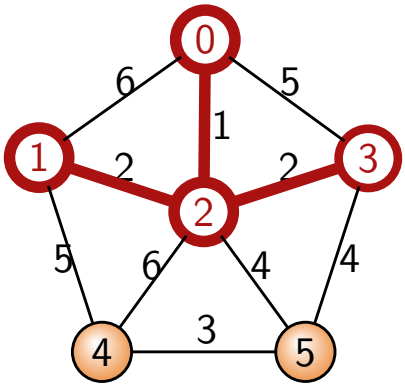
	pai	arv
0	0	0
1	-1	1
2	0	0
3	-1	3
4	-1	4
5	-1	5

Procura a aresta com menor peso conectando vértices com árvores diferentes: vértices 0 e 2.
Atribui vértice 0 como pai do vértice 2.
Todos que possuem árvore igual ao vértice 2 passam a ter árvore igual ao vértice 0.



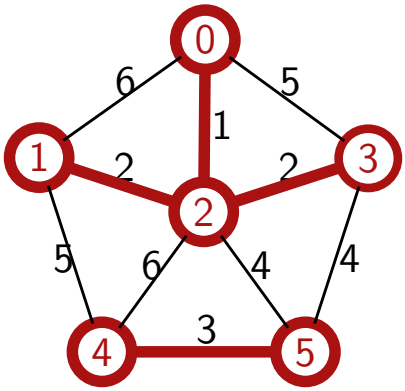
	pai	arv
0	0	1
1	2	1
2	0	1
3	-1	3
4	-1	4
5	-1	5

Procura a aresta com menor peso conectando vértices com árvores diferentes: vértices 1 e 2.
Atribui vértice 2 como pai do vértice 1.
Todos que possuem árvore igual ao vértice 2 passam a ter árvore igual ao vértice 1.



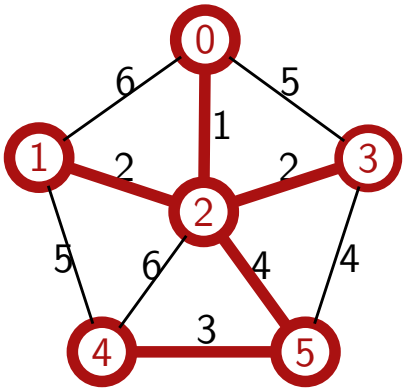
	pai	arv
0	0	1
1	2	1
2	0	1
3	2	1
4	-1	4
5	-1	5

Procura a aresta com menor peso conectando vértices com árvores diferentes: vértices 2 e 3.
Atribui vértice 2 como pai do vértice 3.
Todos que possuem árvore igual ao vértice 3 passam a ter árvore igual ao vértice 1.



	pai	arv
0	0	1
1	2	1
2	0	1
3	2	1
4	5	4
5	-1	4

Procura a aresta com menor peso conectando vértices com árvores diferentes: vértices 4 e 5.
Atribui vértice 2 como pai do vértice 3.
Todos que possuem árvore igual ao vértice 3 passam a ter árvore igual ao vértice 1.



	pai	arv
0	0	1
1	2	1
2	0	1
3	2	1
4	5	1
5	2	1

Procura a aresta com menor peso conectando vértices com árvores diferentes: vértices 2 e 5.
Atribui vértice 2 como pai do vértice 5.
Todos que possuem árvore igual ao vértice 4 passam a ter árvore igual ao vértice 1. **Fim do cálculo.**

Complexidade

- Considerando um grafo $G(V, A)$, onde $|V|$ é o número de vértices e $|A|$ é o número de arestas, a complexidade no pior caso é $O(|V| \times |A|)$.
  - Como $|A|$ é proporcional a $|V|^2$, o custo é $O(|V|^3)$.
- A eficiência depende da forma usada para procurar a aresta de menor peso.
  - Usando uma estrutura de dados união-busca (*Union & Find*), o custo pode ser reduzido para $O(|A| log|V|)$ – veja https://www.ime.usp.br/~pf/analise_de_algoritmos/aulas/MST-kruskal.html#sec:implementation.

Sumário

1. Árvore geradora mínima em grafos
   - Introdução
   - Algoritmo de Prim
   - Algoritmo de Kruskal
   - Exercícios

Exercícios I

1. (CORMEN, 2024) Forneça uma implementação simples do algoritmo de Prim, que seja executada no tempo $O(|V|^2)$, quando o grafo $G(V, E)$ é representado como uma matriz de adjacências.
2. Mostre que um grafo G tem uma árvore geradora se, e somente se, G é conexo. (Sugestão: Use o algoritmo da árvore geradora.)
3. Escreva um algoritmo que conte o número de componentes de um grafo. (Sugestão: Use o algoritmo da árvore geradora.)

Referências

1. BACKES, A. R. **Algoritmos e Estruturas de Dados em Linguagem C**. 1. ed. Rio de Janeiro: LTC, 2023. Disponível em: https://www.sistemas.ufu.br/biblioteca-gateway/minhabiblioteca/9788521638315.
2. CORMEN, T. H.; et al. **Algoritmos**. 4. ed. Rio de Janeiro: GEN LTC, 2024. Disponível em: https://www.sistemas.ufu.br/biblioteca-gateway/minhabiblioteca/9788595159914.

Os materiais de parte desta seção foram gentilmente cedidos por André R. Backes (UFSCar); Bruno Travençolo e Christiane Brasil (FACOM/UFU).
TAD grafo: disponível em:
https://www.facom.ufu.br/~backes/wordpress/ProjGrafo.zip.
Acesso em: 22/jan/2025.
Adaptações: Renato Pimentel, FACOM/UFU