GERAÇÃO DE COGRAFOS COM ATRASO LINEAR

Átila Arueira Jones
Universidade Federal Fluminense
Instituto Federal Sudeste de Minas Gerais
Niterói - RJ
atilajones@id.uff.br

Fábio Protti
Universidade Federal Fluminense
Niterói - RJ
fabio@ic.uff.br

Renata Raposo Del-Vecchio
Universidade Federal Fluminense
Niterói - RJ
renata@vm.uff.br

RESUMO

Os cografos sempre foram alvos de pesquisa em áreas como coloração, otimização, cliques, teoria espectral. Neste artigo trabalhamos neste ponto, onde apresentamos um algoritmo capaz de gerar todos os cografos (sem rotulação), mais especificamente usamos a representação em coárvores para desenvolver um procedimento capaz de gerar, sem isomorfismos, todos os cografos com n vértices. Além disso provamos que o tempo de geração entre dois cografos consecutivos é feito em $O(n)$.

PALAVRAS CHAVE. Cografos, Geração, Algoritmos

Área principal. Teoria e Algoritmos em Grafos (TAG)

ABSTRACT

The cographs have always been research targets in areas such as coloring, optimization, cliques, spectral theory. In this article, we present an algorithm to generate all cographs (unlabeled), more specifically, we use the representation of cotrees to develop a procedure that generates, without isomorphisms, all cographs with n vertices. We also prove that the generation time between two consecutive cographs is done in $O(n)$.

KEYWORDS. Cographs, Generation, Algorithms

Main area. Theory and Algorithms in Graphs (TAG)

1. Introdução

Os cografos foram descobertos independentemente por diversos autores desde 1970 e usualmente são definidos como grafos livres de P_4, segundo equivalência provada em [Corneil et al., 1981]. Mas sua definição original é apresentada em forma recursiva: o grafo trivial é cografo, o complementar de cografo é cografo, a união de cografos é cografo.

Vale lembrar que o **join** entre dois grafos $G_1(V_1, E_1)$ e $G_2(V_2, E_2)$ é o grafo $G_1 \vee G_2$ cujo conjunto de vértices é $V_1 \cup V_2$ e arestas $E_1 \cup E_2 \cup \{xy; x \in E_1 \text{ e } y \in E_2\}$, tal operação também é chamada de **união complementar**, pois vale $\overline{G_1 \cup G_2} = \overline{G_1} \vee \overline{G_2}$. Então um cografo pode ser obtido por sucessivas operações de *join* e *união* entre cografos. Tal recurso foi utilizado em [Corneil et al., 1984] para representação de cografo através de uma árvore, chamada **coárvore**, cujas folhas são os vértices do grafo e os vértices internos possuem ao menos dois filhos e representam operações de *join* ou *união*, nomeados por $tipo$-1 e $tipo$-0, respectivamente. Além disso a coárvore deve ser escrita de forma que os vértices que compõem um caminho possuem seus tipos alternados, o que garante que cada cografo tenha somente uma coárvore associada, a menos de permutação entre vértices, por outro lado temos que cada coárvore é referente a um único cografo.

Como os cografos são identificados por árvores, vamos fixar algumas notações acerca de uma árvore enraizada T: a raiz será denotada por $raiz(T)$; dado um vértice v (diferente da raiz) denotamos P_v o único caminho de v até a raiz; os vértices de P_v (exceto v) são ditos **ancestrais** de v em T, onde o antecessor imediato de v em P_v é chamado **pai** de v e denotado por $pai_T(v)$, fixamos $pai(raiz(T)) := \textbf{\textit{null}}$; para cada vértice interno v definimos seus **filhos** pelo conjunto $filhos_T(v) = \{u \in V(T); pai_T(u) = v\}$ e **irmandade de** v por $I_T(v) = \{filhos_T(pai_T(v))\}$, para $v \neq raiz(T)$ e fixamos $I_T(raiz(T)) = \{raiz(T)\}$; denotaremos por $T(v)$ a subárvore induzida por T cuja raiz é v. Em casos livres de ambiguidade ocultaremos o índice T das notações acima. Além disso denotaremos o isomorfismo entre dois grafos G_1 e G_2 por $G_1 \equiv G_2$.

Em [Corneil et al., 1981] é provado também que dadas duas folhas v e w de uma coárvore, o vértice mais próximo a ambos e que seja comum a P_v e P_w é $tipo$-1 se, e somente se, os respectivos vértices no cografo são adjacentes. Então um cografo é conexo se, e somente se, a raiz da coárvore é $tipo$-1.

O trabalho [Ravelomanana e Thimonier, 2001] faz uma estimativa assintótica para o número de cografos com um determinado número de vértices. Neste artigo utilizamos a representação de cografo através da coárvore para desenvolver o algoritmo 3.4 capaz de gerar sem repetição todos os cografos com n vértices, sem rotulação. Além disso o procedimento fornecerá, naturalmente, o número exato de cografos com n vértices.

Nesta primeira seção introdutória lembramos as definições necessárias para o desenvolvimento deste trabalho, fixamos algumas notações bem como resumimos o objetivo principal. Toda teoria e definições desenvolvidas nas duas seções seguintes (2 e 3) são contribuições deste trabalho. Mais especificamente, na seção 2 definiremos ordenações para *vértices* e *árvores* através de partições de inteiro, seguidos de alguns resultados obtidos. Na seção 3 desenvolveremos os algoritmos que compõem o algoritmo principal do artigo, seguidos de resultados acerca da complexidade e corretude destes. Por fim, na seção 4 é feita a conclusão do trabalho.

2. Ordenação de Vértices e Árvores

Vimos que numa coárvore qualquer caminho possui alternância nos tipos do vértice, sendo então determinados unicamente pelo *tipo* da raiz. Defina $\mathbb{T}$ como o conjunto das árvores enraizadas onde cada vértice interno possui ao menos dois filhos, então cada elemento de $\mathbb{T}$ se refere à exatamente dois cografos distintos: uma com raiz $tipo$-1 e outra $tipo$-0, salvo o cografo com apenas um vértice. Por outro lado, cada cografo está associado a uma única árvore de $\mathbb{T}$, a menos de uma permutação de vértices da mesma irmandade. Então nos deparamos com a necessidade de introduzir uma forma "padrão"para a configuração dessa árvore, que será feito nesta seção.

Seja $T \in \mathbb{T}$, para cada vértice v de T definimos $l(v)$ como o número de folhas da árvore $T(v)$, chamado **número de folhas induzidas por** v, se v é uma folha definimos $l(v) = 1$. Uma

árvore é dita **rotulada** se é conhecido o valor $l(v)$ para cada vértice v, o que decorre o fato a seguir.

Fato 2.1. *Para cada nó interno v vale $l(v) = \sum_{w \in filhos(v)} l(w)$.*

Exemplo 2.1. *A árvore abaixo é rotulada. O vértice em destaque será tratado na seção 3.*

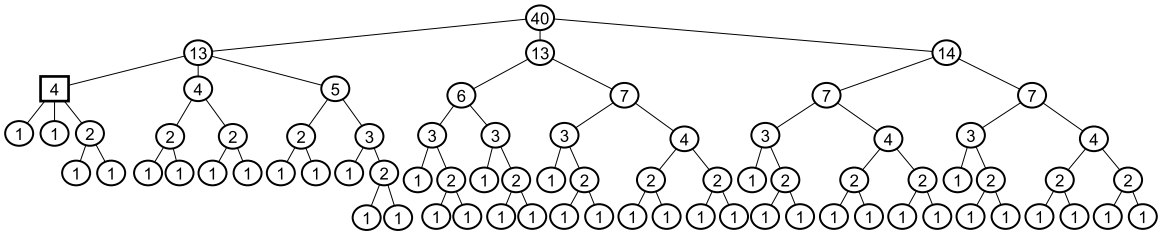

Figura 1: árvore rotulada

Podemos particionar $\mathbb{T} = \bigcup_{n=1}^{\infty} \mathbb{T}_n$, onde $\mathbb{T}_n = \{T \in \mathbb{T};\ l(raiz(T)) = n\}$, desta forma cada cografo com n vértices pode ser representado por uma árvore de $\mathbb{T}_n$. A proposição abaixo, facilmente provada por indução finita, será útil no cálculo de complexidades dos algoritmos que desenvolveremos.

Proposição 2.1. *Para todo n, o número de vértices de uma árvore de $\mathbb{T}_n$ é no máximo $2n - 1$.*

O fato 2.1 e proposição anterior garantem que a rotulação de uma árvore de $\mathbb{T}_n$ pode ser feita em tempo $O(n)$.

O conceito de *Partição de Inteiros* foi introduzido por Euler. Neste trabalho faremos uma variação da definição, assumindo que uma partição deve possuir ao menos dois elementos, conforme descrito abaixo.

Definição 2.1. *Seja $n \geq 2$ um inteiro positivo, uma **partição do inteiro** n é uma sequência não-decrescente de números inteiros positivos $(a_i)_k := (a_1, a_2, a_3, \ldots, a_k)$, tal que $\sum_{i=1}^{k} a_i = n$ e $k \geq 2$. O conjunto das partições de n é denotado por $Part(n)$.*

E então utilizaremos a ordenação *lexicográfica* em partições.

Definição 2.2. *Dados $a, b \in Part(n)$, onde $a = (a_i)_k$ e $b = (b_i)_m$, a ordenação entre tais partições é dada pela ordenação lexicográfica. Isto é, tome $j = min\{i;\ a_i \neq b_i \text{ e } 1 \leq i \leq min\{k, m\}\}$, caso exista, e defina a relação de ordem entre a e b pela relação entre os inteiros a_j e b_j, se j não existir, significa que $a = b$.*

Exemplo 2.2. *Os elementos de $Part(5)$ listados em ordem crescente são: $(1,1,1,1,1)$, $(1,1,1,2,2)$, $(1,1,3)$, $(1,2,2)$, $(1,4)$ e $(2,3)$.*

Fato 2.2. *O menor elemento de $Part(n)$ é $a = (1, \ldots, 1)$ e o maior é $b = \left(\lfloor \frac{n}{2} \rfloor, \lceil \frac{n}{2} \rceil\right)$.*

Como a definição 2.2 baseia-se na comparação de inteiros, então vale a tricotomia de ordem para partições. A seguinte definição relaciona a partição de um inteiro com a distribuições das folhas numa árvore, cuja definição é bem construída de acordo com o fato 2.1.

Definição 2.3. *Seja $T \in \mathbb{T}$ e v um vértice interno de T. Escrevendo os filhos de v na ordem $v_1, \ldots, v_k$ de forma que $l(v_1), l(v_2), \ldots, l(v_k)$ forme uma sequência não-decrescente, a sequência $(l(v_1), \ldots, l(v_k))$ é chamada **partição induzida** por v.*

Definição 2.4. *Seja $T \in \mathbb{T}$, a relação de ordem entre v, w vértices de T é definida de forma recursiva:*

- *CASO 1: Se* $l(v) < l(w)$ *então* $v < w$*;*
- *CASO 2: Se* $l(v) = l(w) = 1$ *então* $v \sim w$*;*
- *CASO 3: Se* $l(v) = l(w)$, $l(v) \neq 1$*:*

  Tome $(a_i)_k$ *a partição induzida por* v *e* $filhos(v) = \{v_1, \ldots, v_k\}$*. Analogamente* $(b_i)_m$ *por* w*, onde* $filhos(w) = \{w_1, \ldots, w_m\}$*. Temos três subcasos a considerar:*

  CASO 3.1: Se $(a_i)_k < (b_i)_m$ *então* $v < w$*;*

  CASO 3.2: Se $(a_i)_k = (b_i)_m$ *(consequentemente* $k = m$*):*

  CASO 3.2a: Se $v_i \sim w_i$ *para todo* $1 \leq i \leq k$ *então* $v \sim w$*;*

  CASO 3.2b: Senão $j = min\{i; v_i \not\sim w_i\}$*. Se* $v_j < w_j$ *então* $v < w$*, senão* $v > w$*.*

No caso $v \sim w$ *diremos que* v *e* w *são* ***equivalentes****, o que permite escrever a classe de equivalência* $[v] = \{u \in V(T); v \sim u\}$*. A ordenação definida acima é feita entre duas classes, onde escrevemos* $[v] < [w]$ *na forma simplificada* $v < w$*. Além disso denotaremos que* $v \sim w$ *ou* $v < w$ *simplesmente por* $v \leq w$ *(analogamente* $v \geq w$*).*

Vale observar que a definição pode ser estendida para vértices de árvores distintas, basta considerar uma terceira árvore T cujos filhos da raiz são os vértices a serem comparados.

Fato 2.3. *(**Tricotomia**) Pela tricotomia em partições de inteiro, dados* v, w *vértices de* $T \in \mathbb{T}_n$*, apenas um dos casos é válido:* $v \sim w$*,* $v < w$ *ou* $v > w$*.*

No decorrer do texto ao usarmos o símbolo de igualdade entre vértices estaremos nos referindo, de fato, ao mesmo vértice. Por outro lado a definição 2.4 fornece a ideia intuitiva de que dois vértices são equivalentes quando as respectivas subárvores induzidas são idênticas, o que é garantida pelo lema provado a seguir.

Lema 2.1. *Dados* v, w *vértices de* $T \in \mathbb{T}$*, então* $T(v) \equiv T(w)$ *se, e somente se,* $v \sim w$*.*

Demonstração. A demonstração é trivial para o caso em que v é uma folha. Suponha que é v vértice interno de T e considere $(a_i)_k$ a sua partição induzida, onde $filhos(v) = \{v_1, \ldots, v_k\}$ e $a_i = l(v_i)$ para $i \in \{1, \ldots, k\}$. Analogamente tome $(b_i)_m$ induzida por w e $filhos(w) = \{w_1, \ldots, w_m\}$.

Suponha que $T(v) \equiv T(w)$, então $k = l$ e $(a_i)_k = (b_i)_k$, onde vale ainda $\sum a_i = \sum b_i = l(v) = l(w) = N$. Vamos provar por indução sobre N que $v \sim w$.

O resultado vale para o caso base $N = 2$, pois $Part(2) = \{(1, 1)\}$. Suponha que o resultado vale para valores menores que algum N. Como $T(v) \equiv T(w)$ então $T(v_i) \equiv T(w_i)$, para $i \in \{1, \ldots, k\}$, então $l(v_i) = l(w_i) < N$ e pela hipótese indutiva concluímos $v_i \sim w_i$. Logo $v \sim w$.

Reciprocamente suponha que $v \sim w$, então $N = l(v) = l(w)$. Vamos novamente provar por indução sobre N que $T(v) \equiv T(w)$. O resultado vale para o caso base onde $N = 2$ e suponha que vale para valores menores que algum N. Então pela hipótese e tricotomia, temos por definição que $v_i \sim w_i$ para todo $i \in \{1, \ldots, k = m\}$, mas $l(v_i) = l(w_i) < N$ então pela hipótese indutiva vale $T(v_i) \equiv T(w_i)$, logo $T(v) \equiv T(w)$. □

Definição 2.5. *Sejam* $T \in \mathbb{T}$ *e* v *vértice de* T*. A irmandade* $I_T(v) = \{v_1, \ldots, v_k\}$ *é dita* ***ordenada*** *se* $v_1 \leq \ldots \leq v_k$*. Se todas as irmandades de uma árvore estão ordenadas, diremos que a árvore está ordenada.*

Exemplo 2.3. *A árvore da figura 1 está ordenada.*

É conhecido que cada cografo é associado a uma única coárvore, a menos de permutação. Na próxima proposição provamos que cada cografo está associado a uma única coárvore ordenada.

Proposição 2.2. *Cada cografo está associado a uma única árvore ordenada em* $\mathbb{T}$*.*

Demonstração. Seja G um cografo e tome T sua coárvore associada, em [Corneil et al., 1981] é provado que T é única a menos de isomorfismos. Considere T' e T'' árvore ordenadas e ambas isomorfas a T. Como os vértices de T' e T'' possuem uma ordem de disposição dos vértices, então são árvores iguais a menos de permutação de vértices equivalentes de uma mesma irmandade. Porém o lema 2.1 assegura que essa permutação geram subárvores idênticas, com exatamente a mesma disposição de vértices. Logo T' e T'' são iguais. □

E então introduzimos uma ordenação total para as árvores de $\mathbb{T}_n$.

Definição 2.6. *Sejam T_1 e T_2 árvores em $\mathbb{T}$. Definimos $T_1 < T_2$ quando $raiz(T_1) < raiz(T_2)$ e $T_1 \sim T_2$ no caso de equivalência entre as raízes.*

Fato 2.4. *O menor elemento de $\mathbb{T}_n$ é a árvore cuja raiz induz a partição $(1, \ldots, 1) \in Part(n)$ (veja o fato 2.2). Tal árvore está associada ao cografo completo K_n e seu complementar.*

3. Geração dos cografos

Assegurado pela proposição 2.2, a geração dos cografos será feita através da geração dos elementos de $\mathbb{T}_n$, cujo primeiro elemento é a menor árvore do conjunto e em seguida geramos a árvore imediatamente maior que a antecessora, até alcançar a maior árvore do conjunto. A noção de elemento imediatamente maior é formalizada na definição abaixo.

Definição 3.1. *Seja $\mathbb{X}$ um conjunto finito não-vazio munido de ordenação total, onde vale a tricotomia. Dado $a \in \mathbb{X}$, diremos que b é imediatamente maior que a em $\mathbb{X}$ se $b \in \mathbb{X}$ e $b > a$, onde para qualquer $c \in \mathbb{X}$ tal que $c > a$ então $c \geq b$. Naturalmente se a não possui elemento imediatamente maior em $\mathbb{X}$, então a é o maior elemento do conjunto.*

Algoritmo 3.1 Partição imediatamente maior

Input: $(a_i)_k \in Part(n)$
Output: $(b_i)_m$ imediatamente maior que $(a_i)_k$ em $Part(n)$

procedure $ParticaoSeguinte((a_i)_k)$
  $n \leftarrow \sum_{i=1}^{k} a_i$
  if $a_1 \neq \lfloor n/2 \rfloor$ **then**
    if $a_k - a_{k-1} \leq 1$ **then return** $(a_1, a_2, \ldots, a_{k-2}, a_{k-1} + a_k)$;
    else ▷ $a_k - a_{k-1} > 1$
      $a_{k-1} \leftarrow a_{k-1} + 1$;
      $a_k \leftarrow a_k - 1$;
      Tome q e r o quociente e resto da divisão a_k por a_{k-1};
      if $q > 1$ **then return** $(a_1, \ldots, a_{k-2}, \underbrace{a_{k-1}, \ldots, a_{k-1}}_{q \text{ vezes}}, (a_{k-1} + r))$;
      else return $(a_1, \ldots, a_{k-2}, a_{k-1}, a_k)$;
    end if
  else
    if $n \neq 3$ **then return *null***
    else ▷ $n = 3$ caso especial
      if $a_2 = \lceil n/2 \rceil$ **then return *null*** ;
      else return $(1, 2)$;
    end if
  end if
end function

Proposição 3.1. *Se **a** $\in Part(n)$, o retorno do algoritmo 3.1 é a partição imediatamente maior que **a** em $Part(n)$. Caso o retorno seja **null** então **a** é o maior elemento de $Part(n)$.*

Demonstração. Seja $a = (a_i)_k \in Part(n)$. Os casos $n = 2$ e $n = 3$ são triviais.

Pela definição de partição, necessariamente $a_1 \leq \lfloor \frac{n}{2} \rfloor$. Suponha que $n > 3$ e $a_1 < \lfloor \frac{n}{2} \rfloor$. Vamos separar a prova em casos:

- **CASO 1**: $a_k - a_{k-1} \leq 1$.

  Então $b := (b_i)_{k-1} = (b_1, b_2, \ldots, b_{k-1})$ é a partição retornada pelo algoritmo, onde $b_i = a_i$ para $i \in \{1, \ldots, k-2\}$ e $b_{k-1} = a_{k-1} + a_k$.

  É claro que $b > a$ e que $b_{k-1} \in Part(n)$.

  Vamos provar que b é a sequência imediatamente maior. Seja $c = \{c_i\}_m \in Part(n)$ tal que $c > a$, queremos provar que $c \geq b$. Por definição $\exists j \leq k$ o menor índice tal que $c_j > a_j$.

  - Se $j \leq k-2$ então $c_j > a_j = b_j$, donde segue $c > b$.
  - Se $j = k-1$ temos $c_i = a_i = b_i$ para $i \in \{1, \ldots, k-2\}$ e $c_{k-1} > a_{k-1}$.

  Então $c_{k-1} \geq a_{k-1} + 1 \geq a_k$. Observe ainda que c tem $k-1$ termos, pois caso contrário teríamos $c_k \geq c_{k-1} \geq a_k$, então $\sum^k c_i > \sum^k a_i = n$, o que é absurdo. Portanto,

  $$\sum_{i=1}^{k-2} c_i + c_{k-1} = \sum_{i=1}^{k-2} a_i + a_{k-1} + a_k \Rightarrow c_{k-1} = a_{k-1} + a_k = b_{k-1} \Rightarrow c = b.$$

  - Por último, se $j = k$ então $\sum^k c_i > \sum^k a_i = n$, o que é absurdo.

  Logo vale $c \geq b$.

- **CASO 2**: $a_k - a_{k-1} > 1$

  Neste caso as linhas 6 e 7 geram a partição $b := (b_i)_k = (b_1, b_2, \ldots, b_k)$, onde $b_i = a_i$ para $i \in \{1, \ldots, k-2\}$, $b_{k-1} = a_{k-1} + 1$ e $b_k = a_k - 1$.

  Observe que $\sum^k b_i = \sum^k a_i = n$, então $b \in Part(n)$. Tome q e r o quociente e resto da divisão b_k por b_{k-1} (linha 8). Temos dois subcasos a considerar:

  - **CASO 2a:** $q > 1$

  Neste caso o retorno do algoritmo é
  $b' = (b_1, \ldots, b_{k-2}, b_{k-1}, \underbrace{b_{k-1}, \ldots, b_{k-1}}_{q-1 \text{ vezes}}, (b_{k-1} + r))$, que possui $k + q - 1$ termos.

  Observe que $\begin{cases} b'_i = a_i, \text{ se } i \in \{1, \ldots, k-2\} \\ b'_i = a_{k-1} + 1, \text{ se } i \in \{k-1, \ldots, k+q-2\} \\ b'_{k+q-1} = a_{k-1} + 1 + r \end{cases}$. Então

  $$\begin{aligned} \sum_{i=1}^{k+q-1} b'_i &= (\sum_{i=1}^{k-2} b'_i) + b'_{k-1} + (\sum_{i=k}^{k+q-2} b'_i) + b'_{k+q-1} \\ &= (\sum_{i=1}^{k-2} a_i) + (a_{k-1} + 1) + (a_{k-1} + 1).(q-1) + (a_{k-1} + 1 + r) = \sum_{i=1}^{k} a_i = n. \end{aligned}$$

  Portanto $b' \in Part(n)$ e por construção vemos que $b' \geq a$.

  Vamos provar que b' é imediatamente maior que a em $Part(n)$. Tome $c = (c_i)_m \in Part(n)$ tal que $c > a$, então por definição $\exists j \leq k$ o menor índice tal que $c_j > a_j$.

  $* \; j \leq k-2 \Rightarrow c_j > a_j = b'_j \Rightarrow c > b'$.

$*$ $j = k-1 \Rightarrow c_{k-1} > a_{k-1} \Rightarrow c_{k-1} \geq a_{k-1} + 1 = b'_{k-1} \Rightarrow c \geq b'$.

$*$ Se $j = k$ então $\sum^k c_i > \sum^k a_i = n$, o que é absurdo.

Logo vale $c \geq b'$

- **CASO 2b:** $q = 1$

Neste caso o algoritmo retorna a própria partição $b = (b_i)_k$. Como $b_{k-1} = a_{k-1} + 1 > a_{k-1}$ então $b > a$.

Vamos provar que b é a sequência imediatamente maior que a em $Part(n)$. Tome $c = \{c_i\}_m \in Part(n)$ tal que $c > a$, então por definição $\exists j \leq k$ o menor índice tal que $c_j > a_j$.

$*$ $j \leq k-2 \Rightarrow c_{k-2} > a_{k-2} = b_{k-2} \Rightarrow c > b$.

$*$ $j = k-1 \Rightarrow c_{k-1} > a_{k-1} \Rightarrow c_{k-1} \geq a_{k-1} + 1 = b_{k-1} \Rightarrow c \geq b$.

$*$ Se $j = k$ então $\sum^k c_i > \sum^k a_i = n$, o que é absurdo.

Logo vale $c \geq b'$

Por fim, se $n > 3$ e $a_1 = \lfloor \frac{n}{2} \rfloor$, então $(a_i)_k$ é a maior partição e o retorno é ***null*** . □

Proposição 3.2. *Se* $\boldsymbol{a} \in Part(n)$*, o algoritmo* $ParticaoSeguinte(\boldsymbol{a})$ *tem complexidade* $O(n)$.

Demonstração. É fácil ver que o único caso que não possui complexidade constante é o referente a linha 9, cujo pior caso ocorre quando a entrada é $a = (1, n-1) \in Part(n)$, onde $q = \lfloor \frac{a_k - 1}{a_{k-1}+1} \rfloor = \frac{n-2}{2}$ e o número de operações executadas é $O(q) = O(n)$. □

Como queremos desenvolver um procedimento para obter uma árvore imediatamente maior que a anterior, precisamos introduzir o conceito de vértice pivô, que indicará onde serão feitas mudanças na árvore.

Definição 3.2. *Seja* $T \in \mathbb{T}$ *ordenada, um vértice* v *de* T *é dito* ***esgotado*** *se é uma folha ou sua partição induzida é o maior elemento de* $Part(l(v))$.

Fato 3.1. *Dada uma árvore* $T \in \mathbb{T}_n$ *ordenada e rotulada, verificar se um vértice* v *de* T *está esgotado pode ser feito em tempo constante (veja o fato 2.2).*

Lembramos que um percurso numa árvore é uma "visita" sistemática a cada um de seus vértices. Em árvores ordenadas $T \in \mathbb{T}$ definimos o percurso **pós-ordem invertido** através do seguinte procedimento recursivo: escrevendo $filhos(raiz(T)) = \{v_1 \leq \ldots \leq v_k\}$, percorremos as subárvores $T(v_k), T(v_{k-1}), \ldots, T(v_2), T(v_1)$ em pós-ordem invertido e por último é feita a visitação de $raiz(T)$.

Para $T \in \mathbb{T}_n$ é fácil ver que tal percurso é feito em tempo $O(n)$, pela proposição 2.1.

Definição 3.3. *Seja* $T \in \mathbb{T}$ *ordenada cujo percurso pós-ordem invertido em* T *visita os vértices na ordem* $v_1, \ldots, v_m$*, onde* $m = |V(T)|$*. O* ***pivô*** *de* T*, caso exista, é o vértice* v_i *de menor índice tal que* v_i *não é esgotado.*

Exemplo 3.1. *O vértice em destaque no grafo ilustrado no exemplo 2.1 é o pivô da árvore.*

Dada uma árvore $T \in \mathbb{T}_n$ ordenada e rotulada, o procedimento para determinação do pivô de T pode ser feito por aplicação direta da definição, ou seja, basta efetuar um percurso pós-ordem invertido e verificar se o vértice visitado é esgotado, tal procedimento chamaremos de $EncontraPivo(T)$. Em caso inexistência de pivô na árvore o procedimento retornará ***null*** . A complexidade decorre diretamente do fato 3.1 e descrevemos abaixo.

Fato 3.2. *A busca pelo vértice pivô em uma árvore de* $\mathbb{T}_n$ *é feita em tempo* $O(n)$.

Escrevemos a seguir um algoritmo que reconstrói um vértice a partir de uma partição dada como entrada, sem alterar o número de folhas da árvore.

Algoritmo 3.2 Reconstrução de vértice através de partição

Input: vértice v de árvore $T \in \mathbb{T}_n$ ordenada e partição $(a_i)_k \in Part(l(v))$
Output: vértice v'

```
1: procedure ReconstroiVertice(v, (a_i)_k)
2:     Substitua os filhos de v por v_1 ..., v_k tais que (a_i)_k seja a partição induzida por v.
3:     for i ← 1 to k do
4:         if a_i > 1 then insira a_i folhas em v_i
5:     end for
6:     return v
7: end function
```

Pelo fato 2.1 garantimos a complexidade do algoritmo 3.2, conforme enunciamos abaixo.

Fato 3.3. *$ReconstroiVertice(v, (a_i)_k)$ tem complexidade $O(l(v))$, onde $(a_i)_k \in Part(l(v))$.*

Lema 3.1. *Uma árvore em $\mathbb{T}_n$ é o maior elemento de $\mathbb{T}_n$ se, e somente se, não possui pivô.*

Demonstração. Faremos a prova por contra-positiva. Seja $T \in \mathbb{T}_n$ com pivô v, cuja partição induzida é $\mathbf{a} \in Part(l(v))$, portanto existe $\mathbf{b} \in Part(l(v))$ tal que $b > a$, então considere a árvore T' obtida de T após substituição de v pelo vértice v' retorno de $ReconstroiVertice(v, \mathbf{b})$. Logo $T' > T$ e portanto T não é o maior elemento de $\mathbb{T}_n$. A recíproca é trivial.

□

Agora somos capazes de desenvolver o procedimento que obtém a árvore imediatamente maior, que será fundamental para o algoritmo de geração dos cografos.

A fim de não sobrecarregar a escrita do algoritmo fixaremos a seguinte notação: Dada $I(v_j) = \{v_1, \ldots, v_j, v_{j+1}, \ldots, v_m\}$ ordenada, definimos os conjuntos $I^+(v_j) = \{v_{j+1}, \ldots, v_m\}$ e $I^-(v_j) = \{v_1, \ldots, v_{j-1}\}$, o que forma uma partição para $I(v_j)$ em $I^-(v_j) \cup \{v_j\} \cup I^+(v_j)$.

Algoritmo 3.3 Árvore imediatamente maior

Input: árvore $T \in \mathbb{T}_n$ ordenada e rotulada
Output: árvore $T' \in \mathbb{T}_n$ imediatamente maior que T

```
1: procedure ArvoreSeguinte(T)
2:     v ← EncontraPivo(T)
3:     if v ≠ null then
4:         b_m ← ParticaoSeguinte((a_i)_k), onde (a_i)_k é partição induzida de v.
5:         v ← ReconstroiVertice(v, (b_i)_m)
6:         x ← v
7:         repeat
8:             for all y ∈ I^+(x) do
9:                 if l(y) = l(x) then copiar subárvore T(x) em T(y)
10:                else y ← ReconstroiVertice(y, c), onde c = (1)_{l(y)}      ▷ l(y) > l(x)
11:            end for
12:            x ← pai(x)
13:        until x = null
14:        return T
15:    else return null                                   ▷ não existe árvore maior
16:    end if
17: end function
```

A ideia geral do algoritmo acima é buscar o pivô da árvore, e em seguida evoluir sua partição induzida e "reiniciar" para a menor configuração possível todos os vértices que a busca pelo pivô passou. Essa ideia é baseada no fato de que a comparação de árvores é feita da esquerda para a direita em cada irmandade, enquanto a busca do pivô ocorre da direita para esquerda. Antes da corretude do algoritmo provaremos a proposição seguinte.

Proposição 3.3. *Seja $T \in \mathbb{T}_n$ ordenada tal que T não é o maior elemento de $\mathbb{T}_n$, então a árvore referente ao retorno de $ArvoreSeguinte(T)$ está ordenada.*

Demonstração. Seja T' o retorno do procedimento, observe que $T' \in \mathbb{T}_n$, pois o número de folhas em cada vértice da árvore é mantido. Denotaremos por v o pivô de T, cuja existência é garantida pelo lema 3.1 e por v' o correspondente de v em T', pela operação da linha 5, Em geral para cada x em T denotaremos por x' o seu correspondente em T', caso exista, segundo as operações nas linhas 9 e 10.

Para cada x' ancestral de v', provaremos que as irmandades $I_{T'}(x')$ e $I_{T'}(v')$, estão ordenadas. Como as demais irmandades não sofreram alteração, a ordenação decorre diretamente da ordenação de T. Mais que isso, pelo mesmo raciocínio vemos que o conjunto $I^-_{T'}(x')$ já está ordenado para cada x', restando apenas provar a ordenação de $I^+_{T'}(x')$ e $I^+_{T'}(v')$. De fato para cada $y' \in I^+_{T'}(x') \cup \{x'\}$ valem os casos:

1. Se $x \leq y$ e $l(x) = l(y)$ então pela operação da linha 9 e lema 2.1 vale $x' \sim y'$.
2. Se $x < y$ e $l(x) < l(y)$ então a partição induzida por y' é a menor de $Part(l(y))$ (linha 10), então $x' < y'$, segundo a definição 2.4.

Portanto em ambos os casos a ordenação dos vértices é mantida, isto é $I_{T'}(x')$ está ordenada. Resta provar a ordenação de $I_{T'}(v')$.

Escrevemos $I_T(v) = \{v_1 \leq \ldots \leq v_j \leq v_{j+1} \leq \ldots \leq v_{j+h}\}$ e $I_{T'}(v') = \{v'_1 \leq \ldots \leq v'_j \leq v'_{j+1} \leq \ldots \leq v'_{j+h}\}$, onde $v := v_j$ e $v' := v'_j$.

Como o algoritmo só altera os vértices do conjunto $I^+_T(v) \cup \{v\}$, então a ordenação de T garante a ordenação de $I^-_{T'}(v')$.

Afirmação: v' está ordenado em relação ao conjunto $I_{T'}(v')$.

Precisamos provar que $v'_{j-1} \leq v'_j \leq v'_{j+1}$. De fato, pela proposição 3.1 e caso 4.1 da definição 2.4 temos $v'_j > v_j$, mas pela ordenação de T vale $v_j \geq v_{j-1} \sim v'_{j-1}$, portanto $v'_j \geq v'_{j-1}$. Por outro lado se $l(v_{j+1}) = l(v_j)$ então pela linha 9 temos $v'_{j+1} \sim v'_j$, senão $l(v_{j+1}) > l(v_j)$ e pela linha 10 e definição 2.4 temos $v'_{j+1} > v'_j$, donde segue a afirmação.

Por outro lado $I^+_{T'}(v')$ está ordenado, pela justificativa análoga ao feito em $I^+_{T'}(x')$, pois ambos são tratados pela operação das linhas 9 e 10. Deste último fato e afirmação anterior garantimos a ordenação de $I_{T'}(v')$, donde segue o resultado.

□

Teorema 3.1. *Seja $T \in \mathbb{T}_n$. O retorno do procedimento $ArvoreSeguinte(T)$ é a árvore imediatamente maior que T em $\mathbb{T}_n$, caso seja **null** então T é o maior elemento do conjunto.*

Demonstração. Se o procedimento retorna ***null*** , o resultado é garantido pelo lema 3.1. Caso contrário, seja T' o retorno do algoritmo.

Como a árvore T' está ordenada pela proposição anterior, podemos efetuar a comparação entre T e T' com a disposição dos vértices retornada pelo algoritmo. Considere v o pivô de T e v' seu correspondente em T', em geral para cada x em T denotaremos por x' o seu correspondente em T', caso exista, segundo as operações nas linhas 9 e 10, mesma notação usada na prova da proposição anterior.

Afirmação: $T' > T$.

De fato, inicialmente observe que a proposição 3.1 garante $v' > v$, pela definição 2.4.

Se $v = raiz(T)$ então é claro que $T' > T$. Caso contrário escrevemos $I_T(v) = \{v_1 \leq \ldots \leq v_k \leq v_{k+1} \leq \ldots \leq v_h\}$ e $I_{T'}(v') = \{v'_1 \leq \ldots \leq v'_k \leq v'_{k+1} \leq \ldots \leq v'_h\}$, onde $v_k := v$ e $v'_k := v'$. Como o algoritmo só faz alteração nos vértices de $I_T^+(v_k)$, então $v_i \sim v'_i$ para $1 \leq i < k$, donde $pai(v') > pai(v)$, pelo caso 3.2b da definição 2.4.

Seja $x = pai(v)$, escreva $I_T(x) = \{x_1 \leq \ldots \leq x_j \leq \ldots \leq x_u\}$ e $I_{T'}(x') = \{x'_1 \leq \ldots \leq x'_j \leq \ldots \leq x'_u\}$, onde $x_j := x$ e $x'_j := x'$. Com raciocínio semelhante ao feito acima temos $x_i \sim x'_i$ para $1 \leq i < j$, então pelo caso 3.2b da definição 2.4 garantimos $pai(x') > pai(x)$, uma vez que $x' > x$. Em geral o mesmo raciocínio pode ser feito de forma sucessiva para cada ancestral x de v, obtendo sempre $x' > x$. Ao alcançar o caso $x = raiz(T)$ garantimos que $T' > T$, donde segue a afirmação feita.

Vamos agora provar que T' é imediatamente maior que T em $\mathbb{T}_n$.

Inicialmente o algoritmo toma o pivô v de T e constrói v' cuja partição induzida é imediatamente maior que a de v, pela linha 5. Seja x um ancestral de v ou ainda $x = v$, então para cada $y \in I_T(x)$ o algoritmo constrói $y'_i \in I_{T'}(x')$ da seguinte forma: se $y \in I_T^-(x)$ então y' é equivalente a x', se $y \in I_T^+(x)$, este é um vértice esgotado pela definição de pivô, então y é construído de forma que este seja o menor vértice possível, conforme detalhado abaixo e semelhante a prova da proposição anterior.

1. Se $x \leq y$ e $l(x) = l(y)$ então $x' \sim y'$ (linha 9).
2. Se $x < y$ e $l(x) < l(y)$ então y' é construído a partir da menor partição $Part(l(y))$, obtendo $x' < y'$ (linha 10).

Como $j = min\{i; x_i \not\sim x'_i\}$ e $x'_j > x_j$, então $pai(x') > pai(x)$, por definição. Além disso, pelos casos vistos acima, para cada $i \in \{j+1, \ldots, u\}$ temos x_i esgotado em T e x'_i em T' construído de forma que seja o menor possível com $l(x_i)$ folhas, então $pai(x')$ é imediatamente maior que $pai(x)$, dentre todos os vértices com $l(pai(x))$ folhas induzidas. Como esse argumento vale para cada x ancestral de v ou $x = v$, então T' é imediatamente maior que T em $\mathbb{T}_n$. □

Proposição 3.4. *Se $T \in \mathbb{T}_n$ o algoritmo $ArvoreSeguinte(T)$ tem complexidade $O(n)$.*

Demonstração. O pior caso ocorre quando T possui pivô $v \neq raiz(T)$.

A proposição 3.2 e fatos 3.2 e 3.3 garantem que todas as operações realizadas fora do *loop* das linhas 7-13 são feitas em $O(n)$.

O *loop* interno (linha 8) executará para cada $y \in I^+(x)$ as operações da linha 10 e 11, escrevendo $I(x) = \{x_1, \ldots, x_k\}$, cuja complexidade de pior caso é $\sum_{i=2}^{k} O(l(x_i)) = O(l(pai(x))$, onde a igualdade decorre do fato 2.1.

Seja $P_v : (v = v_0, v_1, \ldots, v_{k-1}, v_k = raiz(T))$ o caminho do pivô até a raiz da árvore. O *loop* (linha 7) executa o *loop* interno para cada x de P_v (linha 12), então seguindo a ideia desenvolvida no parágrafo anterior concluímos o seguinte raciocínio: Para $x = v_0$ o processo da linha 8 é feito para cada $I^+(v_0)$ cujo custo é $O(l(v_1))$; o tempo de execução até $x = v_i$, $i \in \{1, \ldots, k-1\}$ é:

$$O(l(v_i)) + \sum_{w \in I^+(v_i)} O(l(w))) = O(l(v_{i+1})),$$

onde a primeira parcela é referente a complexidade das i iterações anteriores e a igualdade decorre do fato 2.1.

O processo finaliza ao alcançar o último ancestral $x = v_k$, portanto o *loop* tem complexidade $O(l(v_k)) = O(n)$. Donde segue o resultado. □

Exemplo 3.2. *A árvore imediatamente maior da ilustrada no exemplo 2.1 está representada abaixo, cujo pivô está em destaque.*

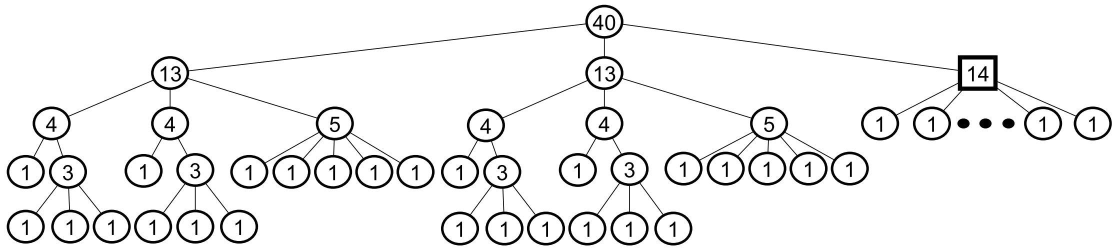

Figura 2: Árvore imediatamente maior

A geração dos elementos de $\mathbb{T}_n$ pode ser feita da seguinte forma: tomamos a menor árvore de $\mathbb{T}_n$ (fato 2.4) e a partir desta obtemos a árvore imediatamente maior por sucessivas aplicações do algoritmo 3.3, até alcançar a maior árvore do conjunto, indicada pela ausência de pivô. Note que a princípio seria necessário a cada iteração ordenar a árvore obtida para então fazer a execução do algoritmo 3.3, o que seria extremamente custoso em termos de tempo de execução. Porém a proposição 3.3 garante que é necessário a ordenação apenas na primeira árvore, que é trivial (fato 2.4) que e as demais terão a ordenação mantida, garantindo a eficiência do procedimento.

Como o número de elementos de $\mathbb{T}_n$ (cografos com n vértices) existentes é exponencial, então é claro que o algoritmo descrito tem complexidade total *não polinomial*, mas a medida conveniente de eficiência para um **algoritmo gerador** é através do seu **atraso** (ou *delay*), isto é, o tempo entre a geração de dois elementos consecutivos do conjunto. No nosso caso a geração entre uma árvore e a seguinte é dada unicamente pelo algoritmo 3.3, então a proposição 3.4 garante que geração dos elementos de $\mathbb{T}_n$ possui atraso linear em n.

Denotaremos por $\mathscr{C}$ o conjunto de todos os cografos, tal conjunto pode ser particionado como $\mathscr{C} = \bigcup_{n=1}^{\infty} \mathscr{C}_n$, onde $\mathscr{C}_n = \{G \in \mathscr{C}; |V(G)| = n\}$. Vamos estabelecer uma relação de ordem entre os elementos de $\mathscr{C}_n$

Definição 3.4. *Sejam $G_1, G_2 \in \mathscr{C}_n$, e T_1 e T_2 suas respectivas coárvores ordenadas enraizadas em r_1 e r_2, respectivamente. Se $T_1 > T_2$ definimos $G_1 > G_2$, se $T_1 = T_2$, então analisamos os tipos das raízes da seguinte forma: se r_1 é tipo-1 e r_2 é tipo-0 definimos $G_1 > G_2$, se r_1 e r_2 são ambas tipo-1 (ou tipo-0) então $G_1 = G_2$.*

Desenvolvemos toda teoria necessária para alcançarmos nosso objetivo principal: Dado um inteiro positivo n, construir um algoritmo que gere de forma eficiente todos os cografos com n vértices, sem rotulação, de forma que o k-ésimo cografo gerado não seja isomorfo a nenhum dos $k - 1$ cografos gerados anteriormente.

Algoritmo 3.4 Gerador de cografos

Input: Inteiro $n \geq 2$
Output: Geração de todos cografos com n vértices.

```
procedure GeradorCografos(n)
    Seja T_1 a menor árvore de 𝕋_n
    i ← 1
    repeat
        Seja G_i cografo associado a T, cuja raiz é tipo-0.
        Seja G'_i cografo associado a T, cuja raiz é tipo-1.
        T_{i+1} ← ArvoreSeguinte(T_i)
        i ← i + 1
    until T_i = null
end function
```

Teorema 3.2. *Dado inteiro $n \geq 2$, o algoritmo 3.4 gera todos os elementos de $\mathscr{C}_n$, sem rotulação nos vértices, de forma que todos os cografos gerados são dois a dois não isomorfos.*

Demonstração. O teorema 3.1 garante que a árvore T_{i+1} é imediatamente maior que T_i em $\mathbb{T}_n$, como algoritmo inicia na menor e finaliza na maior árvore de $\mathbb{T}_n$, então são gerados todos elementos de $\mathbb{T}_n$ em ordem crescente $T_1 < T_2 < \ldots < T_M$, onde todas são não isomorfas dois a dois pelo lema 2.1. Como cada árvore está associada a exatamente dois cografos (complementares entre si) e a proposição 2.2 garante que cada cografo possui uma única coárvore ordenada, então o procedimento em questão gera todos os cografos de n vértices, sem geração de cografos isomorfos. □

Exemplo 3.3. *Abaixo estão representados em ordem crescente todos os cografos com 4 vértices, gerados por $GeradorCografos(4)$.*

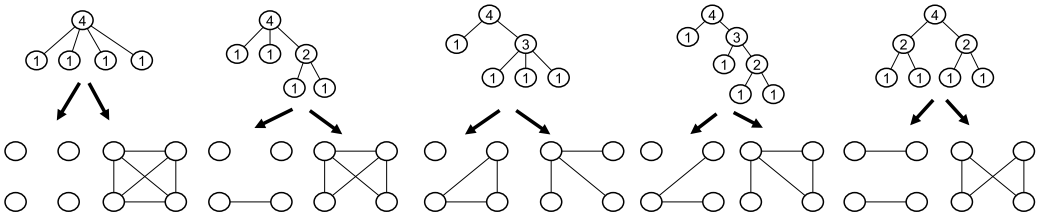

Figura 3: Cografos com 4 vértices

A proposição 3.4 garante a eficiência do algoritmo acima, como enunciado a seguir.

Proposição 3.5. *O algoritmo $GeradorCografos(n)$ possui atraso $O(n)$.*

É natural em teoria dos grafos restringir problemas somente aos grafos conexos, mas observe que basta a retirada da linha 5 do algoritmo 3.4 e obtemos um gerador de cografos conexos.

4. Conclusão

Neste trabalho nos baseamos na representação do cografo pela sua coárvore para desenvolver um algoritmo capaz de gerar todos os cografos, sem repetição, com um determinado número de vértices, cujo tempo de atraso é linear no número de vértices. Em particular, podemos fornecer o número exato de cografos existentes para cada número de vértices fixado, como descrevemos na tabela abaixo, onde a quantidade de cografos conexos é dada pela metade de cada valor obtido. O código foi implementado na linguagem $C\#$ e executado no Sistema Operacional *Windows 7*, equipado com 8GB de RAM e Processador AMD FX-6100 Six-core 3.30GHz.

n	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$\lvert\mathscr{C}_n\rvert$	2	4	10	24	66	180	510	1508	4442	13700	41598	131744	410066	1321628

As principais aplicações do algoritmo gerador ocorrem na formulação de novas conjecturas ou busca de contra-exemplos, uma vez que podemos fazer testes em todos os cografos com um determinado número de vértices. Além disso, muitos parâmetros para cografos são obtidos diretamente da coárvore (como número cromático, b-cromático, número clique), o que torna a aplicação ainda mais evidente, uma vez que a construção do algoritmo é baseada unicamente na coárvore.

Referências

Corneil, D. G., Lerchs, H., e Burlingham, L. S. (1981). Complement reducible graphs. *Discrete Applied Mathematics*, 3(3):163–174.

Corneil, D. G., Perl, Y., e Stewart, L. K. (1984). Cographs: recognition, applications and algorithms. *Congressus Numer*, 43:249–258.

Ravelomanana, V. e Thimonier, L. (2001). Asymptotic enumeration of cographs. *Electronic Notes in Discrete Mathematics*, 7:58–61.