

Universidade de São Paulo
Instituto de Matemática e Estatística
Bacharelado em Ciência da Computação

Tiago Madeira

**Geração uniforme de k -trees para
aprendizado de redes bayesianas**

Supervisor: Prof. Dr. Denis Deratani Mauá

São Paulo
Novembro de 2016

Resumo

O resumo ainda não foi escrito.

Palavras-chave: sem, resumo, por, enquanto.

Abstract

The abstract has not been written yet.

Keywords: no, abstract, yet.

Sumário

1	Introdução	1
1.1	Código desenvolvido	2
1.2	Organização da monografia	3
2	Fundamentos	5
2.1	Grafos	5
2.1.1	k -trees	9
2.2	Probabilidade	10
2.2.1	Redes bayesianas	13
3	Geração aleatória de k-trees	15
3.1	Codificando árvores e k -trees	15
3.2	A solução de Caminiti <i>et al.</i>	17
3.2.1	Codificação	19
3.2.2	Decodificação	23
3.3	Geração uniforme	25
3.4	Utilitários	26
3.4.1	code-ktree	26
3.4.2	decode-ktree	27
3.4.3	generate-ktree	29

3.5	Testes, experimentos e resultados	29
3.5.1	Testes unitários e cobertura	29
3.5.2	Experimentos e resultados	31
4	Aprendizado de redes bayesianas	33
4.1	Aprendendo redes bayesianas	33
5	Conclusão	35

Capítulo 1

Introdução

Em teoria dos grafos, *k-trees* são consideradas uma generalização de árvores. Há interesse considerável em desenvolver ferramentas eficientes para manipular essa classe de grafos, porque todo grafo com *treewidth* k é um subgrafo de uma *k-tree* e muitos problemas NP-completos podem ser resolvidos em tempo polinomial quando restritos a grafos com *treewidth* limitada.

Com efeito, o artigo de Arnborg e Proskurowski [1] apresenta algoritmos para resolver em tempo linear problemas como, dado um grafo com *treewidth* limitada:

- Encontrar o tamanho máximo dos seus conjuntos independentes;
- Computar o tamanho mínimo dos seus conjuntos dominantes;
- Calcular seu número cromático; e
- Determinar se ele tem um ciclo hamiltoniano.

O problema que desperta nosso interesse em *k-trees* é a inferência em redes bayesianas.

Uma rede bayesiana é um modelo probabilístico em grafo usado para raciocinar e tomar decisões em situações com incerteza através de técnicas de inteligência artificial e aprendizagem computacional. Ela representa uma distribuição de probabilidade multivariada num DAG (grafo acíclico dirigido) no qual os vértices correspondem às variáveis aleatórias do domínio e as arestas correspondem, intuitivamente, a influência de uma variável sobre outra.

Segundo Koller e Friedman [8], a inferência em redes bayesianas em geral é NP-difícil; porém, se seu DAG possui *treewidth* limitado, a inferência pode ser realizada em tempo polinomial. Daí a importância de aprender redes bayesianas que tenham *treewidth* limitada.

A partir dessa motivação, este trabalho de conclusão de curso consistiu em estudar os conceitos de teoria dos grafos relacionados a *k-trees* e implementar um algoritmo para gerar *k-trees* de forma uniforme que possam ser usadas no aprendizado de redes bayesianas.

1.1 Código desenvolvido

As implementações deste trabalho foram realizadas na linguagem *Go*¹. *Go* é uma linguagem de código aberto criada em 2007. Ela é compilada e usa tipagem estática como o C, mas por ser uma linguagem muito nova tem *garbage collection* e recursos para programação concorrente.

Escolhemos *Go* porque ela tem boa performance e é agradável de usar. Tem sistemas de pacotes (`go get`), testes (`go test`) e documentação (*GoDoc*²) padronizados facilitando que os códigos sejam testados e reutilizados. Produz código limpo e padronizado (identação, espaçamento e outros detalhes de estilo são automatizados pela ferramenta `gofmt` que vem com ela).

¹*The Go Programming Language*: <https://golang.org/>

²*GoDoc*: <https://godoc.org/>

Todo o código desenvolvido neste trabalho está num repositório público no *GitHub*³ cujo endereço é <https://github.com/tmadeira/tcc/>.

A documentação de todas as estruturas e funções declaradas no código está disponível em <https://godoc.org/github.com/tmadeira/tcc>.

Para baixar o código, rodar os testes e instalar os utilitários, recomenda-se usar as ferramentas da linguagem *Go*:

```
1 | $ export ${GOPATH:=$HOME/go}
2 | $ mkdir -p $GOPATH
3 | $ go get github.com/tmadeira/tcc/...
4 | $ go test -v github.com/tmadeira/tcc/...
5 | $ go install github.com/tmadeira/tcc/examples/...
```

1.2 Organização da monografia

No capítulo 2, apresentamos definições fundamentais de teoria dos grafos, teoria da probabilidade e redes bayesianas que o leitor deve conhecer para compreender o trabalho.

No capítulo 3, apresentamos o problema de codificar *k-trees*, discutimos os algoritmos lineares para codificar e decodificar *k-trees* propostos no artigo “*Bijective Linear Time Coding and Decoding for k-Trees*” [4], explicamos como eles foram implementados neste trabalho para gerar *k-trees* aleatórias uniformemente e apresentamos o resultado que obtivemos através de experimentos.

No capítulo 4, explicamos como as *k-trees* que geramos no capítulo 3 foram usadas para aprender redes bayesianas a partir do arcabouço desenvolvido no artigo “*Advances in Learning Bayesian Networks of Bounded Treewidth*” [10].

³*GitHub*: <https://github.com/>

No capítulo 5, comparamos os resultados obtidos com o estado da arte e listamos as conclusões do trabalho.

Capítulo 2

Fundamentos

Neste capítulo, apresentamos definições fundamentais de teoria dos grafos, teoria da probabilidade e redes bayesianas que o leitor deve conhecer para compreender o trabalho.

Outras definições mais específicas, como as utilizadas para construir o algoritmo para codificar e decodificar *k-trees*, estão localizadas nos capítulos subsequentes.

Partimos do pressuposto de que o leitor conhece notações básicas de conjuntos.

2.1 Grafos

Nesta seção apresentamos de forma breve apenas os conceitos de teoria dos grafos necessários para a compreensão deste trabalho. Mais detalhes podem ser encontrados no livro de Bondy e Murty [3], que foi utilizado como referência.

Definição 1 (grafo). Um **grafo** é um par ordenado $G = (V, E)$. Os elementos de V são chamados de **vértices** de G . Os elementos de E são chamados

de **arestas** de G e consistem em pares (não-ordenados) de vértices distintos¹. Dados $u, v \in V$, se $(u, v) \in E$ dizemos que u e v são **adjacentes** em G .

A figura 2.1(a) mostra como costuma ser representado um grafo. Os vértices são representados pelos círculos e as arestas são representadas pela ligação entre eles. Se $(u, v) \in E$, há uma linha ligando os vértices u e v .

Dado um grafo $G = (V, E)$, o número de arestas que incide num determinado vértice $v \in V$ é chamado de **grau** do vértice v .

Há diferentes estruturas de dados que podem ser usadas para representar um grafo na memória do computador. Uma das mais comuns, que usamos nas implementações deste trabalho, é a **lista de adjacência**. Suponha, sem perda de generalidade, que os vértices de um grafo G sejam representados por inteiros de 0 a $|V| - 1$. Então, a representação desse grafo consiste em um vetor de listas `Adj`. A lista `Adj[i]` contém os vértices adjacentes ao vértice de rótulo i (para todo $i \in [0, |V|)$).

Definição 2 (grafo dirigido). Um grafo $G = (V, E)$ é dito **dirigido** se E consiste em pares *ordenados* de vértices. Se $(a, b) \in E$, dizemos que a aponta para b , que há uma aresta de a para b ou que b é filho de a .

A figura 2.1(b) mostra como costuma ser representado um grafo dirigido. Como o conjunto de arestas consiste em pares ordenados, elas são representadas por setas. Se $(u, v) \in E$, então a seta aponta de u para v .

Definição 3 (grafo completo). Um grafo $G = (V, E)$ é dito **completo** se $(u, v) \in E$ para todo $u, v \in V, u \neq v$. Um grafo completo com n vértices é geralmente denotado K_n .

Na figura 2.1(c), a representação de um grafo completo com 4 vértices.

¹A rigor, por causa da palavra “distintos”, essa é a definição do que a literatura costuma chamar de *grafo simples*. Tal definição é utilizada porque neste trabalho não temos interesse em grafos que possuam arestas (u, v) com $u = v$.

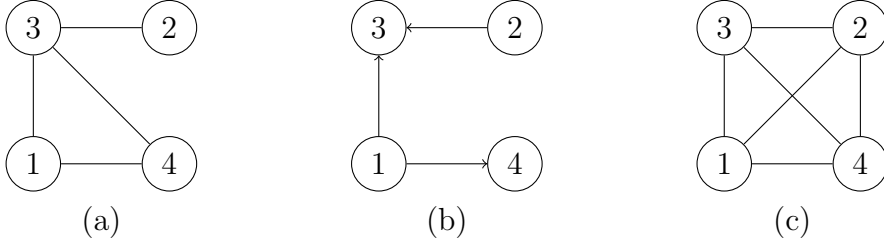


Figura 2.1: **(a)** Representação do grafo $G = (V_G, E_G)$ com $V_G = \{1, 2, 3, 4\}$ e $E_G = \{(1, 3), (1, 4), (2, 3), (3, 4)\}$. **(b)** Representação do grafo dirigido $D = (V_D, E_D)$ com $V_D = \{1, 2, 3, 4\}$ e $E_D = \{(1, 3), (1, 4), (2, 3)\}$. **(c)** Representação do K_4 , o grafo completo com 4 vértices.

Definição 4 (subgrafo). Um grafo $F = (V_F, E_F)$ é chamado de **subgrafo** de $G = (V_G, E_G)$ se $V_F \subseteq V_G$ e $E_F \subseteq E_G$.

Definição 5 (subgrafo induzido). Dado um grafo $G = (V, E)$ e um subconjunto V' de V , o subgrafo de G **induzido** por V' , $G' = (V', E')$, é o grafo formado pelos vértices $V' \subseteq V$ e arestas que só contém elementos de V' , ou seja, $E' = \{(u, v) \in E \mid u, v \in V'\}$.

Definição 6 (k -clique). Seja $G = (V, E)$ um grafo. Um **k -clique** (também chamado de **clique de tamanho k**) é um subconjunto dos vértices, $C \subseteq V$, tal que $(u, v) \in E \forall u, v \in C, u \neq v$ (ou seja, tal que o subgrafo induzido por C é completo).

Definição 7 (caminho). Dado um grafo $G = (V, E)$, um **caminho** em G é um subgrafo de G cujos vértices podem ser arranjados numa sequência linear de forma que dois vértices são adjacentes se eles são consecutivos na sequência e não-adjacentes caso contrário. Se $u, v \in V$ pertencem a um caminho P , dizemos que eles estão conectados pelo caminho P .

Definição 8 (distância). Dado um grafo $G = (V, E)$ e dois vértices $(u, v) \in V$, a **distância** entre u e v é o número de arestas num menor caminho que os conecte.

Definição 9 (ciclo). Dado um grafo $G = (V, E)$, um **ciclo** em G é um caminho formado por vértices $x_1, \dots, x_k \in V$ onde $x_1 = x_k$.

Definição 10 (DAG). Um grafo $G = (V, E)$ é chamado de **DAG** (do inglês *directed acyclic graph*: grafo dirigido acíclico) se ele é dirigido e não possui ciclos.

Dizemos que um ciclo tem uma **corda** se dois vértices no ciclo são conectados por uma aresta que não está no ciclo. Um **grafo cordal** é um grafo no qual todos os ciclos com pelo menos 4 vértices têm uma corda. Qualquer grafo pode ser transformado num grafo cordal adicionando-se arestas num processo chamado de **cordalização**.

O **grafo moral** de um DAG $G = (V, E)$ é um grafo não-dirigido obtido conectando-se todo par de vértices com um filho em comum e retirando a direção das arestas.

Definição 11 (*treewidth*). [10] Dado um grafo $G = (V, E)$, seu ***treewidth*** é um inteiro definido da seguinte forma:

- Se G é um grafo cordal, então seu *treewidth* é o tamanho do seu maior clique menos 1.
- Se G é um grafo não-dirigido arbitrário, então seu *treewidth* é o mínimo entre os *treewidth* de todas as suas cordalizações.
- Se G é um DAG, então seu *treewidth* é o *treewidth* do seu grafo moral.

Definição 12 (árvore). Dado um grafo $G = (V, E)$, dizemos que ele é uma **árvore** se cada dois vértices $u, v \in V$ são conectados por exatamente um caminho.

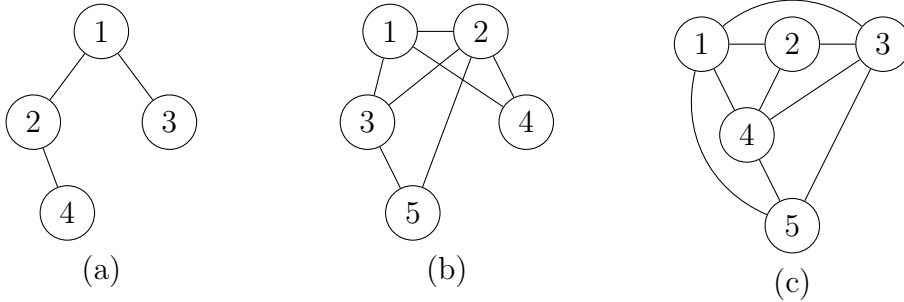


Figura 2.2: **(a)** Uma 1-*tree* (ou seja, uma árvore comum) com 4 vértices. **(b)** Uma 2-*tree* com 5 vértices. **(c)** Uma 3-*tree* com 5 vértices.

Dada uma árvore $T = (V, E)$, os vértices em V que tem grau 1 são chamados de **folhas**.

Dada uma árvore $T = (V, E)$, às vezes é conveniente destacar um vértice $r \in V$ e chamá-lo de **raiz** da árvore T . Chamamos o par formado pela árvore T e pela raiz $r \in V$ de **árvore enraizada**.

2.1.1 k -trees

Definição 13 (k -tree). [7] Uma k -*tree* é definida da seguinte forma recursiva:

1. Um grafo completo com k vértices é uma k -*tree*.
2. Se $T'_k = (V, E)$ é uma k -*tree*, $K \subseteq V$ é um k -clique e $v \notin V$, então $T_k = (V \cup \{v\}, E \cup \{(v, x) \mid x \in K\})$ é uma k -*tree*.

Na figura 2.2(a), um exemplo de k -*tree* com $k = 1$ (ou seja, uma árvore comum) e $n = 4$ vértices rotulados com inteiros em $[1, 4]$; na figura 2.2(b), um exemplo de k -*tree* com $k = 2$ e $n = 5$ vértices rotulados com inteiros em $[1, 5]$; na figura 2.2(c), um exemplo de k -*tree* com $k = 3$ e $n = 5$ vértices rotulados em $[1, 5]$.

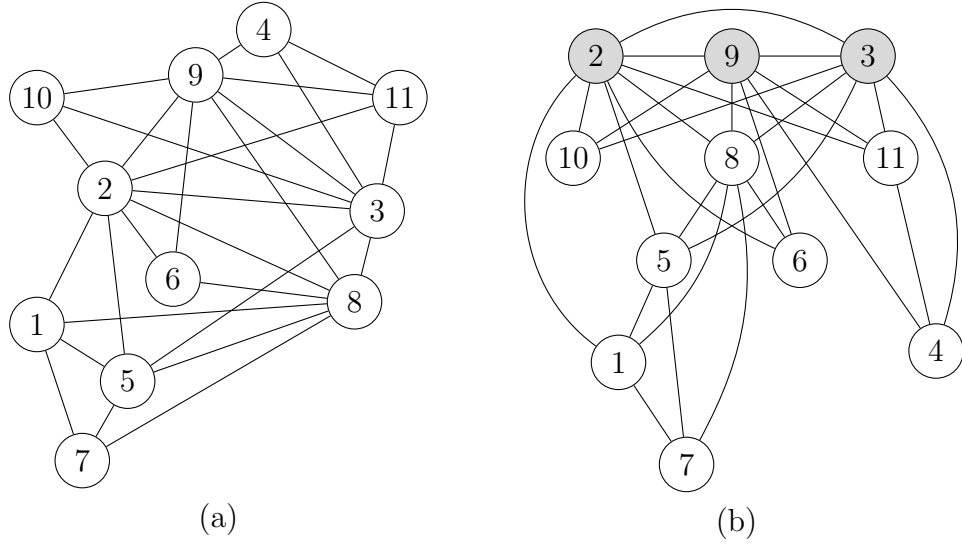


Figura 2.3: **(a)** Uma 3-tree T_3 com 11 vértices. **(b)** A mesma 3-tree (T_3) enraizada no 3-clique $\{2, 3, 9\}$.

Definição 14 (k -tree enraizada). [4] Uma k -tree enraizada é uma k -tree com um k -clique destacado $R = \{r_1, r_2, \dots, r_k\}$ que é chamado de *raiz* da k -tree enraizada.

Na figura 2.3(a), um exemplo de uma k -tree com $k = 3$ e $n = 11$ vértices rotulados com inteiros em $[1, 11]$. Na figura 2.3(b), a mesma k -tree, dessa vez enraizada no 3-clique $R = \{2, 3, 9\}$.

Definição 15 (*partial k-tree*). [2] Um subgrafo de uma k -tree é chamado de *partial k-tree*. Um grafo é uma *partial k-tree* se e só se ele tem *treewidth* menor ou igual a k .

2.2 Probabilidade

Nesta seção apresentamos de forma sintética alguns conceitos de teoria da probabilidade necessários para a compreensão deste trabalho. Mais detalhes podem ser encontrados no livro de Koller e Friedman [8], que foi utilizado

como referência.

Um **experimento aleatório** é um fenômeno que possui resultado imprevisível. Por exemplo, o lançamento de um dado de seis faces é um experimento aleatório.

O **espaço amostral** de um experimento aleatório, geralmente denotado Ω , é o conjunto de todos os resultados possíveis do experimento. Por exemplo, o espaço amostral do lançamento de um dado de seis faces é $\{1, 2, 3, 4, 5, 6\}$.

Chamamos de **eventos** os subconjuntos de um espaço amostral para os quais pretendemos atribuir probabilidades. Por exemplo, no lançamento de um dado de seis faces o evento $\{6\}$ corresponde ao caso em que o lançamento resulta em 6 e o evento $\{1, 2\}$ corresponde ao caso em que o lançamento resulta em 1 ou 2.

A teoria da probabilidade requer que o **espaço dos eventos** $S \subseteq \Omega$ satisfaça três propriedades:

- Deve conter o **evento vazio** \emptyset e o **evento trivial** Ω .
- Se $\alpha, \beta \in S$, então $\alpha \cup \beta \in S$.
- Se $\alpha \in S$, então $\Omega \setminus \alpha \in S$.

O requisito de que o espaço dos eventos é fechado sob a união e o complemento implica que ele também seja fechado sob outras operações booleanas como interseção e diferença.

Definição 16 (distribuição de probabilidade). Seja Ω um espaço amostral e S o espaço dos eventos. Uma **distribuição de probabilidade** P sobre (Ω, S) é um mapeamento dos eventos em S para valores reais que satisfaz as seguintes condições:

- Probabilidades são não-negativas, ou seja, $P(\alpha) \geq 0$ para todo $\alpha \in S$.
- O evento trivial tem a maior probabilidade possível, ou seja, $P(\Omega) = 1$.
- A probabilidade de que um de dois eventos disjuntos ocorra é a soma das probabilidades de cada evento, ou seja, se $\alpha, \beta \in S$ e $\alpha \cup \beta = \emptyset$, então $P(\alpha \cup \beta) = P(\alpha) + P(\beta)$.

Essas condições implicam em outras. Em particular, vale destacar que $P(\emptyset) = 0$ e $P(\alpha \cup \beta) = P(\alpha) + P(\beta) - P(\alpha \cap \beta)$.

Definição 17 (probabilidade condicional). Seja P uma distribuição de probabilidade sobre (Ω, S) e sejam $\alpha, \beta \in S$. A **probabilidade condicional** de β dado α , $P(\beta|\alpha)$, é definida como:

$$P(\beta|\alpha) = \frac{P(\alpha \cap \beta)}{P(\alpha)}$$

Duas consequências da definição da probabilidade condicional são as fórmulas que chamamos de **regra da cadeia** e **regra de Bayes**:

Regra da Cadeia. Se $\alpha_1, \dots, \alpha_k \in S$, então

$$P(\alpha_1 \cap \dots \cap \alpha_k) = P(\alpha_1)P(\alpha_2|\alpha_1) \dots P(\alpha_k|\alpha_1 \cap \dots \cap \alpha_{k-1})$$

Regra de Bayes. Se $\alpha, \beta \in S$, então

$$P(\alpha|\beta) = \frac{P(\beta|\alpha)P(\alpha)}{P(\beta)}$$

Uma **variável aleatória** é uma função que associa a cada elemento em Ω um valor. Por exemplo, no caso do lançamento de um dado de seis faces, podemos definir uma variável aleatória $f_{\text{Resultado}}(\omega)$ para $\omega \in \Omega$ que vale

“*bom*” se o resultado do lançamento é 6 ou “*ruim*” caso contrário. Nesse caso, podemos representar $P(\{1, 2, 3, 4, 5\})$ pela notação $P(\text{Resultado} = \text{“ruim”})$, que equivale a $P(\{\omega \in \Omega : f_{\text{Resultado}}(\omega) = \text{“ruim”}\})$.

Muitas vezes a notação $P(X = x)$ é redundante, porque x é um valor que só faz sentido como imagem de X . Nesses casos, é comum usar a abreviação $P(x)$. Além disso, no lugar de $P((X = x) \cap (Y = y))$ é comum escrever $P(X = x, Y = y)$ ou simplesmente $P(x, y)$.

2.2.1 Redes bayesianas

Redes bayesianas são modelos probabilísticos gráficos que representam distribuições de probabilidade conjunta e são usados para raciocinar em situações com incerteza. Formalmente:

Definição 18 (rede bayesiana). [10] Seja $N = \{1, \dots, n\}$ e seja $X = \{X_i : i \in N\}$ um conjunto de variáveis aleatórias X_i tomando valores em conjuntos finitos \mathcal{X}_i . Uma **rede bayesiana** é uma tripla (X, G, θ) , onde $G = (V, E)$ é um DAG (que chamamos de **estrutura** da rede bayesiana) cujos vértices correspondem a variáveis em X e $\theta = \{\theta_i(x_i, x_{\pi_i})\}$ é um conjunto de parâmetros numéricos especificando valores de probabilidade condicional $\theta_i(x_i, x_{\pi_i}) = P(x_i | x_{\pi_i})$ para todo vértice $i \in V$, valor $x_i \in \mathcal{X}_i$ e atribuição x_{π_i} para os pais π_i de X_i (em G).

Redes bayesianas são geralmente usadas para fazer inferências como computar a probabilidade de alguma variável depois que alguma evidência é observada. Por exemplo, uma rede bayesiana pode representar as relações de probabilidade entre doenças e sintomas. Observados alguns sintomas, a rede pode ser usada para computar a probabilidade da presença das doenças.

No capítulo 4, discutimos o problema de aprender estruturas de redes

bayesianas a partir de datos.

Capítulo 3

Geração aleatória de k -trees

O problema de gerar k -trees está intimamente relacionado ao problema de codificá-las e decodificá-las. De fato, se há uma codificação bijetiva que associa k -trees a *strings*, basta gerar *strings* aleatórias para gerar k -trees aleatórias.

Neste capítulo, apresentamos o problema de codificar k -trees, discutimos a solução linear para codificar e decodificar k -trees de forma bijetiva proposta por Caminiti *et al.* [4], explicamos como ela foi implementada neste trabalho para gerar k -trees aleatórias e mostramos os resultados obtidos.

3.1 Codificando árvores e k -trees

O problema de codificar árvores já foi amplamente estudado na literatura. Como destaca Caminiti *et al.* [4]:

Codificar árvores rotuladas por meio de *strings* de rótulos de vértices é uma alternativa interessante à representação usual de estruturas de dados de árvore na memória e tem muitas aplicações práticas (por exemplo, algoritmos evolucionários sobre árvores, geração aleatória de árvores, compressão de dados e computação

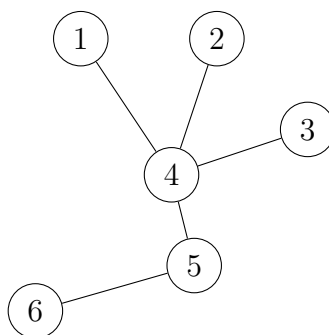


Figura 3.1: A árvore rotulada equivalente ao código de Prüfer $\{4, 4, 4, 5\}$.

do volume de floresta de grafos). Diversos códigos bijetivos diferentes que realizam associações entre árvores rotuladas e *strings* de rótulos foram introduzidas. De um ponto de vista algorítmico, o problema foi cuidadosamente investigado e algoritmos ótimos de codificação e decodificação desses códigos são conhecidos.

Em 1889, Cayley [5] demonstrou que para um conjunto de n vértices distintos existem n^{n-2} árvores possíveis. Desde lá, foram criados vários códigos para associar *strings* e árvores.

Um dos mais conhecidos é o código de Prüfer [11], que surgiu em 1918 e é bijetivo, associando cada árvore (rotulada) de n vértices a uma lista distinta de comprimento $n - 2$ no alfabeto dos rótulos da árvore.

Codificar uma árvore usando o código de Prüfer é trivial: basta remover iterativamente as folhas da árvore até que apenas dois vértices sobrem, escolhendo sempre a folha de menor rótulo. Quando uma folha é removida, adiciona-se ao código o rótulo do seu vizinho.

A figura 3.1 exemplifica a codificação de Prüfer mostrando uma árvore cujo o código resultante do algoritmo é $\{4, 4, 4, 5\}$.

k -trees [7] são consideradas uma generalização de árvores. Há interesse

considerável em desenvolver ferramentas eficientes para manipular essa classe de grafos, porque todo grafo com *treewidth* k é um subgrafo de uma k -tree e muitos problemas NP-completos podem ser resolvidos em tempo polinomial quando restritos a grafos com *treewidth* limitada, como destacado no capítulo 1 deste trabalho.

Há estudos sobre a codificação de k -trees há pelo menos quatro décadas. Em 1970, Rényi e Renyi apresentaram uma codificação redundante (ou seja, não bijetiva) para um subconjunto de k -trees rotuladas que chamamos de k -trees de Rényi e que são definidas como segue:

Definição 19 (k -tree de Rényi). [12] Uma k -tree de Rényi R_k é uma k -tree enraizada com n vértices rotulados em $[1, n]$ e raiz $R = \{n - k + 1, n - k + 2, \dots, n\}$.

Entretanto, até onde sabemos, apenas em 2008 surgiu um código bijetivo para k -trees com algoritmos lineares de codificação e decodificação. Foram esses algoritmos, propostos por Caminiti *et al.* [4], que implementamos neste trabalho.

3.2 A solução de Caminiti *et al.*

O artigo “*Bijective Linear Time Coding and Decoding for k -Trees*” [4] apresenta um código bijetivo para k -trees rotuladas, juntamente a algoritmos lineares para realizar a codificação e a decodificação.

O código é formado por uma permutação de tamanho k e uma generalização do *Dandelion Code* [13], que consiste em $n - k - 2$ pares (onde n é o número de vértices) definidos no conjunto $\{(0, \varepsilon)\} \cup ([1, n - k] \times [1, k])$. Portanto, dizemos que a codificação das k -trees associa elementos em \mathcal{T}_k^n (conjunto das k -trees com n vértices) com elementos em:

$$\mathcal{A}_k^n = \binom{[1, n]}{k} \times (\{(0, \varepsilon)\} \cup ([1, n - k] \times [1, k]))^{n-k-2}$$

Caminiti *et al.* [4] mostra que a estrutura dessas *strings* que o *Dandelion Code* gera é essencial para garantir a bijetividade.

Os algoritmos consistem em uma série de transformações. Para compreendê-los, é necessário definir esqueleto de uma k -tree enraizada e árvore característica:

Definição 20 (esqueleto de uma k -tree enraizada). [4] O esqueleto de uma k -tree enraizada T_k com raiz R , denotado por $S(T_k, R)$, é definido da seguinte forma recursiva:

1. Se T_k é apenas o k -clique R , seu esqueleto é uma árvore com um único vértice R .
2. Dada uma k -tree enraizada T_k com raiz R , obtida por T'_k enraizada em R através da adição de um novo vértice v conectado a um k -clique K (ver definição 13), seu esqueleto $S(T_k, R)$ é obtido adicionando a $S(T'_k, R)$ um novo vértice $X = \{v\} \cup K$ e uma nova aresta (X, Y) , onde Y é o vértice de $S(T'_k, R)$ que contém K com uma distância mínima da raiz. Chamamos Y de pai de X .

Definição 21 (árvore característica). [4] A árvore característica $T(T_k, R)$ de uma k -tree enraizada T_k com raiz R é obtida rotulando os vértices e arestas de $S(T_k, R)$ da seguinte forma:

1. O vértice R é rotulado 0 e cada vértice $\{v\} \cup K$ é rotulado v ;

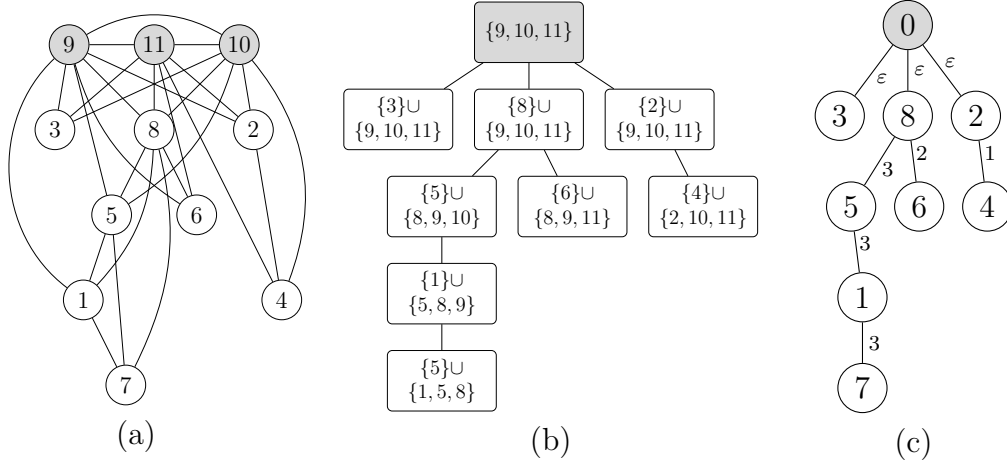


Figura 3.2: **(a)** Uma 3-tree de Rényi R_3 com 11 vértices e raiz $\{9, 10, 11\}$. **(b)** O esqueleto de R_3 . **(c)** A árvore característica de R_3 .

2. Cada aresta do vértice $\{v\} \cup K$ ao seu pai $\{v'\} \cup K'$ é rotulada com o índice do vértice em K' (visualizando-o como um conjunto ordenado) que não aparece em K . Quando o pai é R a aresta é rotulada ε .

Note que a existência de um único vértice em $K' \setminus K$ é garantida pela definição 20. De fato, v' precisa aparecer em K , caso contrário $K' = K$ e o pai de $\{v'\} \cup K'$ contém K . Isso contradiz o fato de que cada vértice em $S(T_k, R)$ é ligado à distância mínima da raiz.

A figura 3.2 mostra uma k -tree de Rényi com 11 vértices, seu esqueleto e sua árvore característica. O *Dandelion Code* generalizado correspondente a essa árvore é $[(0, \varepsilon), (2, 0), (8, 2), (8, 1), (1, 2), (5, 2)]$. A forma como codificamos e decodificamos árvores características usando esse código será vista a seguir, nos algoritmos de codificação e decodificação.

3.2.1 Codificação

O algoritmo para codificar uma k -tree rotulada consiste em cinco passos e tem complexidade $O(nk)$. Aqui apresentamos esse algoritmo indicando onde

cada um dos passos pode ser encontrado na nossa implementação.

ALGORITMO DE CODIFICAÇÃO

Entrada: uma k -tree T_k com n vértices

Saída: um código (Q, S) em \mathcal{A}_k^n

1. Identificar Q , o k -clique adjacente à folha de maior rótulo l_M de T_k ;
2. Através de um processo de re-rotulação ϕ (computado a partir de Q e detalhado a seguir), transformar T_k numa k -tree de Rényi R_k ;
3. Gerar a árvore característica T para R_k ;
4. Computar o *Dandelion Code* generalizado S para T ;
5. Remover da *string* obtida S o par correspondente a $\phi(l_M)$.

O algoritmo retorna o par (Q, S) computado durante esse processo.

Na nossa implementação, uma k -tree (estrutura definida no pacote `ktree`) é representada através de uma lista de adjacências (`Adj`) e um inteiro k (`K`).

O algoritmo de codificação é implementado pela função `CodingAlgorithm` do pacote `codec`. A seguir, detalhamos os cinco passos.

Passo 1. Primeiramente precisamos encontrar l_M , a folha de T_k com maior rótulo. Uma folha em uma k -tree consiste em um vértice de grau k , portanto basta iterar na lista de adjacências em ordem decrescente nos rótulos até encontrar um vértice com grau k . Isso foi implementado na função `FindLm`, localizada no pacote `ktree`.

Encontrado l_M , atribuímos a Q a lista `Adj[l_M]` (ver função `GetQ` do pacote `ktree`).

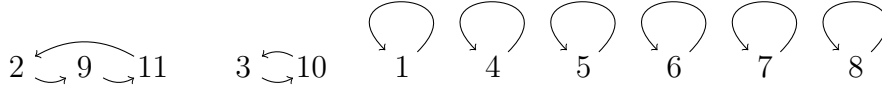


Figura 3.3: Representação gráfica da função ϕ computada para a 3-*tree* mostrada na figura 2.3.

Passo 2. Queremos transformar T_k numa k -*tree* de Rényi enraizada em Q . Para isso, precisamos definir uma permutação que associe os vértices de Q a $\{n - k + 1, n - k + 2, \dots, n\}$. A função de permutação, que chamamos de ϕ , é definida da seguinte forma:

1. Se q_i é o i -ésimo menor vértice em Q , fazemos $\phi(q_i) = n - k + i$;
2. Para cada $q \notin Q \cup \{n - k + 1, \dots, n\}$, fazemos $\phi(q) = q$;
3. O restante dos valores são usados para fechar os ciclos de permutação, ou seja, para cada $q \in \{n - k + 1, \dots, n\} \setminus Q$, fazemos $\phi(q) = i$ tal que $\phi^j(i) = q$ e j é maximizado.

Essa computação é implementada pela função `ComputePhi` no pacote `ktree`.

Usamos a função ϕ para re-rotular os vértices de T_k , obtendo a k -*tree* de Rényi R_k . A implementação desse processo foi realizada na função `Relabel` do pacote `ktree`.

A figura 3.3 mostra uma representação gráfica da função ϕ usada para re-rotular a 3-*tree* mostrada na figura 2.3 com $Q = \{2, 3, 9\}$ produzindo a k -*tree* de Rényi mostrada na figura 3.2(a).

Passo 3. As definições 20 e 21 sugerem algoritmos triviais para gerar a árvore característica T para a k -*tree* de Rényi R_k obtida no passo anterior por meio do seu esqueleto (o processo visto na figura 3.2).

Para garantir tempo linear, no entanto, o artigo de Caminiti *et al.* [4] sugere evitar a construção explícita do esqueleto $S(R_k)$ e construir os conjuntos de vértices e arestas de T separadamente.

Para computar o conjunto de vértices, identifica-se cliques maximais em R_k através da poda sucessiva das k -folhas de R_k . Esse processo pode ser visto na função `pruneRk` do pacote `characteristic`. Para cada vértice v podado, essa função guarda uma lista $K_v \subseteq \text{Adj}(v)$ dos exatamente k vértices adjacentes a v que ainda não foram podados.

Ao fim desse processo, que tem complexidade $O(nk)$, a k -tree de Rényi é reduzida apenas à sua raiz $R = \{n - k + 1, \dots, n\}$.

A partir das listas K_i ($i \in V$) e da ordem em que os vértices foram podados, constrói-se o conjunto das arestas num processo de complexidade $O(nk)$ detalhado no programa 7 do artigo [4] cuja implementação encontra-se na função `addEdges` do pacote `characteristic`.

Na nossa implementação, as arestas são representadas por duas listas (vetores), $p(v)$ e $l(v)$. Elas indicam para cada $v \in V(T)$, respectivamente, o pai de v na árvore e o rótulo da aresta $(p(v), v)$.

Passo 4. A ideia do *Dandelion Code* é enraizar a árvore T no vértice 0 e transformá-la para garantir a existência da aresta $(0, x)$. Por meio dessa transformação, o vetor de pais da árvore (transformada) vai conter duas informações inúteis (os pais de 0 e x), cuja eliminação leva a uma representação da árvore com $n - 2$ rótulos.

Escolhemos $x = \phi(\bar{q})$ onde $\bar{q} = \min\{v \notin Q\}$ e, enquanto $p(x) \neq 0$, fazemos sucessivas trocas $p(x) \leftrightarrow p(w)$, $l(x) \leftrightarrow l(w)$ escolhendo w como o vértice de maior rótulo no caminho entre 0 e x .

A implementação desse processo pode ser vista na função `Code` do pacote `dandelion`.

Ao final, o código S é dado por uma lista ordenada de pares $(p(v), l(v)) \forall v \in V(T) \setminus \{0, x\}$.

Passo 5. Como l_M foi escolhida como a folha de maior rótulo adjacente a Q , ela não é \bar{q} (porque $\bar{q} = \min\{v \notin Q\}$ e $n \geq k + 2$). A prova formal desse fato pode ser encontrada no Lema 1 do artigo [4]. Além disso, $\phi(l_M)$ não estava no caminho de 0 a $x = \phi(\bar{q})$ em T (porque é uma folha).

Como l_M é adjacente a Q , $\phi(l_M)$ é adjacente a 0. Portanto $(p(\phi(l_M)), l(\phi(l_M))) = (0, \varepsilon)$ pode ser removido da lista S de forma que o tamanho do código passe a ser $n - k - 2$. Isso é crucial para o código ser bijetivo.

O algoritmo retorna o par (Q, S) .

3.2.2 Decodificação

O algoritmo para decodificar um par $(Q, S) \in \mathcal{A}_k^n$ em uma k -tree rotulada T_k com n vértices consiste numa sequência de transformações inversas às transformações usadas no algoritmo de codificação. Aqui apresentamos esse algoritmo, de complexidade $O(nk)$, indicando onde cada um dos passos pode ser encontrado na nossa implementação.

ALGORITMO DE DECODIFICAÇÃO

Entrada: um código (Q, S) em \mathcal{A}_k^n

Saída: uma k -tree T_k com n vértices

1. Computar ϕ , \bar{q} , x e l_M (definidos como no algoritmo de codificação);
2. Inserir o par $(0, \varepsilon)$ correspondente a l_M em S e decodificar S para obter a árvore característica T ;
3. Reconstruir a k -tree de Rényi R_k a partir de T ;
4. Aplicar ϕ^{-1} a R_k para obter T_k .

O algoritmo de decodificação é implementado pela função `DecodingAlgorithm` do pacote `codec`. A seguir, detalhamos os quatro passos.

Passo 1. Para computar ϕ , \bar{q} , x e l_M , os procedimentos são exatamente os mesmos usados no algoritmo de codificação.

Passo 2. Como já computamos ϕ e l_M no passo anterior, inserimos o par $(0, \varepsilon)$ na posição $\phi(l_M)$ do vetor S .

O procedimento para decodificar o *Dandelion Code* numa árvore característica, implementado na função `Decode` do pacote `dandelion`, consiste em:

1. Construir o grafo a partir do código S , gerando vetores p (de pais) e l (de rótulos das arestas $(p(v), v)$);
2. Identificar todos os ciclos do grafo e guardar num vetor m , para cada ciclo, o vértice com maior rótulo;
3. Ordenar o vetor m em ordem crescente e iterar nele fazendo trocas $p(x) \leftrightarrow p(m_i)$, $l(x) \leftrightarrow l(m_i)$ (para $i = 1, \dots, |m|$).

A árvore característica T é dada pelo par (p, l) resultante desse processo.

Passo 3. A reconstrução da k -tree de Rényi R_k a partir de T foi implementada na função `RenyiKtreeFrom` do pacote `characteristic`.

O processo consiste em inicializar R_k com o k -clique $\{n - k + 1, \dots, n\}$ e percorrer T na ordem da busca em largura (a partir dos filhos do vértice de rótulo 0) para inserir vértices em R_k .

O programa 8 do artigo de Caminiti *et al.* [4] detalha esse passo.

Passo 4. Para transformar a k -tree de Rényi R_k na k -tree rotulada T_k , basta aplicar o inverso da permutação ϕ . Esse processo foi implementado na função `TkFrom` do pacote `ktree`.

3.3 Geração uniforme

Como comentamos no início deste capítulo, se temos uma codificação bijetiva que associa k -trees a *strings*, basta gerar *strings* aleatórias para gerar k -trees aleatórias.

Para gerar k -trees aleatórias de forma uniforme, usamos o código de Caminiti *et al.* [4] e o algoritmo linear para decodificar uma *string* em uma k -tree rotulada que apresentamos na seção 3.2.

As *strings* que estamos interessados em gerar são elementos do conjunto:

$$\mathcal{A}_k^n = \binom{[1, n]}{k} \times (\{(0, \varepsilon)\} \cup ([1, n-k] \times [1, k]))^{n-k-2}$$

A função que implementamos para gerar tais *strings* é `randomCode`, que recebe n e k como parâmetros e pertence ao pacote `generator`.

Primeiramente, ela sorteia Q em $\binom{[1, n]}{k}$ (e inicializa um *Dandelion Code* vazio):

```

1  C := &codec.Code{
2      rand.Perm(n)[:k],
3      &dandelion.DandelionCode{
4          make([]int, n-k-2),
5          make([]int, n-k-2),
6      },
7  }
8
9  sort.Ints(C.Q)
```

Depois, ela gera S sorteando $n-k-2$ pares em $\{(0, \varepsilon)\} \cup ([1, n-k] \times [1, k])$. Para gerar um par nesse intervalo de forma uniforme, gera-se um inteiro r no intervalo $[0, (n-k)k+1)$. Se $r = 0$, então o par é $(0, \varepsilon)$. Caso contrário, o par é dado por $(1 + \frac{r-1}{k}, (r-1) \bmod k)$:

```

1  for i := 0; i < n-k-2; i++ {
2      r := rand.Intn((n-k)*k + 1)
3      if r == 0 {
4          C.S.P[i] = 0
5          C.S.L[i] = characteristic.E
6      } else {
7          r--
8          C.S.P[i] = 1 + r/k
9          C.S.L[i] = r % k
10     }
11 }
```

Decodificamos o código usando o algoritmo de decodificação apresentado na seção 3.2 para transformar essa string $(Q, S) \in \mathcal{A}_k^n$ em uma k -tree rotulada.

3.4 Utilitários

Para exemplificar como se usa a biblioteca desenvolvida nas seções anteriores, foram desenvolvidos três utilitários que se encontram no pacote `examples`: `code-ktree`, `decode-ktree` e `generate-ktree`.

Eles permitem codificar/decodificar k -trees e gerar k -trees aleatórias.

3.4.1 code-ktree

O utilitário `code-ktree` serve para codificar k -trees usando o algoritmo da subseção 3.2.1. Sua entrada deve ser dada no formato¹:

```

1  n k
2  m
3  x_1 y_1
```

¹A leitura da entrada despreza espaços e quebras de linha.

```

4 || ...
5 || x_m y_m

```

Onde:

- n é o número de vértices;
- k é o parâmetro k da k -tree;
- m é o número de arestas;
- $x_i \ y_i$ corresponde à i -ésima aresta ($0 \leq x_i, y_i < n$).

Um exemplo de entrada equivalente à k -tree da figura 2.3(a) é:

```

1 || 11 3
2 || 27
3 || 0 1 0 4 0 6 0 7
4 || 1 2 1 4 1 5 1 7 1 8 1 9 1 10
5 || 2 3 2 4 2 7 2 8 2 9 2 10
6 || 3 8 3 10 4 6
7 || 4 7
8 || 5 7 5 8
9 || 6 7
10 || 7 8
11 || 8 9 8 10

```

A saída desse utilitário é um par (Q, S) no formato de entrada esperado pelo utilitário `decode-ktree`, que será descrito a seguir.

3.4.2 decode-ktree

O utilitário `decode-ktree` serve para decodificar um código (Q, S) numa k -tree usando o algoritmo da subseção 3.2.2. Sua entrada deve ser dada no formato:

```

1 || k
2 || Q_1
3 || ...
4 || Q_k
5 || s
6 || p_1 l_1
7 || ...
8 || p_s l_s

```

Onde:

- k é o tamanho de Q ;
- Q_i corresponde ao i -ésimo valor em Q ;
- s é o tamanho do *Generalized Dandelion Code*, $|S|$;
- $p_i \ l_i$ corresponde ao i -ésimo valor em S .

Um exemplo de entrada equivalente ao código gerado pela k -tree da figura 2.3(a) é:

```

1 || 3
2 || 1 2 8
3 || 6
4 || 0 -1
5 || 2 0
6 || 8 2
7 || 8 1
8 || 1 2
9 || 5 2

```

A saída desse utilitário é uma k -tree no formato de entrada esperado pelo utilitário `code-ktree`.

3.4.3 generate-ktree

O utilitário `generate-ktree` serve para gerar uma *k-tree* aleatória usando o algoritmo desenvolvido na seção 3.3.

Sua entrada deve ser dada no formato:

```
1 || n k
```

Sua saída é uma *k-tree* com n vértices no formato de entrada esperado pelo utilitário `code-ktree`.

3.5 Testes, experimentos e resultados

3.5.1 Testes unitários e cobertura

Como escrevemos na introdução deste trabalho, um dos motivos pelos quais escolhemos a linguagem *Go* para a implementação foi a facilidade para escrever testes.

Todos os pacotes desenvolvidos neste trabalho possuem testes unitários que podem ser executados usando o utilitário `go test`:

```
1 | $ go get github.com/tmadeira/tcc/...
2 | $ go test -v github.com/tmadeira/tcc/...
3 | === RUN    TestTreeFrom
4 | --- PASS: TestTreeFrom (0.00s)
5 | === RUN    TestRenyiKtreeFrom
6 | --- PASS: TestRenyiKtreeFrom (0.00s)
7 | PASS
8 | ok      github.com/tmadeira/tcc/characteristic 0.002s
9 | === RUN    TestCodingAlgorithm
10 | --- PASS: TestCodingAlgorithm (0.00s)
11 | === RUN    TestDecodingAlgorithm
12 | --- PASS: TestDecodingAlgorithm (0.00s)
```

```
13 | PASS
14 | ok      github.com/tmadeira/tcc/codec 0.017s
15 | === RUN   TestCodeFig2C
16 | --- PASS: TestCodeFig2C (0.00s)
17 | === RUN   TestDecodeFig2C
18 | --- PASS: TestDecodeFig2C (0.00s)
19 | === RUN   TestDecodeFig3
20 | --- PASS: TestDecodeFig3 (0.00s)
21 | PASS
22 | ok      github.com/tmadeira/tcc/dandelion 0.002s
23 | === RUN   TestRandomKtree
24 | --- PASS: TestRandomKtree (0.03s)
25 | PASS
26 | ok      github.com/tmadeira/tcc/generator 0.029s
27 | === RUN   TestGetQ
28 | --- PASS: TestGetQ (0.00s)
29 | === RUN   TestComputePhi
30 | --- PASS: TestComputePhi (0.00s)
31 | === RUN   TestRelabel
32 | --- PASS: TestRelabel (0.00s)
33 | === RUN   TestRkFrom
34 | --- PASS: TestRkFrom (0.00s)
35 | === RUN   TestTkFrom
36 | --- PASS: TestTkFrom (0.00s)
37 | PASS
38 | ok      github.com/tmadeira/tcc/ktree 0.002s
```

Com efeito, 96% das linhas do código são cobertas por testes, como mostra o relatório da ferramenta *Coveralls*².

²Esse relatório pode ser visto em: <https://coveralls.io/github/tmadeira/tcc?branch=master>

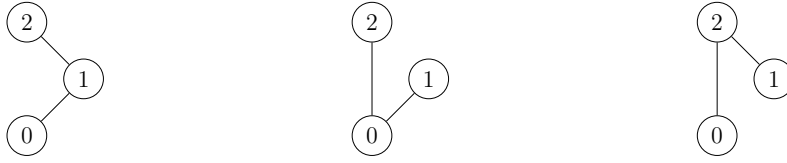


Figura 3.4: Representação das três 1-trees rotuladas distintas com $n = 3$ vértices.

3.5.2 Experimentos e resultados

Corretude e uniformidade

Para mostrar que nossa implementação gera k -trees aleatórias corretamente e uniformemente, realizamos dezenas de milhares de testes com n e k pequenos.

Escrevemos um pequeno *script* em Bash para nos auxiliar nesse experimento. Ele usa o utilitário `generate-ktree` para gerar 10000 k -trees com parâmetros (n, k) constantes e imprime quantas vezes cada k -tree diferente foi gerada:

```
1 i=0
2 while [ $i -lt 10000 ]; do
3     echo $N $K | generate-ktree | xargs echo
4     i=$((i+1))
5 done | sort | uniq -c
```

Com $n = 3$ e $k = 1$, existem 3 k -trees rotuladas distintas, como mostra a figura 3.4.

Ao executar o *script* com $N=3$ $K=1$ esperamos portanto que as três 1-trees apareçam com uma frequência similar. O resultado que obtivemos foi:

```
1 3320 3 1 2 0 1 0 2
2 3345 3 1 2 0 1 1 2
3 3335 3 1 2 0 2 1 2
```

O primeiro inteiro que aparece em cada linha é a quantidade de vezes que a k -tree apareceu e o restante é a k -tree gerada no formato de saída do

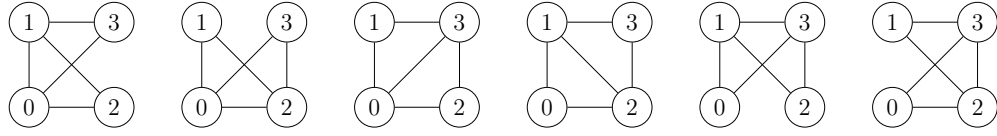


Figura 3.5: Representação das seis 2-trees rotuladas distintas com $n = 4$ vértices.

utilitário `generate-ktree` (sem quebras de linha).

Como a frequência de cada uma das 3 k -trees com $n = 3$ e $k = 1$ está similar, o experimento mostra que o algoritmo gera k -trees aleatórias de forma uniforme.

Testes com outros pares (n, k) também mostram frequências similares, comprovando a uniformidade. Por exemplo, existem 6 2-trees com $n = 4$ vértices, como pode-se ver na figura 3.5. Rodando o *script* com $N=4$ $K=2$ obtivemos:

```
1 | 1703 4 2 5 0 1 0 2 0 3 1 2 1 3
2 | 1627 4 2 5 0 1 0 2 0 3 1 2 2 3
3 | 1573 4 2 5 0 1 0 2 0 3 1 3 2 3
4 | 1709 4 2 5 0 1 0 2 1 2 1 3 2 3
5 | 1717 4 2 5 0 1 0 3 1 2 1 3 2 3
6 | 1671 4 2 5 0 2 0 3 1 2 1 3 2 3
```

E com $N=5$ $K=3$ obtivemos:

```
1 | 970 5 3 9 0 1 0 2 0 3 0 4 1 2 1 3 1 4 2 3 2 4
2 | 1023 5 3 9 0 1 0 2 0 3 0 4 1 2 1 3 1 4 2 3 3 4
3 | 1009 5 3 9 0 1 0 2 0 3 0 4 1 2 1 3 1 4 2 4 3 4
4 | 1014 5 3 9 0 1 0 2 0 3 0 4 1 2 1 3 2 3 2 4 3 4
5 | 994 5 3 9 0 1 0 2 0 3 0 4 1 2 1 4 2 3 2 4 3 4
6 | 1019 5 3 9 0 1 0 2 0 3 0 4 1 3 1 4 2 3 2 4 3 4
7 | 1008 5 3 9 0 1 0 2 0 3 1 2 1 3 1 4 2 3 2 4 3 4
8 | 1000 5 3 9 0 1 0 2 0 4 1 2 1 3 1 4 2 3 2 4 3 4
9 | 978 5 3 9 0 1 0 3 0 4 1 2 1 3 1 4 2 3 2 4 3 4
10 | 985 5 3 9 0 2 0 3 0 4 1 2 1 3 1 4 2 3 2 4 3 4
```


Capítulo 4

Aprendizado de redes bayesianas

4.1 Aprendendo redes bayesianas

Neste trabalho, quando falamos em aprender redes bayesianas estamos nos referindo ao processo de inferir a estrutura (ou seja, o DAG) de uma rede bayesiana a partir de dados. Como mostra Chickering [6], este é um problema NP-completo.

A continuar [9].

Capítulo 5

Conclusão

Ainda não foi escrita.

Referências Bibliográficas

- [1] Stefan Arnborg and Andrzej Proskurowski. Linear time algorithms for np-hard problems restricted to partial k-trees. *Discrete Applied Mathematics*, 23:11–24, 1989.
- [2] Hans L. Bodlaender. Treewidth: Structure and algorithms. *Structural Information and Communication Complexity*, 4474:11–25, 2007.
- [3] John A. Bondy and Uppaluri S. R. Murty. *Graph Theory*. Springer, 2008.
- [4] Saverio Caminiti, Emanuele G. Fusco, and Rossella Petreschi. Bijective linear time coding and decoding for k -trees. *Theory of Computing Systems*, 46:284–300, 2010.
- [5] Arthur Cayley. A theorem on trees. *Quart J. Math*, 23:376–378, 1889.
- [6] David Maxwell Chickering. *Learning Bayesian Networks is NP-Complete*, pages 121–130. Springer New York, New York, NY, 1996.
- [7] Frank Harary and Edgar M. Palmer. On acyclic simplicial complexes. *Mathematika*, 15:115–122, 1968.
- [8] Daphne Koller and Nir Friedman. *Probabilistic Graphical Models: Principles and Techniques*. The MIT Press, 2009.

- [9] Siqi Nie, Cassio P. de Campos, and Qiang Ji. *Learning Bounded Tree-Width Bayesian Networks via Sampling*, pages 387–396. Springer International Publishing, Cham, 2015.
- [10] Siqi Nie, Denis Deratani Mauá, Cassio Polpo de Campos, and Qiang Ji. Advances in learning bayesian networks of bounded treewidth. *CoRR*, abs/1406.1411, 2014.
- [11] Heinz Prüfer. Neuer beweis eines satzes über permutationen. *Archiv der Mat. und Physik*, 27:142–144, 1918.
- [12] C. Rényi and A. Rényi. The prüfer code for k -trees. *Combinatorial Theory and its Applications*, pages 945–971, 1970.
- [13] Ömer Eğecioğlu and J. B. Remmel. Bijections for cayley trees, spanning trees, and their q -analogues. *Journal of Combinatorial Theory*, 42:15–30, 1986.