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

Usando aleatoriedade para lidar com problemas computacionalmente difíceis em grafos

Guilherme Vinicius Ferreira de Assis
Supervisor: Prof.º Dr.º Fábio Happ Botler

Monografia

MAC0499 – Trabalho de Formatura Supervisionado

São Paulo
2025

O conteúdo deste trabalho é publicado sob a licença CC BY 4.0.

(Creative Commons Attribution 4.0 International License)

Resumo

ASSIS, G. V. F. de. **Usando aleatoriedade para lidar com problemas computacionalmente difíceis em grafos**. 2025. Monografia (Bacharelado em Ciência da Computação) – Instituto de Matemática e Estatística, Universidade de São Paulo, São Paulo, 2025.

Muitos problemas relevantes em computação são computacionalmente difíceis – não se conhecem algoritmos polinomiais para resolvê-los –, o que motiva a busca por abordagens alternativas, como algoritmos para subclasses específicas, soluções aproximadas, paralelização e, particularmente, o uso de aleatoriedade. Este trabalho apresenta uma revisão bibliográfica sobre a aplicação de aleatoriedade em problemas difíceis em grafos, abordando duas formas principais: algoritmos aleatorizados e análise probabilística de algoritmos. Na primeira, considera-se algoritmos que têm acesso a bits aleatórios, permitindo o desenvolvimento de soluções mais simples, embora a aleatoriedade provavelmente não aumente o poder computacional. Na segunda, substituímos, de forma justificada, a análise de pior caso pela análise de caso médio ou pela análise suavizada, possibilitando resultados positivos para problemas complexos sob essa perspectiva.

Palavras-chave: Algoritmos aleatorizados. Análise probabilística de algoritmos. Complexidade computacional. Grafos.

Abstract

ASSIS, G. V. F. de. **Using randomness to address computationally hard problems in graphs**. 2025. Undergraduate thesis (Bachelor's degree in Computer Science) — Institute of Mathematics and Statistics, University of São Paulo, São Paulo, 2025.

Many relevant problems in computer science are computationally hard – no polynomial-time algorithms are known to solve them – which motivates the search for alternative approaches, such as algorithms for specific subclasses, approximation methods, parallelization, and particularly the use of randomness. This work presents a literature review on the use of randomness to tackle hard graph problems, addressing two main approaches: randomized algorithms and probabilistic analysis of algorithms. In the first, algorithms have access to random bits, allowing the development of simpler solutions, although randomness is unlikely to increase computational power. In the second, for good reason reasons, worst-case analysis is replaced by average-case analysis or smoothed analysis, allowing for positive results in complex problems from this perspective.

Keywords: Randomized algorithms. Probabilistic analysis of algorithms. Computational complexity. Graphs.

Sumário

1 Introdução **4**

2 Algoritmos aleatorizados **6**

2.1 Classificação de algoritmos 7

2.1.1 Monte Carlo 8

2.1.2 Las Vegas 13

2.1.3 Conversão entre algoritmos Monte Carlo e Las Vegas 19

2.2 Modelo computacional e classes de complexidade 21

2.3 Pseudoaleatoriedade e derandomização 27

3 Análise probabilística de algoritmos **32**

3.1 Grafos aleatórios 33

3.2 Análise de caso médio 37

3.2.1 Ciclo hamiltoniano em tempo esperado polinomial 38

3.2.2 Lema Minimax de Yao 47

3.2.2.1 Analisando algoritmos através de jogos . 50

3.3 Análise suavizada 52

4 Conclusão **57**

1 Introdução

Uma das questões fundamentais na análise de algoritmos e na complexidade computacional é identificar quais problemas são fáceis ou difíceis de resolver, adotando estratégias apropriadas para cada caso. Embora muitas questões fundamentais permaneçam abertas, como a conjectura $\mathcal{P} \neq \mathcal{NP}$, pesquisadores desenvolveram diversas ferramentas para lidar com problemas computacionalmente difíceis, para os quais não se conhecem algoritmos polinomiais. Entre elas estão algoritmos eficientes para subclasses específicas, aproximações, paralelização, análise parametrizada da complexidade e heurísticas [1].

A aleatoriedade surge na computação na década de 1950 como uma ferramenta promissora, cuja compreensão tem promovido avanços significativos em complexidade e desenvolvimento de algoritmos. O estudo da aleatoriedade computacional oferece uma perspectiva diferente das abordagens matemáticas clássicas, como será brevemente discutido na Seção 2.3. Entre suas contribuições, destacam-se resultados fundamentais em criptografia e profundas contribuições conceituais em complexidade computacional, reconhecidas recentemente pelo prêmio Turing concedido a Avi Wigderson [2].

Dada a relevância da aleatoriedade no contexto computacional, este trabalho apresenta uma revisão bibliográfica introdutória sobre o uso de aleatoriedade para lidar com problemas computacionalmente difíceis, com ênfase em grafos. De forma geral, a aleatoriedade pode ser incorporada diretamente nos algoritmos, na esperança de ampliar o que podemos computar em tempo polinomial, ou aplicada na análise probabilística de algoritmos, buscando resultados mais favoráveis que os sugeridos pela análise de pior caso.

Na Seção 2, exploramos algoritmos com acesso a bits aleatórios, incluindo exemplos introdutórios e a investigação do poder computacional desses algoritmos através de classes de complexidade probabilísticas. Na Seção 3, discutimos a motivação para substituir a análise de pior caso por abordagens alternativas, como caso médio ou suavizada, apresentando os métodos correspondentes e seus principais resultados positivos. Também examinaremos a relação curiosa entre o desempenho médio de algoritmos determinísticos e o de algoritmos aleatorizados. Embora o trabalho se restrinja ao contexto de grafos, muitos dos resultados apresentados se aplicam a outros domínios de problemas. Ao longo do texto, usamos conceitos e termos convencionais de teoria da computação, grafos e probabilidade.

Este trabalho concentra-se exclusivamente na complexidade de tempo de algoritmos *offline*, cujas entradas são totalmente conhecidas antecipadamente. Entretanto, a aleatoriedade também pode ser aplicada para reduzir a memória utilizada ou resolver problemas *online*. Detalhes de implementação de estruturas de dados são omitidos, mantendo o foco nas ideias algorítmicas, e algumas suposições simplificadoras são feitas implicitamente, como consultas a arestas de grafos em tempo constante, facilmente realizáveis em implementações concretas.

Por fim, este é um trabalho estritamente introdutório. Para um estudo mais aprofundado sobre aleatoriedade em computação, veja [3,4,5].

2 Algoritmos aleatorizados

Uma maneira natural de incorporar aleatoriedade no contexto de algoritmos é por meio de *algoritmos aleatorizados* (ou *probabilísticos*), nos quais certo grau de aleatoriedade é integrado à lógica de execução. Contrário ao que uma primeira impressão possa sugerir, o comportamento caótico introduzido pelo acaso pode, em muitos casos, ser explorado de forma vantajosa.

A origem dos algoritmos aleatorizados remonta aos métodos de Monte Carlo, empregados em análise numérica, física estatística e simulação, desenvolvidos no final da década de 1940[1] [6]. Atualmente, seu uso estende-se a diversas outras áreas, incluindo sistemas distribuídos, hashing, criptografia e otimização combinatória [7].

Formalmente, um algoritmo A é dito *aleatorizado* se utiliza uma sequência $S \in \{0,1\}^d$ composta por d bits aleatórios, independentes e imparciais[2] como entrada auxiliar para guiar seu comportamento. Algoritmos aleatorizados são úteis porque, em muitos casos, são significativamente mais rápidos e/ou mais simples que alternativas determinísticas. Contudo, tais vantagens têm um custo: o resultado pode ter uma pequena probabilidade de erro[3], ou a eficiência do algoritmo pode ser garantida apenas com certa probabilidade.

A análise desses algoritmos busca então estabelecer limitantes para o valor esperado de uma medida de desempenho – como o tempo de execução ou a correção da saída – que sejam válidos para qualquer entrada. Essa medida é, então, uma variável aleatória, cuja distribuição depende das escolhas guiadas pelos bits aleatórios.

Em certo sentido, um algoritmo determinístico eficiente pode ser visto como um caso particular de algoritmo aleatorizado: correto e eficiente com probabilidade 1. Isso sugere que o modelo de algoritmos aleatorizados é estritamente mais geral do que o determinístico.

A aleatoriedade oferece um meio eficaz de explorar o espaço de soluções de forma não trivial, o que sugere ser possível construir algoritmos aleatorizados eficientes para problemas difíceis com a utilização de passos aleatórios. Embora essa abordagem possa ocasionalmente produzir respostas incorretas, muitas vezes é possível controlar rigorosamente a probabilidade de erro, resultando em algoritmos *probabilisticamente corretos*.

Já sob a ótica da eficiência algorítmica, algoritmos aleatorizados podem ser especialmente eficientes. Por exemplo, suponha que um adversário escolha a entrada de modo a maximizar o tempo de execução do algoritmo,

[1] Tais técnicas foram idealizadas por Stanisław Ulam com o advento dos primeiros computadores eletrônicos. O nome faz referência ao famoso cassino de Monte Carlo, em Mônaco, associado aos hábitos de jogo do tio de Ulam.

[2] Note que os bits seguem uma distribuição de probabilidade uniforme. Isso é suficiente para gerar todas as distribuições de probabilidade usualmente consideradas [8].

[3] Ainda que possa parecer contraproducente projetar um algoritmo que ocasionalmente produza respostas incorretas, uma taxa de erro suficientemente pequena pode ser aceitável se proporcionar uma redução significativa no tempo de execução.

como na análise de pior caso. As decisões aleatórias impedem que o adversário preveja o comportamento do algoritmo, tornando impossível escolher antecipadamente uma entrada que garanta o pior desempenho. Por esse motivo, certos algoritmos aleatorizados são *probabilisticamente eficientes*, isto é, seu tempo de execução esperado é eficiente.

É importante destacar que essa abordagem difere da análise de caso médio (veja Seção 3.2), na qual a variação do desempenho decorre de uma distribuição de probabilidade sobre as entradas do problema, e não da aleatoriedade interna do algoritmo.

Na Seção 2.1, examinaremos com mais detalhe esses dois principais tipos de algoritmos aleatorizados: Monte Carlo e Las Vegas.

Em seguida, na Seção 2.2, apresentaremos os modelos computacionais probabilísticos e discutiremos as classes de complexidade computacional definidas sobre algoritmos aleatorizados e suas relações com classes clássicas, como $\mathcal{P}$ e $\mathcal{NP}$.

A obtenção dos bits aleatórios utilizados nesses algoritmos é, por si só, um problema complexo[4]. Como de costume, assumiremos que os algoritmos aqui apresentados e analisados operam no modelo computacional RAM[5], agora estendido: por simplicidade, consideramos que o modelo dispõe de acesso em tempo constante a uma fonte perfeita de números inteiros aleatórios, uniformemente distribuídos em um intervalo de cardinalidade polinomial no tamanho da entrada[6]. As questões relacionadas à obtenção de bits aleatórios serão discutidas com mais detalhes na Seção 2.3.

Um resultado fundamental da teoria de algoritmos aleatorizados, o Lema Minimax de Yao, que estabelece uma conexão direta entre algoritmos aleatorizados e a análise de caso médio, é apresentado posteriormente na Seção 3.2.2.

[4] Bits aleatórios são um recurso computacional não trivial. Na prática, usam-se geradores pseudoaleatórios (veja Seção 2.3) em vez de aleatoriedade genuína.

[5] O modelo RAM assume um computador com acesso instantâneo à memória e capacidade de realizar operações aritméticas básicas em tempo constante. Trata-se do modelo padrão na análise de algoritmos.

[6] O limitante no tamanho do intervalo garante que apenas um número polinomial de bits aleatórios seja necessário, permitindo que o modelo seja simulado em tempo polinomial por uma máquina de Turing probabilística (cuja definição é apresentada na Seção 2.2), mantendo assim a consistência teórica dos resultados sobre complexidade computacional probabilística.

2.1 Classificação de algoritmos

Os algoritmos aleatorizados podem ser classificados de acordo com qual aspecto de seu comportamento é tratado como variável aleatória. Quando a validade da resposta depende das escolhas probabilísticas, o algoritmo é do tipo *Monte Carlo*. Quando, ao contrário, a duração da execução é que varia de acordo com as decisões aleatórias, o algoritmo é classificado como *Las Vegas*[7].

Embora seja possível combinar as duas abordagens em um mesmo algo-

[7] Um algoritmo aleatorizado que fosse sempre correto e eficiente seria, na verdade, um algoritmo determinístico eficiente. Portanto, é necessário abrir mão de um desses aspectos.

ritmo – os quais são, em geral, também classificados como do tipo Monte Carlo –, por simplicidade, estudaremos apenas algoritmos que se enquadram inteiramente em uma ou outra categoria.

2.1.1 Monte Carlo

A seguir, apresentamos a definição formal de um algoritmo Monte Carlo[8].

[8] A denominação *Monte Carlo* tem origem nos métodos de Monte Carlo, empregados para resolver problemas matemáticos por meio da aproximação de resultados via amostragem estatística computacional.

Definição 1 (*Algoritmo Monte Carlo*)**.** *Um algoritmo aleatorizado A é dito Monte Carlo se existe $\varepsilon \in (0,1)$ tal que, para toda entrada x, $A(x)$ retorna uma resposta correta com probabilidade ao menos $1-\varepsilon$, em que ε representa a probabilidade de erro.*

O tempo de execução de um algoritmo de Monte Carlo é, em geral, determinístico. Contudo, estamos particularmente interessados em algoritmos Monte Carlo eficientes.

No caso de problemas de decisão, um algoritmo Monte Carlo pode apresentar *erro unilateral* ou *erro bilateral*[9]:

[9] Perceba que a definição de algoritmo Monte Carlo ainda se aplica: se ε_S é a probabilidade de erro ao responder SIM e ε_N é a probabilidade de erro ao responder NÃO, simplesmente definimos $\varepsilon = \max\{\varepsilon_S, \varepsilon_N\}$.

- O erro é unilateral quando a probabilidade de erro é nula para uma das duas possíveis respostas (SIM ou NÃO). Isto é, sempre que o algoritmo responde SIM (ou, analogamente, NÃO), essa resposta é necessariamente correta.
- Dizemos que o erro é bilateral quando existe probabilidade não nula de erro tanto ao responder SIM quanto ao responder NÃO.

Para ilustrar esses conceitos, apresentaremos um algoritmo aleatorizado simples que aproxima o *problema do Corte Máximo*[10] com alta probabilidade[11].

[10] Mesmo em sua versão simples (*i.e.*, sem pesos associados às arestas), o problema do Corte Máximo é $\mathcal{NP}$-completo [9].

[11] Uma sequência de eventos $\{\mathcal{E}_n\}_{n\in\mathbb{N}}$ ocorre com *alta probabilidade* se $\Pr(\mathcal{E}_n) \to 1$ quando $n \to \infty$.

Dado um grafo $G = (V, E)$, um *corte* é um subconjunto $S \subseteq E$ induzido por uma bipartição (X, Y) dos vértices V, de modo que S consiste de todas as arestas que possuem uma extremidade em X e outra em Y.

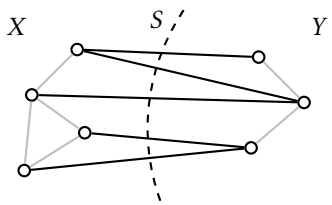


Figura 1: Conjunto de arestas S (em preto) que forma um corte definido por uma bipartição dos vértices do grafo.

Enunciamos então o problema proposto:

Corte Máximo

Dado grafo $G = (V, E)$, encontre um corte $S \subseteq E$ de cardinalidade máxima.

Apresentamos então o seguinte resultado clássico:

Teorema 1. *Seja $G = (V, E)$ um grafo com m arestas. Então existe um corte $S \subseteq E$ tal que $|S| \geq \lfloor m/2 \rfloor$.*

Demonstração. Considere uma bipartição aleatória (X, Y) de V, em que cada vértice é atribuído a X ou Y de forma independente, com probabilidade $1/2$ cada.

Seja S o corte induzido por essa bipartição e, para cada aresta $e \in E$, definimos $\mathbb{1}_e$ como a variável indicadora do evento $e \in S$.

Como $\mathbb{E}[\mathbb{1}_e] = 1/2$, pela linearidade da esperança, temos,

$$\mathbb{E}[|S|] = \mathbb{E}\left[\sum_{e \in E} \mathbb{1}_e\right] = \sum_{e \in E} \mathbb{E}[\mathbb{1}_e] = \frac{m}{2}.$$

Logo, segue que, com probabilidade positiva, existe um corte S tal que $|S| \geq \lfloor m/2 \rfloor$. □

Uma questão natural é se o método acima[12] pode ser adaptado para construir um algoritmo aleatorizado que produza, com alta probabilidade, um corte de tamanho ao menos $\lfloor m/2 \rfloor$ em qualquer grafo[13].

Algoritmo 1 (*Corte aleatório*)

A função CorteAleatório recebe um grafo G e retorna o corte induzido por uma bipartição aleatória dos vértices. A função auxiliar BitAleatório retorna um bit escolhido uniformemente ao acaso.

Entrada: Grafo $G = (V, E)$, com n vértices e m arestas.

Saída: Um corte $S \subseteq E$ tal que $|S| \geq \lfloor m/2 \rfloor$.

```
CorteAleatório(G)
1.  X, Y ← ∅
2.  Para cada vértice v ∈ V faça
3.      Se BitAleatório() então
4.          X ← X ∪ {v}
5.      Senão
6.          Y ← Y ∪ {v}

7.  S ← ∅
```

[12] A técnica empregada na demonstração é conhecida como *método probabilístico*, amplamente utilizada em combinatória para demonstrar, de forma não construtiva, a existência de objetos com certas propriedades desejadas [10].

[13] Isso corresponderia a uma 0, 5–aproximação para o problema do Corte Máximo, uma vez que o número máximo possível de arestas em um corte é m.

```
8.  Para cada aresta e = {u, v} ∈ E faça
9.      Se (u ∈ X e v ∈ Y) ou (u ∈ Y e v ∈ X) então
10.         S ← S ∪ {e}

11. Retorne S
```

O Algoritmo 1 tem tempo de execução determinístico $\Theta(n+m)$, mas produz um resultado aleatório: o corte S retornado pode não satisfazer $|S| \geq \lfloor m/2 \rfloor$. Isto o caracteriza como do tipo Monte Carlo.

Teorema 2. *Dado um grafo $G = (V, E)$ com m arestas, o Algoritmo 1 retorna um corte $S \subseteq E$ com tamanho $|S| \geq \lfloor m/2 \rfloor$ com probabilidade de erro $\varepsilon \leq m/(m+2)$.*

Demonstração. Queremos estimar

$$\varepsilon = \Pr(|S| < \lfloor m/2 \rfloor) = \Pr(|S| \leq \lfloor m/2 \rfloor - 1),$$

já que $|S|$ é inteiro.

Considere a variável aleatória $X = m - |S| \geq 0$. Aplicando a desigualdade de Markov[14], temos:

$$\Pr(|S| \leq \lfloor m/2 \rfloor - 1) = \Pr(X \geq \lceil m/2 \rceil + 1) \leq \frac{\mathbb{E}[X]}{\lceil m/2 \rceil + 1} = \frac{m - \mathbb{E}[|S|]}{\lceil m/2 \rceil + 1}.$$

[14] Se X é uma variável aleatória não negativa, a desigualdade de Markov estabelece que $\Pr(X \geq \lambda) \leq \mathbb{E}[X]/\lambda$.

Sabemos que $\mathbb{E}[|S|] = m/2$ pela demonstração do Teorema 1. Logo,

$$\varepsilon \leq \frac{m - m/2}{\lceil m/2 \rceil + 1} \leq \frac{m}{m+2}.$$

□

Embora o limitante pareça fraco[15], podemos reduzir arbitrariamente a probabilidade de erro aplicando uma técnica conhecida como *amplificação*.

[15] Note que $\varepsilon \to 1$ quando $m \to \infty$.

Suponha que executemos k vezes o Algoritmo 1 de forma independente, mantendo o maior corte obtido entre as execuções. Mostraremos que essa estratégia reduz exponencialmente a probabilidade de erro.

Algoritmo 2 (*Maior corte aleatório*)

A função MaiorCorteAleatório recebe um grafo G e um inteiro k, retornando o maior corte encontrado em k execuções independentes de CorteAleatório.

Entrada: Grafo $G = (V, E)$, com n vértices e m arestas.

Saída: Um corte $S \subseteq E$ tal que $|S| \geq \lfloor m/2 \rfloor$.

```
MaiorCorteAleatório(G, k)
1.  S ← ∅
2.  Para i = 1 até k faça
3.      S_i ← CorteAleatório(G)
4.      Se |S_i| > |S| então
5.          S ← S_i
6.  Retorne S
```

Lema 1. *Dado um grafo $G = (V, E)$ com m arestas, o Algoritmo 2 retorna um corte corte $S \subseteq E$ com $|S| \geq \lfloor m/2 \rfloor$ com probabilidade de erro*

$$\varepsilon \leq \left(\frac{m}{m+2}\right)^k.$$

Demonstração. Seja S_i o corte obtido na i-ésima execução de CorteAleatório. Pela independência das execuções, segue que:

$$\begin{aligned}
\varepsilon &= \Pr(|S_1| < \lfloor m/2 \rfloor \cap \cdots \cap |S_k| < \lfloor m/2 \rfloor), \\
\varepsilon &= \prod_{i=1}^{k} \Pr(|S_i| < \lfloor m/2 \rfloor), \\
\varepsilon &\leq \left(\frac{m}{m+2}\right)^k.
\end{aligned}$$

□

Se escolhermos $k \gg m$, então, pelo Lema 1 e usando a desigualdade $1 - x \leq e^{-x}$, obtemos:

$$\varepsilon \leq \left(1 - \frac{2}{m+2}\right)^k \leq e^{-\frac{2k}{m+2}} \to 0,$$

quando $m \to \infty$.

Corolário 1. *Para $k \gg m$, o Algoritmo 2 produz, com alta probabilidade, uma $0,5$-aproximação para o problema do Corte Máximo, em tempo $\Theta(k \cdot (n+m))$.*

Em particular, k pode ser escolhido como uma função polinomial de m, e o algoritmo resultante mantém-se polinomial.

Assim, a amplificação permite reduzir a probabilidade de erro arbitrariamente, ao custo de aumentar o tempo de execução. Para amplificar algoritmos de decisão, basta executar o algoritmo k vezes de forma independente e retornar a resposta majoritária. Em problemas de otimização, é natural executar o algoritmo diversas vezes e devolver a melhor solução obtida. Em todos os casos, a amplificação baseia-se em argumentos probabilísticos semelhantes, que levam ao seguinte resultado geral.

Teorema 3. *Seja A um algoritmo Monte Carlo com probabilidade de erro ε_A. Para qualquer ε tal que $0 < \varepsilon < \varepsilon_A$, é possível construir, por repetições independentes de A, um novo algoritmo Monte Carlo B cuja probabilidade de erro ε_B satisfaz $\varepsilon_B \leq \varepsilon$.*

Uma primeira ideia é investigar se a técnica de amplificação poderia tornar o Algoritmo 2 capaz de resolver de forma ótima o problema do Corte Máximo com alta probabilidade em tempo polinomial. Mostraremos, contudo, que isso não é possível sem hipóteses adicionais sobre o grafo de entrada.

Teorema 4. *Dado um grafo $G = (V, E)$ com n vértices, o Algoritmo 1 resolve o problema do Corte Máximo com probabilidade de erro*

$$\varepsilon \leq 1 - \frac{1}{2^{n-1}}.$$

Demonstração. Existem 2^{n-1} bipartições distintas de V, todas equiprováveis. A probabilidade de escolher uma bipartição que resulte em um corte máximo é, portanto, pelo menos $1/2^{n-1}$, uma vez que o grafo pode possuir um único corte máximo. Logo,

$$\varepsilon \leq 1 - \frac{1}{2^{n-1}}.$$

□

Não é difícil se convencer de que grafos bipartidos completos possuem um único corte máximo, o que torna o limitante anterior justo[16] nesses casos. Assim, na ausência de suposições adicionais, não é possível reduzir o limitante superior do erro.

[16] Um limitante é *justo* quando vale com igualdade.

Analogamente ao Lema 1, segue que:

Teorema 5. *Dado um grafo $G = (V, E)$ com n vértices, o Algoritmo 2 resolve o problema do Corte Máximo com probabilidade de erro*

$$\varepsilon \leq \left(1 - \frac{1}{2^{n-1}}\right)^k.$$

Por uma argumentação semelhante à do Corolário 1, verifica-se que, para que $\varepsilon \to 0$ quando $m \to \infty$, é necessário que $k = \omega(2^n)$, resultando em um algoritmo exponencial.

Portanto, a princípio, o Algoritmo 2 não pode ser utilizado para resolver o problema do Corte Máximo com alta probabilidade em tempo polinomial.

Algoritmos do tipo Monte Carlo desempenham um papel importante no desenvolvimento de algoritmos eficientes para diversos problemas em grafos e otimização combinatória. Em muitos casos, obtemos algoritmos assintoticamente mais rápidos do que as soluções determinísticas, ou ainda

aproximações de alta qualidade para problemas computacionalmente difíceis. Técnicas como amostragem aleatória e arredondamento probabilístico de programas lineares são amplamente empregadas em problemas que envolvem cortes, emparelhamentos e fluxos em grafos [11,12,13,14].

2.1.2 Las Vegas

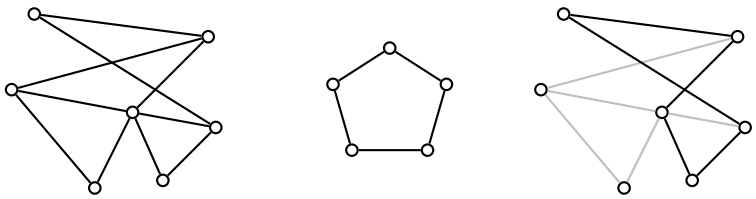
A seguir, apresentamos a definição formal de um algoritmo do tipo Las Vegas[17].

Definição 2 (*Algoritmo Las Vegas*)**.** *Um algoritmo aleatorizado A é dito Las Vegas se sempre produz uma resposta correta e se o seu tempo de execução em uma instância x do problema, denotado por $T_A(x)$, é uma variável aleatória.*

Dizemos que um algoritmo Las Vegas é eficiente se seu tempo de execução *esperado* é polinomial no tamanho da entrada.

Para exemplificar os conceitos apresentados, considere o *problema do* Isomorfismo de Subgrafos[18]:

> **Isomorfismo de subgrafos**
>
> Dado um grafo G e um grafo H, determine se H é isomorfo a algum subgrafo de G.



G $\quad$ $H \cong C_5$ $\quad$ $H \subset G$

Figura 2: Exemplo de isomorfismo de subgrafo, mostrando a identificação de um subgrafo H (em preto) dentro de um grafo G.

Nos restringiremos a uma versão específica do problema: encontrar um caminho simples de comprimento $k-1$ em G, que é computacionalmente difícil para k arbitrário, pois generaliza o problema do Caminho Hamiltoniano, que consiste em decidir se um grafo admite um caminho simples que visita todos os seus vértices, problema já demonstrado ser NP-completo [17].

Para isso, usaremos uma técnica introduzida por Alon, Yuster e Zwick [16], chamada de *codificação por cores*[19]. O método consiste em colorir aleatoriamente os vértices do grafo G usando k cores, a fim de encontrar um *caminho colorido* de comprimento $k-1$, isto é, um caminho em que

[17] A denominação *Las Vegas* foi introduzida por Babai em 1979 [15] para caracterizar algoritmos duais aos *Monte Carlo* – daí o nome, em alusão a outra cidade célebre por seus cassinos. Na formulação original, um algoritmo Las Vegas possui tempo de execução finito e falha aleatória certificável, definição equivalente à formulação moderna adotada neste trabalho: ao limitar o tempo de execução obtém-se uma versão com falha certificável; reciprocamente, ao reiniciar o algoritmo após falhas, recupera-se um algoritmo sem erro, com tempo de execução aleatório.

[18] O problema do Isomorfismo de Subgrafos é, em geral, $\mathcal{NP}$-completo. Ainda assim, admite algoritmos polinomiais para algumas classes de grafos [16].

[19] Para além de encontrar caminhos simples de comprimento reduzido em grafos, o método também pode ser usado para lidar com outras classes, como ciclos e árvores.

cada vértice recebe uma cor distinta. Note que, por ser colorido, o caminho encontrado é necessariamente simples.

Lema 2. *Seja $G = (V, E)$ um grafo e $c : V \to \{1, 2, \ldots, k\}$ uma k-coloração escolhida uniformemente ao acaso. Então, qualquer caminho de comprimento $k-1$ em G é um caminho colorido segundo c com probabilidade $k!/k^k > e^{-k}$.*

Demonstração. Há k^k colorações distintas possíveis para um caminho de comprimento $k-1$, das quais $k!$ são coloridas. Logo, a probabilidade de um caminho ser colorido segundo c é $k!/k^k$. □

Agora apresentaremos um algoritmo determinístico de programação dinâmica que, dada uma k-coloração dos vértices de $G = (V, E)$ e um vértice inicial $s \in V$, identifica todos os vértices $v \in V \setminus \{s\}$ para os quais existe um caminho colorido de comprimento $k-1$ que começa em s e termina em v. Além disso, para cada vértice alcançável, é possível reconstruir um caminho colorido de comprimento $k-1$ que o conecte a s.

Suponha que, no passo i, já tenhamos determinado, para cada vértice $v \in V \setminus \{s\}$, todos os conjuntos de cores que podem aparecer em algum caminho colorido de comprimento $i-1$ ligando s a v. Denotemos essa coleção por $\mathcal{C}_v^{i-1}$. Observe que ela contém, no máximo, $\binom{k}{i-1}$ conjuntos de cores.

Opcionalmente, para cada conjunto de cores $C \in \mathcal{C}_v^{i-1}$, podemos armazenar um caminho que o realize – o que nos permitirá reconstruir caminhos de comprimento $k-1$ ao final do algoritmo. Então, denotamos por $\mathcal{P}_v^{i-1}$ a coleção correspondente de caminhos e por $\mathcal{P}_v^{i-1}(C)$ o caminho associado ao conjunto de cores C.

Examinamos então cada conjunto $C \in \mathcal{C}_v^{i-1}$ e cada aresta $\{v, u\} \in E$. Se $c(u) \notin C$, adicionamos o conjunto

$$C' = C \cup \{c(u)\}$$

à coleção $\mathcal{C}_u^i$, correspondente aos caminhos coloridos de comprimento i que terminam em u. Opcionalmente, se $P = \mathcal{P}_v^{i-1}(C)$ é o caminho colorido associado a C, adicionamos o caminho estendido $P \cdot u$ à $\mathcal{P}_u^i$, como o caminho que realiza C'.

Ao final do algoritmo, verificamos se G contém um caminho colorido de comprimento $k-1$ segundo a coloração c observando se alguma coleção final $\mathcal{C}_v^{k-1}$ é não vazia para algum $v \in V \setminus \{s\}$. Para recuperar um dos caminhos que conecta s a tal vértice v, basta recuperar algum caminho $P \in \mathcal{P}_v^{k-1}$.

Algoritmo 3 (*Caminhos coloridos*)

A função CaminhosColoridos recebe um grafo G, um inteiro k, uma k-coloração dos vértices e um vértice s, e identifica todos os vértices

alcançáveis a partir de s por um caminho colorido de comprimento $k-1$. Opcionalmente, também permite recuperar algum desses caminhos para cada vértice alcançado.

Entrada: Grafo $G = (V, E)$ com n vértices e m arestas, uma coloração $c : V \to \{1, 2, \ldots, k\}$ e um vértice inicial $s \in V$.

Saída: O conjunto de vértices $v \in V \setminus \{s\}$ para os quais existe um caminho colorido de comprimento $k-1$ conectando s a v, e algum caminho correspondente para cada um deles.

```
CaminhosColoridos(G, k, c, s)
1.  Para cada vértice v ∈ V faça
2.      C_v^0 ← {c(v)}
3.      P_v^0 ← {v}

4.  Para i = 1 até k − 1 faça
5.      Para cada vértice v ∈ V faça
6.          C_v^i, P_v^i ← ∅

7.  Para i = 1 até k − 1 faça
8.      Para cada vértice v ∈ V faça
9.          Para cada conjunto C ∈ C_v^{i−1} faça
10.             P ← P_v^{i−1}(C)
11.                 Para cada aresta {v, u} ∈ E faça
12.                     Se c(u) ∉ C então
13.                         C' = C ∪ {c(u)}
14.                         C_u^i ← C_u^i ∪ {C'}
15.                         P_u^i(C') ← P · u

16. Retorne {v ∈ V : C_v^{k−1} ≠ ∅}, {P ∈ P_v^{k−1} : v ∈ V, P_v^{k−1} ≠ ∅}
```

Note que o tempo de execução do Algoritmo 3 é dado por $\sum_{i=0}^{k} i\binom{k}{i}m = \mathcal{O}(k2^k m)$.

Para encontrar um caminho de comprimento $k-1$ em G que inicie em qualquer vértice, basta adicionar um novo vértice s' em V, atribuir-lhe uma nova cor insignificante e conectá-lo a todos os vértices de V. Nesse cenário, buscamos um caminho colorido de comprimento k que começa em s'.

Não é difícil de ver que o algoritmo pode ser adaptado para encontrar ciclos de comprimento k: execute-o tomando cada vértice $v \in V$ como vértice inicial. Em seguida, para cada par $\{u, v\}$ conectado por um caminho de comprimento $k-1$, verifique se $\{u, v\} \in E$. Em caso afirmativo, obtém-se um ciclo de comprimento k. Observe que um resultado análogo ao Lema 2

aplica-se igualmente a ciclos. No trabalho original, Alon, Yuster e Zwick [16] apresentam um algoritmo mais sofisticado, capaz de encontrar *todos* os caminhos simples de comprimento $k-1$ em G.

Assim, combinando os dois resultados anteriores, obtemos um algoritmo aleatorizado Las Vegas que encontra um caminho de comprimento $k-1$ em G, caso exista: repetidamente colorimos o grafo aleatoriamente e tentamos encontrar um caminho colorido de comprimento $k-1$.

Algoritmo 4 (*Caminho simples*)

A função CaminhoSimples recebe um grafo G e um inteiro k, e encontra um caminho simples de comprimento $k-1$ em G, caso exista. A função auxiliar InteiroAleatório retorna um número inteiro escolhido uniformemente ao acaso no intervalo especificado.

Entrada: Grafo $G = (V, E)$, com n vértices e m arestas.

Saída: Caminho simples P de comprimento $k-1$, caso exista.

```
CaminhoSimples(G, k)
1.  H = (V ∪ {s}, E ∪ {{s, v} : v ∈ V})
2.  Repita
3.      c(s) ← 0
4.      Para cada vértice v ∈ V faça
5.          c(v) ← InteiroAleatório([1, k])
6.          S, P ← CaminhosColoridos(H, k + 1, c, s)
7.          Se P ≠ ∅ então
8.              Retorne P ∈ P
```

Teorema 6. *Dado um grafo $G = (V, E)$ com m arestas, o Algoritmo 4 retorna um caminho de comprimento $k-1$, caso exista, em tempo de execução esperado $\mathcal{O}(2^k k^{k+1} m/k!)$.*

Demonstração. Note que o algoritmo tem sucesso sempre que a coloração aleatória dos vértices torna algum caminho de comprimento $k-1$ colorido. Uma vez fixada a coloração, a verificação é totalmente determinística.

Seja K o número de colorações escolhidas ao acaso até que uma delas permita encontrar um caminho colorido desejado. Então, o tempo de execução $T(x)$ é dado por

$$T(x) = K \cdot \mathcal{O}(k 2^k m).$$

Pelo Lema 2, sabemos que cada coloração aleatória é bem-sucedida com probabilidade ao menos $k!/k^k$. Portanto, K é uma variável aleatória

geométrica com probabilidade de sucesso $p = k!/k^k$, e assim

$$\mathbb{E}[K] = \frac{1}{p} = \frac{k^k}{k!}.$$

Logo,

$$\mathbb{E}[T(x)] = \mathbb{E}[K] \cdot \mathcal{O}(k2^k m) = \mathcal{O}(2^k k^{k+1} m/k!).$$

□

Corolário 2. *Para $k = \mathcal{O}(\lg n)$, em que $\lg n = \log_2 n$, o Algoritmo 4 encontra um caminho de comprimento k em G em tempo de execução esperado polinomial – mais precisamente, $\mathcal{O}(n^{1+\lg e}\sqrt{\lg n}\, m)$.*

É importante ressaltar que o Algoritmo 4 não é eficiente quando usado para *decidir* se o grafo G contém um caminho de comprimento $k-1$, mesmo para k constante. Embora um certificado de SIM possa ser obtido em tempo esperado $\mathcal{O}(2^k k^{k+1} m/k!)$, o certificado de NÃO requer a verificação de todas as k^n colorações possíveis de G, implicando tempo exponencial no tamanho da entrada[20].

Um questionamento interessante surge no estudo de algoritmos do tipo Las Vegas. Considere um algoritmo Las Vegas que realiza decisões aleatórias incrementalmente, de forma que, à medida que a execução avança, seus passos dependem cada vez mais das escolhas aleatórias anteriores. Se observarmos que a execução está consumindo um tempo significativamente maior do que o esperado, poderíamos inferir que as escolhas aleatórias feitas até então foram desfavoráveis e, nesse caso, reiniciar o algoritmo com uma nova sequência de decisões aleatórias. Será que tal estratégia é capaz de reduzir o tempo de execução esperado?

Formalmente, uma *estratégia de reinicialização* para um algoritmo Las Vegas A é uma sequência

$$S = (t_1, t_2, \dots),$$

em que A é executado por t_1 passos, interrompido e reiniciado. Em seguida, é executado novamente por t_2 passos de forma independente, e assim sucessivamente. O processo termina assim que alguma dessas execuções é concluída com sucesso.

Denotamos por $S(A)$ o algoritmo Las Vegas resultante da aplicação de S em A. Naturalmente, $T_{S(A)}(x)$ denota a variável aleatória correspondente ao tempo total de execução do algoritmo A na instância x sob essa estratégia, e $\mathbb{E}[T_{S(A)}(x)]$ denota seu tempo de execução esperado.

Como exemplo ilustrativo, mostramos como uma estratégia particularmente simples já é suficiente para reduzir significativamente a probabilidade de cauda[21] do tempo de execução de um algoritmo Las Vegas arbitrário, *i.e.*, a

[20] Existe, entretanto, uma forma de derandomizar o algoritmo apresentado, isto é, eliminar o uso de aleatoriedade (veja Seção 2.3), com um acréscimo de apenas um fator $\lg n$ no tempo de execução [16]. Essa derandomização substitui a enumeração exponencial de colorações por uma família determinística de tamanho $2^{\mathcal{O}(k)} \lg n$, superando a limitação mencionada.

[21] A *probabilidade de cauda* de uma variável aleatória X é simplesmente a probabilidade de que X exceda ou fique abaixo de um limiar de interesse λ.

probabilidade de o algoritmo demorar muito tempo decai exponencialmente [18]. Observe que, na ausência de qualquer informação adicional sobre a distribuição do tempo de execução, a desigualdade de Markov fornece o melhor limitante geral disponível para a probabilidade de cauda:

$$\Pr(T_A(x) \geq t) \leq \frac{\mathbb{E}[T_A(x)]}{t}.$$

Teorema 7. *Dado um algoritmo Las Vegas A e uma instância x, seja $\mu = \mathbb{E}[T_A(x)]$. A estratégia de reinicialização*

$$S = (t_i = 2\mu)_{i \geq 1},$$

isto é, a estratégia que consiste em executar A por 2μ passos e repetir esse processo até obter sucesso, garante que a probabilidade de cauda do tempo de execução satisfaz

$$\Pr(T_{S(A)}(x) \geq t) \leq 2^{-\lfloor t/2\mu \rfloor}.$$

Demonstração. Pela desigualdade de Markov, vale que

$$\Pr(T_A(x) \geq 2\mu) \leq \frac{\mu}{2\mu} \leq \frac{1}{2}.$$

Assim, cada execução limitada a 2μ passos falha com probabilidade no máximo $1/2$. Como as execuções são independentes, a probabilidade de que $k \in \mathbb{N}$ execuções consecutivas falhem é no máximo 2^{-k}.

Logo,

$$\Pr(T_{S(A)}(x) \geq k2\mu) \leq \frac{1}{2^k},$$

ou, equivalentemente,

$$\Pr(T_{S(A)}(x) \geq t) \leq 2^{-\lfloor t/2\mu \rfloor},$$

como desejado. □

De forma mais geral, podemos definir o tempo de execução esperado ótimo de A, tomado sobre todas as estratégias de reinicialização S, como

$$\ell_A(x) = \inf_S \mathbb{E}[T_{S(A)}(x)].$$

Luby, Sinclair e Zuckerman [19] demonstraram a existência de uma *estratégia universal* S_{univ} que, para *todo* algoritmo Las Vegas A, vale que

$$\mathbb{E}[T_{S_{\text{univ}}(A)}(x)] = \mathcal{O}(\ell_A(x) \cdot \lg \ell_A(x)),$$

ou seja, a estratégia S_{univ} alcança um tempo de execução esperado ótimo

entre todas as estratégias, a menos de um fator logarítmico. A demonstração desse resultado foge ao escopo deste texto.

Diversos trabalhos posteriores aprofundaram o estudo de estratégias de reinicialização, abordando, por exemplo, estratégias ótimas no contexto de múltiplos processadores executando em paralelo [20], estratégias adaptativas que se atualizam com base no comportamento de execuções anteriores [21], bem como o estudo de *portfólios de algoritmos*, nos quais diferentes algoritmos randomizados são intercalados com o objetivo de reduzir o tempo de execução esperado total [22].

Algoritmos do tipo Las Vegas são particularmente relevantes para problemas computacionalmente difíceis, especialmente em grafos e otimização combinatória. Entre outras razões, sua importância decorre do fato de permitirem a exploração do espaço de soluções de forma aleatorizada, o que frequentemente possibilita contornar o crescimento exponencial do conjunto de soluções candidatas sem comprometer a validade da resposta obtida. Problemas que se enquadram nesse paradigma são conhecidos como *problemas de busca*, isto é, aqueles que envolvem encontrar uma testemunha que satisfaça determinadas condições – categoria que inclui muitos problemas $\mathcal{NP}$-completos.

Considere, por exemplo, o procedimento de *busca por backtracking aleatorizada*, no qual o algoritmo constrói incrementalmente uma solução parcial e, quando se torna evidente que o caminho de solução atual não é viável ou não conduz a uma solução ótima, o procedimento retrocede e faz escolhas aleatórias diferentes, explorando o espaço de soluções de maneira probabilística até encontrar uma solução satisfatória. As questões discutidas anteriormente – estratégias de reinicialização e redução da probabilidade de cauda do tempo de execução – estão diretamente ligadas a esse tipo de abordagem e surgem com frequência nesse contexto [23].

2.1.3 Conversão entre algoritmos Monte Carlo e Las Vegas

Uma propriedade interessante a respeito dos dois tipos de algoritmos abordados anteriormente é que, sob determinadas condições, é possível converter um algoritmo eficiente de um tipo em outro. Formalizamos esses resultados a seguir.

Teorema 8. *Seja A um algoritmo Monte Carlo eficiente com probabilidade de erro $\varepsilon \in (0,1)$ suficientemente pequena. Mais especificamente, $\varepsilon \leq 1 - 1/\operatorname{poly}(n)$, em que n denota o tamanho da entrada. Se a validade da resposta*

retornada por A pode ser verificada em tempo polinomial, então A pode ser transformado em um algoritmo Las Vegas B eficiente.

Demonstração. Definimos B como o algoritmo que executa A repetidamente até que a resposta retornada seja verificada como correta. Claramente B produz apenas respostas corretas, e seu tempo de execução $T_B(x)$ é uma variável aleatória que depende da aleatoriedade de A. Portanto, B é um algoritmo Las Vegas. Resta mostrar que seu tempo de execução esperado é polinomial.

Seja $p(n)$ um polinômio que limita o tempo de execução de uma chamada de A seguida da verificação. Seja K a variável aleatória que conta quantas repetições de A são necessárias até obtermos uma resposta correta. Então,

$$T_B(x) = K \cdot p(n).$$

Como cada execução tem probabilidade de sucesso $1 - \varepsilon$, a variável K é geométrica com probabilidade de sucesso $p = 1 - \varepsilon$. Logo,

$$\mathbb{E}[K] = \frac{1}{p} = \frac{1}{1-\varepsilon}.$$

Pela hipótese, existe um polinômio $q(n)$ tal que

$$\varepsilon \leq 1 - \frac{1}{q(n)}.$$

Logo,

$$1 - \varepsilon \geq \frac{1}{q(n)} \implies \mathbb{E}[K] = \frac{1}{1-\varepsilon} \leq q(n).$$

Portanto,

$$\mathbb{E}[T_B(x)] = \mathbb{E}[K] \cdot p(n) \leq q(n) \cdot p(n),$$

que é polinomial em n. Assim, B é um algoritmo Las Vegas eficiente. □

Para ilustrar o resultado, considere o Algoritmo 1. A validade da resposta pode ser verificada em $\mathcal{O}(1)$: basta comparar o tamanho do corte gerado com $\lfloor m/2 \rfloor$. Aplicando o Teorema 8, obtemos um algoritmo Las Vegas que sempre retorna um corte de tamanho pelo menos $\lfloor m/2 \rfloor$ e cujo tempo de execução esperado é $\mathcal{O}(m \cdot (n+m))$.

Teorema 9. *Seja A um algoritmo Las Vegas eficiente. Então, A pode ser transformado em um algoritmo Monte Carlo B eficiente, com probabilidade de erro arbitrariamente pequena.*

Demonstração. Escolhemos uma função $t = t(n)$ polinomial em n, o tamanho da entrada, e definimos B como o algoritmo que executa A por no máximo t passos: caso A não termine nesse limite, B é interrompido e retorna um

resultado de `Erro`.

Como B sempre executa no máximo t passos, e escolhemos t polinomial em n, segue que B é um algoritmo Monte Carlo eficiente. Resta então estimar sua probabilidade de erro $\varepsilon \in (0, 1)$.

Note que

$$\varepsilon = \Pr(T_A(x) > t).$$

Aplicando a desigualdade de Markov, temos

$$\varepsilon = \Pr(T_A(x) > t) \leq \Pr(T_A(x) \geq t) \leq \frac{\mathrm{E}[T_A(x)]}{t}.$$

Como A é um algoritmo Las Vegas eficiente, o tempo de execução esperado $\mathrm{E}[T_A(x)]$ é polinomial em n. Portanto, podemos escolher t arbitrariamente grande – desde que ainda polinomial – de modo que $t \gg \mathrm{E}[T_A(x)]$.

Com essa escolha, vale que

$$\varepsilon \leq \frac{\mathrm{E}[T_A(x)]}{t} \to 0,$$

quando $n \to \infty$, o que garante que B retorne a resposta correta com alta probabilidade e execute em tempo polinomial. □

Para ilustrar o resultado, considere o Algoritmo 4. Suponha que sejam testadas exatamente k^k colorações. Nesse caso, temos $t = \mathcal{O}(2^k k^{k+1} m)$. Aplicando o Teorema 9, obtemos um algoritmo Monte Carlo que encontra o caminho desejado em tempo de execução determinístico $\mathcal{O}(2^k k^{k+1} m)$, cuja probabilidade de erro, isto é, de não encontrar o caminho mesmo quando ele existe, é no máximo $1/k!$.

2.2 Modelo computacional e classes de complexidade

Agora formalizaremos o modelo computacional probabilístico, permitindo generalizar naturalmente as classes de complexidade convencionais e incorporar as ferramentas aleatorizadas apresentadas.

Seguindo a abordagem usual, adotamos o modelo de máquina de Turing, estendido ao contexto aleatorizado pela noção de *máquina de Turing probabilística*.

Definição 3 (*Máquina de Turing probabilística*)**.** *Uma máquina de Turing*

probabilística M é uma máquina de Turing com duas funções de transição, δ_0 e δ_1, tal que, a cada passo da computação, uma delas é selecionada uniformemente ao acaso, independentemente das escolhas anteriores.

O objetivo desta seção é investigar o poder computacional desse tipo de máquina de Turing.

Observe que, em uma execução de uma máquina de Turing probabilística com t passos, existem 2^t ramos possíveis de execução, cada um escolhido com probabilidade $1/2^t$. Assim, para uma entrada x, a probabilidade $\Pr[M(x) = 1]$ corresponde simplesmente à fração dos ramos de execução nos quais a máquina M termina retornando 1.

Primeiramente, a fim de demonstrar a generalidade do modelo, mostraremos – de forma análoga ao resultado que trata da simulação de máquinas de Turing com alfabetos maiores que o binário – que a suposição feita na introdução da Seção 2 é justificada. Em particular, provaremos que o acesso apenas a bits aleatórios é suficiente para simular, em tempo eficiente, uma distribuição uniforme sobre o conjunto $\{1, \ldots, n\}$. Assim, o modelo RAM estendido descrito anteriormente é equivalente, a menos de fatores polinomiais, ao modelo de máquina de Turing probabilística.

Teorema 10. *Dada uma fonte de bits aleatórios cujo custo de obtenção é unitário e um inteiro $n \in \mathbb{N}$, é possível simular eficientemente a amostragem de um inteiro aleatório uniformemente distribuído em $\{1, \ldots, n\}$. Mais precisamente, para todo $n \in \mathbb{N}$ e todo $\varepsilon > 0$, existe um algoritmo aleatorizado que executa em tempo $\mathcal{O}(\lg(1/\varepsilon) \cdot \lg n)$ e retorna um inteiro escolhido uniformemente ao acaso no intervalo desejado com probabilidade ao menos $1 - \varepsilon$.*

Demonstração. Considere o seguinte algoritmo A. Sorteamos $\lceil \lg n \rceil$ bits aleatórios, obtendo um inteiro r uniformemente distribuído no intervalo $\{0, \ldots, 2^{\lceil \lg n \rceil} - 1\}$. Se $r < n$, retornamos $r + 1$. De fato, como r é uniforme em $\{0, \ldots, 2^{\lceil \lg n \rceil} - 1\}$, a distribuição de $r + 1$ condicionada ao evento $r < n$ é uniforme em $\{1, \ldots, n\}$, como gostaríamos. Caso contrário, se $r \geq n$, dizemos que a execução falhou.

Para $n = 2$, o algoritmo é trivialmente correto. Suponha então $n \geq 3$. Como

$$2^{\lceil \lg n \rceil - 1} \leq n \leq 2^{\lceil \lg n \rceil} - 1,$$

o número de valores de r tais que $r \geq n$ é no máximo $n/2$.

Logo, a probabilidade de falha de uma única execução de A é estritamente menor que $1/2$.

Considere agora o procedimento A_k, que repete o algoritmo A independentemente por $k = \lceil \lg(1/\varepsilon) \rceil$ vezes, interrompendo assim que uma execução

é bem-sucedida. A probabilidade de que todas as k execuções falhem é então

$$\Pr(A_k \text{ falhar}) = \prod_{i=1}^{k} \Pr(i\text{-ésima execução de } A \text{ falhar}) < 2^{-k} \leq \varepsilon.$$

Evidentemente, o tempo total de execução de A_k é $\mathcal{O}(\lg(1/\varepsilon) \cdot \lg n)$, como queríamos. □

Definimos então a classe de complexidade $\mathcal{BPP}$, análoga à classe $\mathcal{P}$, com o objetivo de capturar a noção de computação probabilística eficiente com erro limitado.

Definição 4 ($\mathcal{BPP}$)**.** *A classe $\mathcal{BPP}$ (bounded-error probabilistic polynomial-time) consiste no conjunto das linguagens $L \subset \{0,1\}^*$ para as quais existe um algoritmo aleatorizado A que executa em tempo polinomial tal que, para toda entrada $x \in \{0,1\}^*$,*

- $x \in L \implies \Pr(A(x) \textit{ aceita}) \geq 2/3$;
- $x \notin L \implies \Pr(A(x) \textit{ aceita}) \leq 1/3$.

Como visto na Seção 2.1, a técnica de amplificação permite reduzir exponencialmente a probabilidade de erro de um algoritmo aleatorizado por meio de um número apenas polinomial de repetições independentes, desde que a probabilidade de erro inicial seja pequena o suficiente. Consequentemente, o limitante de erro $1/3$ adotado na definição de $\mathcal{BPP}$ torna-se arbitrário: poderíamos, por exemplo, trocá-lo por uma margem do tipo $1/2 - 1/\operatorname{poly}(n)$ sem impactar significativamente a definição da classe[22].

[22] A classe $\mathcal{PP}$ (*probabilistic polynomial-time*) corresponde ao caso limítrofe em que entradas pertencentes à L são aceitas com probabilidade estritamente maior que $1/2$, enquanto entradas fora de L são aceitas com probabilidade no máximo $1/2$. Essa classe não constitui um modelo razoável de computação probabilística eficiente, pois a amplificação pode exigir um número exponencial de repetições independentes, uma vez que não há garantia de que a probabilidade de acerto seja suficientemente maior que $1/2$.

Note que a classe tem profunda relação com algoritmos Monte Carlo com erro bilateral suficientemente pequeno.

Como as máquinas de Turing determinísticas constituem um caso particular de máquinas de Turing probabilísticas, obtemos imediatamente o resultado a seguir.

Proposição 1. $\mathcal{P} \subseteq \mathcal{BPP}$.

Utilizando técnicas triviais de derandomização, que serão apresentadas posteriormente (veja Teorema 12), obtemos também o resultado abaixo de forma direta.

Proposição 2. $\mathcal{BPP} \subseteq \mathcal{EXP}$.

Convém ainda introduzir outras classes de complexidade associadas a algoritmos Monte Carlo com erro unilateral suficientemente pequeno.

Definição 5 ($\mathcal{RP}$)**.** *A classe $\mathcal{RP}$ (randomized polynomial-time) consiste no conjunto das linguagens $L \subset \{0,1\}^*$ para as quais existe um algoritmo*

aleatorizado A que executa em tempo polinomial tal que, para toda entrada $x \in \{0,1\}^*$,

- $x \in L \implies \Pr(A(x)\ \textit{aceita}) \geq 1/2$;
- $x \notin L \implies \Pr(A(x)\ \textit{aceita}) = 0$.

Novamente, pelos mesmos argumentos discutidos anteriormente, a escolha do limitante $1/2$ para a probabilidade de erro é essencialmente arbitrária: qualquer constante $\delta \in (0,1)$, ou mesmo $1 - 1/\operatorname{poly}(n)$, levaria à mesma classe de linguagens por meio da técnica de amplificação.

A classe co-$\mathcal{RP}$ é definida de maneira análoga à classe $\mathcal{RP}$, consistindo das linguagens que admitem algoritmos aleatorizados de tempo polinomial que podem errar apenas quando $x \notin L$.

Note que o seguinte resultado decorre naturalmente das definições das classes.

Proposição 3. *$\mathcal{RP} \subseteq \mathcal{BPP}$ e co-$\mathcal{RP} \subseteq \mathcal{BPP}$.*

Observando que em $\mathcal{RP}$ (resp. co-$\mathcal{RP}$) toda computação que aceita (resp. rejeita) fornece um certificado de que a entrada pertence à linguagem (resp. não pertence), ou, de forma equivalente, que a sequência de bits aleatórios utilizada – de tamanho polinomial – pode ser usada como testemunha, obtemos imediatamente o seguinte resultado.

Proposição 4. *$\mathcal{RP} \subseteq \mathcal{NP}$ e co-$\mathcal{RP} \subseteq$ co-$\mathcal{NP}$.*

Por fim, introduzimos a classe de complexidade associada a algoritmos do tipo Las Vegas.

Definição 6 ($\mathcal{ZPP}$)**.** *A classe $\mathcal{ZPP}$ (zero-error probabilistic polynomial-time) consiste no conjunto das linguagens $L \subset \{0,1\}^*$ para as quais existe um algoritmo aleatorizado A que decide corretamente a linguagem L e cujo tempo de execução esperado é polinomial.*

Note que trivialmente também vale o resultado abaixo.

Proposição 5. *$\mathcal{P} \subseteq \mathcal{ZPP}$.*

Demonstramos então um resultado fundamental a respeito da classe $\mathcal{ZPP}$.

Teorema 11. *$\mathcal{ZPP} = \mathcal{RP} \cap$ co-$\mathcal{RP}$.*

Demonstração. Começamos mostrando que $\mathcal{ZPP} \subseteq \mathcal{RP} \cap$ co-$\mathcal{RP}$. Seja $L \in \mathcal{ZPP}$. Então, existe um algoritmo A que decide L corretamente e satisfaz

$$\mathbb{E}[T_A(x)] \leq p(n),$$

para algum polinômio p.

Pela desigualdade de Markov,

$$\Pr(T_A(x) \geq 2p(n)) \leq \frac{1}{2}.$$

Definimos então um algoritmo B que executa A por no máximo $2p(n)$ passos: se A termina dentro desse limite, B devolve sua resposta. Caso contrário, B rejeita. Assim,

- $x \in L \implies \Pr(B(x) \text{ aceita}) = \Pr(T_A(x) \leq 2p(n)) \geq 1/2$;
- $x \notin L \implies \Pr(B(x) \text{ aceita}) = 0$.

Logo, $L \in \mathcal{RP}$. Um argumento análogo, agora aceitando quando o limite de tempo é excedido, mostra que $L \in \text{co-}\mathcal{RP}$. Portanto,

$$\mathcal{ZPP} \subseteq \mathcal{RP} \cap \text{co-}\mathcal{RP}.$$

Por fim, mostraremos que $\mathcal{RP} \cap \text{co-}\mathcal{RP} \subseteq \mathcal{ZPP}$.

Seja $L \in \mathcal{RP} \cap \text{co-}\mathcal{RP}$. Então, existe um algoritmo A tal que, para toda entrada x:

- $x \in L \implies \Pr(A(x) \text{ aceita}) \geq 1/2$;
- $x \notin L \implies \Pr(A(x) \text{ aceita}) = 0$.

Além disso, existe um algoritmo B tal que, para toda entrada x:

- $x \in L \implies \Pr(B(x) \text{ aceita}) = 1$;
- $x \notin L \implies \Pr(B(x) \text{ aceita}) \leq 1/2$.

Construímos um algoritmo C que executa A e B de forma intercalada, repetindo suas execuções de maneira independente. O algoritmo C aceita a entrada x assim que A aceita e rejeita x assim que B rejeita. Note que C sempre decide corretamente a linguagem L:

- $x \in L \implies \Pr(C(x) \text{ rejeita } x) = \Pr(B(x) \text{ rejeita } x) = 0$;
- $x \notin L \implies \Pr(C(x) \text{ aceita } x) = \Pr(A(x) \text{ aceita } x) = 0$.

Resta analisar o tempo de execução esperado de C. Seja K_A (resp. K_B) a variável aleatória que denota o número de execuções de A (resp. B) até que uma decisão definitiva seja obtida. Como cada execução tem probabilidade de sucesso ao menos $1/2$, K_A e K_B são variáveis geométricas com

$$\mathbb{E}[K_A] \leq 2 \quad \text{e} \quad \mathbb{E}[K_B] \leq 2.$$

O tempo total de execução satisfaz

$$T_C(x) = K_A \cdot T_A(x) + K_B \cdot T_B(x).$$

Pela linearidade da esperança,

$$\mathbb{E}[T_C(x)] = \mathbb{E}[K_A] \cdot T_A(x) + \mathbb{E}[K_B] \cdot T_B(x) \leq 2(T_A(x) + T_B(x)).$$

Como $T_A(x)$ e $T_B(x)$ são polinomiais, segue que $\mathbb{E}[T_C(x)]$ é limitado por um polinômio.

Concluímos que $L \in \mathcal{ZPP}$, e portanto

$$\mathcal{RP} \cap \text{co-}\mathcal{RP} \subseteq \mathcal{ZPP}.$$

Os dois resultados completam a prova. □

Corolário 3. $\mathcal{ZPP} \subseteq \mathcal{BPP}$.

Dessa forma, ilustramos as seguintes relações entre as classes de complexidade discutidas:

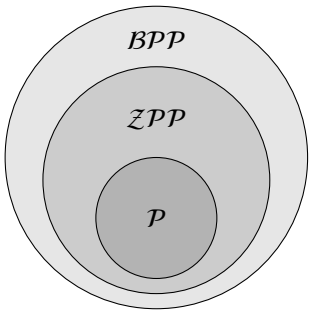


Figura 3: Relação entre algumas das classes de complexidade computacional probabilística. As inclusões podem não ser estritas.

Apesar desses resultados, muitas relações entre classes probabilísticas permanecem em aberto. Encerramos a seção enunciando uma das questões mais fundamentais da teoria da complexidade computacional probabilística.

Conjectura 1. $\mathcal{BPP} = \mathcal{P}$.

O interesse por essa questão é natural, pois demonstrar que $\mathcal{BPP} = \mathcal{P}$ levaria a uma conclusão extremamente forte: a aleatoriedade não confere poder computacional adicional, isto é, não torna tratáveis problemas que já não sejam tratáveis por máquinas determinísticas.

Embora a Conjectura 1 possa parecer contraintuitiva à primeira vista, na Seção 2.3 apresentaremos argumentos que sustentam a possibilidade de que ela seja verdadeira – de fato, muitos especialistas acreditam que esse seja o caso [5]. Essa crença baseia-se na expectativa de que todo algoritmo probabilístico eficiente possa ser *derandomizado*, isto é, convertido em um algoritmo determinístico equivalente, com apenas uma sobrecarga polino-

mial. Discutiremos esse tema em detalhe na Seção 2.3, onde buscaremos esclarecer por que essa conjectura é plausível.

2.3 Pseudoaleatoriedade e derandomização

Como vimos na introdução da Seção 2, algoritmos aleatorizados necessitam de bits aleatórios, independentes e imparciais para funcionarem corretamente. Naturalmente, nos perguntamos então como poderíamos obter tais bits, e, mais importante ainda, se é possível remover, total ou parcialmente, a necessidade de bits aleatórios nesses algoritmos.

Deve ficar claro que, ao longo desta seção, ao nos referirmos a um algoritmo aleatorizado, salvo menção em contrário, estaremos tratando de um algoritmo aleatorizado eficiente, com probabilidade de erro razoavelmente pequena, ou seja, o problema decidido pelo algoritmo pertencente à $\mathcal{BPP}$, pois constitui o foco principal de nosso estudo.

Apresentamos inicialmente uma forma trivial de *derandomizar* um algoritmo aleatorizado, isto é, construir um algoritmo equivalente que não utilize aleatoriedade – e que, portanto, seja determinístico – à custa de alguma sobrecarga no tempo de execução. Observe que, como o algoritmo obtido é determinístico, não há mais probabilidade associada à validade da resposta: o algoritmo estará sempre correto[23]. A técnica a seguir é conhecida como *enumeração*[24].

Teorema 12. *Seja A um algoritmo aleatorizado que utiliza d bits aleatórios e cuja probabilidade de erro é $\varepsilon < 1/2$.*

Então, existe um algoritmo determinístico B equivalente a A, que sempre retorna a resposta correta, com tempo de execução

$$T_B(x) = 2^d \cdot T_A(x),$$

para qualquer entrada x.

Demonstração. Observe que o número de sequências possíveis de bits aleatórios de A é finito e igual a 2^d.

Construímos então o algoritmo determinístico B da seguinte forma: para uma entrada x, B simula $A(x)$ com todas as 2^d sequências possíveis de bits aleatórios. Como a probabilidade de erro de A é $\varepsilon < 1/2$, por definição, a maioria dessas sequências produz a saída correta. Portanto, B retorna a saída que aparece com maior frequência. Dessa forma, B sempre fornece a

[23] Alternativamente, no caso de algoritmos de otimização, produzirá sempre uma solução que satisfaz as garantias de qualidade do algoritmo original, e assim por diante.

[24] Apresentamos a enumeração de algoritmos para problemas de decisão, mas argumentos semelhantes se aplicam a algoritmos para outros tipos de problemas, como problemas de otimização.

resposta correta.

Cada simulação de A leva tempo $T_A(x)$, e como existem 2^d sequências, a complexidade total de B é dada por

$$T_B(x) = 2^d \cdot T_A(x).$$

□

Note que, em geral, essa técnica é inviável, pois implica uma sobrecarga exponencial no tempo de execução. Entretanto, quando o algoritmo utiliza poucos bits aleatórios, essa abordagem simples passa a ser aplicável, resultando no corolário a seguir.

Corolário 4. *Seja A um algoritmo aleatorizado que utiliza $d = \mathcal{O}(\lg n)$ bits aleatórios em entradas de tamanho n, com probabilidade de erro $\varepsilon < 1/2$.*

Então, existe um algoritmo determinístico B equivalente a A, que sempre retorna a resposta correta, com tempo de execução

$$T_B(x) = 2^{\mathcal{O}(\lg n)} \cdot T_A(x) = \mathcal{O}(n)T_A(x),$$

para qualquer entrada x de tamanho n.

Dessa forma, a derandomização de qualquer algoritmo aleatorizado poderia ser alcançada demonstrando que o número de bits aleatórios necessários sempre pode ser reduzido a $\mathcal{O}(\lg n)$. Aqui, o conceito de *pseudoaleatoriedade* se torna particularmente relevante. Porém, antes de introduzi-lo formalmente, é necessário compreender melhor o que se entende por aleatoriedade computacional e como ela é obtida.

É essencial esclarecer que, na teoria da aleatoriedade computacional, a aleatoriedade – ou, mais precisamente, a *imprevisibilidade* – não é considerada uma propriedade intrínseca de um fenômeno, mas sim uma característica que depende do observador[25]. Portanto, se não somos capazes de distinguir um processo genuinamente aleatório de um processo determinístico complexo, ambos são considerados aleatórios do ponto de vista computacional [5].

Dessa forma, para obter bits verdadeiramente aleatórios – isto é, bits independentes, imparciais e distribuídos uniformemente – é necessário recorrer a fenômenos imprevisíveis, o que normalmente significa explorar atributos do ambiente que são praticamente impossíveis de modelar com precisão, como o clima, o decaimento radioativo, dados extraídos do próprio hardware do computador, entre outros[26].

Naturalmente, algumas fontes de aleatoriedade são mais robustas e confiáveis do que outras[27], e o processo de obtenção de bits verdadeiramente

[25] De fato, até onde sabemos, o universo pode ser completamente determinístico, de modo que a chamada aleatoriedade verdadeira sequer possa existir.

[26] Na prática, essas fontes de aleatoriedade frequentemente apresentam vieses e correlações indesejáveis, sendo denominadas *fontes fracas de aleatoriedade*. Para lidar com esse problema, utilizam-se *extratores*, algoritmos determinísticos capazes de extrair bits quase uniformes – mais precisamente, estatisticamente próximos da distribuição uniforme – a partir de fontes enviesadas e correlacionadas, produzindo bits adequados para aplicações práticas [5].

[27] Esse aspecto é particularmente relevante em criptografia, pois a "qualidade" da aleatoriedade está diretamente relacionada à segurança do processo criptográfico [5].

aleatórios torna-se, em geral, custoso. Surge, então, novamente, o questionamento de se é possível economizar a quantidade necessária de bits verdadeiramente aleatórios para que um algoritmo aleatorizado funcione corretamente. Observe que, se essa economia for suficientemente grande, torna-se possível até mesmo derandomizar completamente o algoritmo, conforme mencionado anteriormente como consequência do Corolário 4.

É aqui que a observação anterior se torna fundamental: se a aleatoriedade de um objeto depende do observador, talvez seja possível "enganar" um algoritmo fornecendo-lhe bits que não são verdadeiramente aleatórios, mas que se assemelham o suficiente à distribuição uniforme de forma que o algoritmo não consiga distingui-los em tempo computacionalmente viável. Nesse caso, o algoritmo continuaria a funcionar com uma probabilidade de sucesso semelhante à original, mesmo sem utilizar aleatoriedade perfeita. Formalmente, essa intuição significa exigir que nenhuma computação eficiente consiga distinguir entre bits verdadeiramente aleatórios e bits produzidos por um determinado processo determinístico.

Isso motiva o conceito de *distribuições indistinguíveis*.

Definição 7 (*Distribuições ε-indistinguíveis*)**.** *Sejam $\{X_n\}_{n\in\mathbb{N}}$ e $\{Y_n\}_{n\in\mathbb{N}}$ duas sequências de distribuições, com X_n, Y_n distribuídas sobre $\{0,1\}^n$.*

Dizemos que $\{X_n\}_{n\in\mathbb{N}}$ e $\{Y_n\}_{n\in\mathbb{N}}$ são computacionalmente ε-indistinguíveis se, para todo algoritmo aleatorizado polinomial A e para n suficientemente grande, vale que

$$|\Pr[A(X_n) = 1] - \Pr[A(Y_n) = 1]| < \varepsilon(n),$$

em que $\varepsilon(n)$ é uma função negligenciável em n (i.e., decresce mais rápido que o inverso de qualquer polinômio em n) e a probabilidade é tomada sobre X_n, Y_n e as decisões aleatórias de A.

Em outras palavras, nenhuma computação eficiente – nem mesmo aleatorizada – consegue distinguir X_n de Y_n com vantagem não negligenciável.

Portanto, qualquer algoritmo aleatório eficiente mantém seu desempenho probabilístico mesmo quando suas escolhas internas, originalmente aleatórias, são substituídas por uma sequência pseudoaleatória ε-indistinguível de U_d, a distribuição uniforme sobre $\{0,1\}^d$.

Nosso objetivo, então, é construir de forma determinística uma distribuição pseudoaleatória D sobre d bits que seja ε-indistinguível de U_d. Para minimizar o uso de bits verdadeiramente aleatórios, idealmente gostaríamos que D pudesse ser gerada sem nenhum bit aleatório, permitindo uma simulação determinística direta de qualquer algoritmo aleatorizado. Contudo, isso é claramente impossível, pois a aleatoriedade não pode ser gerada de forma

puramente determinística. Portanto, em vez disso, buscamos construir D a partir de um número significativamente menor de bits aleatórios, digamos $k \ll d$, por meio de um *gerador pseudoaleatório*.

Definição 8 (*Gerador pseudoaleatório*)**.** *Uma função determinística*

$$G : \{0,1\}^k \to \{0,1\}^{\ell(k)}$$

é chamada de ε-gerador pseudoaleatório se $G(U_k)$ é ε-indistinguível de $U_{\ell(k)}$.

Em outras palavras, um gerador pseudoaleatório é capaz de "esticar" uma sequência curta de k bits aleatórios em uma sequência maior de $\ell(k)$ bits que, do ponto de vista de qualquer algoritmo eficiente, é praticamente indistinguível de uma sequência verdadeiramente aleatória de $\ell(k)$ bits. Note que é trivial construir um gerador pseudoaleatório para $\ell(k) \leq k$.

Observe que, se nosso objetivo final é uma simulação totalmente determinística de um algoritmo aleatorizado, não podemos, de fato, utilizar k bits aleatórios para gerar D. No entanto, explorando ideias semelhantes às do Teorema 12, podemos contornar essa limitação por meio de uma abordagem por enumeração.

Dado um algoritmo aleatorizado A que utiliza d bits aleatórios, um gerador pseudoaleatório $G : \{0,1\}^k \to \{0,1\}^{\ell(k)}$ com $\ell(k) = d$, e uma entrada x, enumeramos todas as sequências possíveis $s \in \{0,1\}^k$. Para cada uma delas, computamos $r = G(s)$ e executamos o algoritmo A em x utilizando r como sua fonte de aleatoriedade. Após obter os 2^k resultados correspondentes, tomamos o resultado majoritário e o declaramos como resposta para a entrada x. Note que esse procedimento determinístico fornece sempre a resposta correta para toda entrada x.

Se k for suficientemente pequeno – por exemplo, $k = \mathcal{O}(\lg n)$, de modo que $2^k = \mathcal{O}(n)$, em que n denota o tamanho da entrada – e se o gerador G puder ser computado eficientemente, a sobrecarga total será apenas polinomial. Assim, obtemos uma simulação determinística em tempo polinomial do algoritmo aleatorizado, como desejado.

Perceba como a viabilidade do processo descrito implica diretamente a Conjectura 1. Infelizmente, ainda não conseguimos demonstrar a existência de geradores pseudoaleatórios suficientemente bons, isto é, que satisfaçam todas as condições acima, mas, sem entrar em detalhes, há fortes evidências e uma ampla crença na comunidade de que tais geradores existem [5]. É nesse contexto que se fundamenta a Conjectura 1.

Existem outras técnicas para derandomizar algoritmos aleatorizados, embora todas apresentem limitações intrínsecas. A seguir, apresentamos algumas dessas abordagens.

Uma maneira não construtiva de derandomizar algoritmos pode ser obtida por meio do método probabilístico:

Teorema 13. *Seja A um algoritmo aleatorizado que utiliza $d = d(n)$ bits aleatórios e cuja probabilidade de erro é $\varepsilon < 2^{-n}$, em que n denota o tamanho da entrada. Então, para todo n, existe uma sequência de bits $r_n^* \in \{0,1\}^d$ tal que $A_{r_n^*}(x)$ retorna a resposta correta para toda entrada x de tamanho n.*

Demonstração. Para simplificar o argumento, sem perda de generalidade, representamos uma entrada x de tamanho n em binário, *i.e.*, $x \in \{0,1\}^n$. Denotamos por $A_r(x)$ o resultado da execução de A na entrada x quando $r \in \{0,1\}^d$ é usada como sua sequência de aleatoriedade.

Considere então uma sequência de bits $R \in \{0,1\}^d$ escolhida uniformemente ao acaso. Segue que

$$\Pr\left(\exists x \in \{0,1\}^n : \ A_R(x) \text{ incorreto}\right) \leq \sum_{x \in \{0,1\}^n} \Pr\left(A_R(x) \text{ incorreto}\right)$$

$$\Pr\left(\exists x \in \{0,1\}^n : \ A_R(x) \text{ incorreto}\right) < 2^n \cdot 2^{-n} = 1$$

Como a probabilidade de falha é estritamente menor que 1, existe pelo menos uma sequência $r_n^* \in \{0,1\}^d$ que funciona corretamente para todas as entradas x de tamanho n. □

Note que, uma vez conhecida a sequência r_n^*, computar A utilizando r_n^* como sequência aleatória pode ser feito em tempo polinomial. No entanto, como a demonstração é não construtiva, não é imediato saber como obter r_n^* de forma eficiente.

Vale ressaltar que, por amplificação, a probabilidade de erro de qualquer algoritmo aleatorizado pode ser reduzida, em tempo polinomial, para um valor menor que 2^{-n}. Dessa forma, o Teorema 13 é sempre aplicável aos algoritmos de interesse.

3 Análise probabilística de algoritmos

Se a aleatoriedade provavelmente não é capaz de aumentar o poder computacional de algoritmos, então precisamos usá-la de outra forma para tratar problemas difíceis.

Outra estratégia – um pouco mais radical – para introduzir aleatoriedade é substituir a análise de pior caso por uma *análise probabilística de algoritmos*[28]. Em vez de exigir eficiência em todas as entradas – algo irrealista para problemas difíceis – buscamos algoritmos que sejam "tipicamente" eficientes. Há boas razões para considerar essa mudança de perspectiva. A análise de algoritmos busca, afinal, avaliar o desempenho dessas ferramentas na resolução de problemas, fornecendo um modelo no qual algoritmos distintos podem ser comparados. Nesse sentido, a análise de pior caso fornece, de fato, a melhor garantia possível de eficiência. No entanto, quando resulta em uma estimativa negativa, tal resultado pode ser excessivamente pessimista para aplicações práticas. Note que, nesse paradigma, uma única instância problemática já é suficiente para que um algoritmo seja classificado como exponencial.

[28] Observe que os resultados sobre a dificuldade e a intratabilidade de problemas computacionais geralmente pressupõem uma análise de pior caso. Dizemos que provavelmente não existem algoritmos polinomiais para resolver problemas $\mathcal{NP}$-difíceis no sentido de que não há algoritmo capaz de resolvê-los em tempo polinomial para todas as instâncias, isto é, adotando explicitamente o critério de pior caso.

Mas e se conseguíssemos projetar um algoritmo para um problema difícil que, em média, executasse em tempo polinomial? Ou, ainda, se considerássemos as instâncias de pior caso, mas as perturbássemos levemente de forma aleatória, eliminando estruturas patológicas raras[29]? Talvez isso bastasse para assegurar que tais algoritmos são suficientemente práticos, justificando seu uso em aplicações reais em detrimento de outras alternativas.

[29] Embora, à primeira vista, esse possa parecer um modelo arbitrário de instâncias, sua utilidade prática se tornará clara na Seção 3.3.

Embora o termo análise probabilística de algoritmos seja frequentemente usado para se referir apenas à análise de caso médio, neste trabalho adotamos um sentido mais amplo, incluindo qualquer tipo de análise de algoritmos que envolva aleatoriedade nas instâncias.

Nesta seção, investigaremos como esse paradigma pode oferecer resultados menos pessimistas sobre a tratabilidade desses problemas computacionalmente difíceis. Naturalmente, tal paradigma de análise pressupõe algum tipo de aleatoriedade na geração das instâncias sobre as quais o desempenho do algoritmo será avaliado. Por simplicidade, analisaremos apenas algoritmos determinísticos, de modo que a única fonte de aleatoriedade provenha da escolha das instâncias. Contudo, as definições e os resultados aqui apresentados estendem-se naturalmente também a algoritmos aleatorizados.

No contexto de grafos, instâncias médias correspondem a grafos gerados aleatoriamente. Na Seção 3.1 introduziremos o modelo de grafo aleatório frequentemente estudado, bem como algumas terminologias e resultados

básicos.

Posteriormente, nas Seções 3.2 e 3.3, apresentaremos dois paradigmas distintos de análise probabilística de algoritmos: a análise de caso médio e a análise suavizada, respectivamente. No primeiro, a complexidade de um algoritmo é definida como o tempo médio de execução, assumindo que as instâncias são amostradas segundo uma dada distribuição de probabilidade. No segundo, considera-se o pior caso entre os tempos médios resultantes de pequenas perturbações aleatórias das instâncias.

3.1 Grafos aleatórios

Grafos definidos segundo modelos probabilísticos são denominados *grafos aleatórios*[30]. Em modelos clássicos de grafos aleatórios, dado um número fixo de vértices n, construímos um grafo aleatório G selecionando aleatoriamente quais arestas estarão presentes no grafo. O modo como sorteamos essas arestas define diferentes construções de grafos aleatórios. Dentre os diversos modelos propostos e estudados ao longo do tempo, um se destaca por sua relevância: o modelo de Erdős–Rényi, denotado por $G(n, p)$ [24].

[30] A teoria dos grafos aleatórios foi fundamentada por Erdős e Rényi no final da década de 1950, após Erdős perceber, a partir de resultados obtidos alguns anos antes, que métodos probabilísticos eram frequentemente úteis na resolução de problemas extremais em teoria dos grafos [24].

Nesse modelo, define-se um espaço de probabilidade $G(n, p)$ sobre todos os grafos com n vértices. Em um grafo aleatório $G \sim G(n, p)$, cada aresta é incluída de forma independente com probabilidade $0 < p < 1$. Por conveniência, $G(n, p)$ é definido sobre grafos rotulados, isto é, os n vértices recebem rótulos em $\{1, 2, \dots, n\}$, e grafos com rotulações distintas são tratados como objetos distintos, ainda que sejam isomorfos, pois isso simplifica significativamente os cálculos realizados[31].

[31] Deve ficar claro que essa distinção é, em grande medida, irrelevante: em geral, as propriedades estudadas são invariantes por isomorfismo, além de que grafos rotulados constituem um modelo natural para entradas de algoritmos. Do ponto de vista assintótico, a rotulação não introduz novos fenômenos e apenas traz uma complexidade técnica residual. Uma discussão mais detalhada a respeito desse tema pode ser encontrada em [24].

Em particular, a probabilidade de um dado grafo H ser sorteado nesse modelo é dada por

$$\Pr(G \cong H) = p^{e(H)}(1-p)^{\binom{n}{2}-e(H)},$$

em que $e(H)$ representa o número de arestas de H.

Observe que, para $p = 1/2$, todos os grafos com n vértices são equiprováveis, de modo que $G(n, 1/2)$ induz a distribuição uniforme sobre o conjunto de grafos rotulados.

Em geral, p é uma função de n, isto é, $p = p(n)$. Consideramos, então, uma sequência de espaços de probabilidade, um para cada $n \in \mathbb{N}$, formada por todos os grafos com exatamente n vértices. Nosso interesse recai sobre as propriedades assintóticas desses espaços quando $n \to \infty$.

Dizemos que um grafo aleatório *possui uma propriedade $\mathcal{P}$ com alta probabilidade* se a probabilidade de que um grafo com n vértices tenha $\mathcal{P}$ tende a 1 à medida que $n \to \infty$. Equivalentemente, afirmamos que *quase todo grafo satisfaz a propriedade $\mathcal{P}$*.

Como um exercício ilustrativo, analisaremos para quais valores de p um grafo aleatório contém ou não um triângulo. A demonstração será apresentada em duas etapas.

Teorema 14. *Seja $G \sim G(n,p)$. Se $p \ll 1/n$, então*

$$\Pr(K_3 \subset G) \to 0,$$

quando $n \to \infty$.

Demonstração. Seja X a variável aleatória que conta o número de triângulos em G. Claramente,

$$\Pr(K_3 \subset G) = \Pr(X \geq 1).$$

Seja $\mathcal{T}$ o conjunto de todos os triângulos contidos em K_n. Podemos escrever

$$X = \sum_{T \in \mathcal{T}} \mathbb{1}_T,$$

em que $\mathbb{1}_T$ é a variável indicadora do triângulo T, isto é, $\mathbb{1}_T = 1$ se $T \subset G$ e $\mathbb{1}_T = 0$ caso contrário.

Pela linearidade da esperança e usando a desigualdade $\binom{n}{k} \leq n^k$, temos que

$$\mathbb{E}[X] = \sum_{T \in \mathcal{T}} \mathbb{E}[\mathbb{1}_T] = \binom{n}{3} p^3 \leq (np)^3 \to 0,$$

quando $n \to \infty$, pois $p \ll 1/n$.

Aplicando a desigualdade de Markov, concluímos que

$$\Pr(K_3 \subset G) = \Pr(X \geq 1) \leq \mathbb{E}[X] \to 0,$$

como afirmado. □

Teorema 15. *Seja $G \sim G(n,p)$. Se $p \gg 1/n$, então*

$$\Pr(K_3 \subset G) \to 1,$$

quando $n \to \infty$.

Demonstração. Seja X a variável aleatória que conta o número de triângulos em G. Note que

$$\Pr(K_3 \subset G) = \Pr(X \geq 1) = 1 - \Pr(X = 0).$$

Assim, basta mostrar que $\Pr(X = 0) \to 0$ quando $n \to \infty$. Para isso, usaremos a desigualdade de Chebyshev[32], o que exige estimar $\mathbb{E}[X]$ e $\mathrm{Var}(X)$.

[32] Para uma variável aleatória X com variância finita e diferente de 0, a desigualdade de Chebyshev determina que $Pr(|X - \mathbb{E}[X]| \geq \lambda) \leq \mathrm{Var}(X)/\lambda^2$.

Como antes,

$$X = \sum_{T \in \mathcal{T}} \mathbb{1}_T,$$

em que $\mathcal{T}$ é o conjunto de todos os triângulos em K_n e $\mathbb{1}_T$ é a variável indicadora do evento $\{T \subset G\}$.

Pela linearidade da esperança e usando a desigualdade $\binom{n}{k} \geq (n/k)^k$, temos

$$\mathbb{E}[X] = \sum_{T \in \mathcal{T}} \mathbb{E}[\mathbb{1}_T] = \binom{n}{3} p^3 \geq \left(\frac{np}{3}\right)^3.$$

Em particular, como $p \gg 1/n$, segue que $\mathbb{E}[X] \to \infty$.

Para limitar a variância de X, calculamos $\mathbb{E}[X^2]$. Pela linearidade da esperança,

$$\mathbb{E}[X^2] = \sum_{S \in \mathcal{T}} \sum_{T \in \mathcal{T}} \mathbb{E}[\mathbb{1}_S \mathbb{1}_T].$$

Note que

$$\mathbb{1}_S \mathbb{1}_T = 1 \iff S \cup T \subset G.$$

Então, particionando a soma dupla de acordo com a interseção entre os triângulos S e T, temos

$$\mathbb{E}[X^2] = \sum_{\substack{S,T \in \mathcal{T} \\ |S \cap T| = 3}} \mathbb{E}[\mathbb{1}_S \mathbb{1}_T] + \sum_{\substack{S,T \in \mathcal{T} \\ |S \cap T| = 1}} \mathbb{E}[\mathbb{1}_S \mathbb{1}_T] + \sum_{\substack{S,T \in \mathcal{T} \\ |S \cap T| = 0}} \mathbb{E}[\mathbb{1}_S \mathbb{1}_T].$$

Observe que não há caso $|S \cap T| = 2$, pois dois triângulos que compartilham duas arestas são idênticos, e portanto $|S \cap T| = 3$.

Analisamos então os seguintes casos:

- Se $|S \cap T| = 3$, então $S = T$ e, neste caso, $\mathbb{1}_S \mathbb{1}_T = \mathbb{1}_T^2 = \mathbb{1}_T$ e $\mathbb{E}[\mathbb{1}_S \mathbb{1}_T] = \mathbb{E}[\mathbb{1}_T]$;
- Se $|S \cap T| = 1$, então a ocorrência simultânea exige a presença de cinco arestas distintas, logo $\mathbb{E}[\mathbb{1}_S \mathbb{1}_T] = p^5$;
- Se $|S \cap T| = 0$, então $\mathbb{1}_T$ e $\mathbb{1}_S$ são variáveis independentes e vale que $\mathbb{E}[\mathbb{1}_S \mathbb{1}_T] = \mathbb{E}[\mathbb{1}_S]\mathbb{E}[\mathbb{1}_T]$.

Assim,

$$\mathbb{E}[X^2] = \sum_{T \in \mathcal{T}} \mathbb{E}[\mathbb{1}_T] + \sum_{\substack{S,T \in \mathcal{T} \\ |S \cap T| = 1}} p^5 + \sum_{\substack{S,T \in \mathcal{T} \\ |S \cap T| = 0}} \mathbb{E}[\mathbb{1}_S]\mathbb{E}[\mathbb{1}_T].$$

Como o número de pares de triângulos que compartilham exatamente uma aresta é $O(n^4)$, e o último somatório é limitado superiormente por

$$\sum_{\substack{S,T\in\mathcal{T}\\|S\cap T|=0}} \mathbb{E}[\mathbb{1}_S]\mathbb{E}[\mathbb{1}_T] \leq \sum_{S,T\in\mathcal{T}} \mathbb{E}[\mathbb{1}_S]\mathbb{E}[\mathbb{1}_T] = \mathbb{E}[X]^2,$$

concluímos que

$$\mathbb{E}[X^2] \leq \mathbb{E}[X] + n^4p^5 + \mathbb{E}[X]^2.$$

Logo,

$$\mathrm{Var}(X) = \mathbb{E}[X^2] - \mathbb{E}[X]^2 \leq \mathbb{E}[X] + n^4p^5 \ll \mathbb{E}[X]^2.$$

Finalmente, aplicando a desigualdade de Chebyshev, temos

$$\Pr(X = 0) \leq \Pr(\mathbb{E}[X] - X \geq \lambda) \leq \Pr(|X - \mathbb{E}[X]| \geq \lambda) \leq \frac{\mathrm{Var}(X)}{\lambda^2}.$$

Tomando $\lambda = \mathbb{E}[X]/2$, temos

$$\Pr(X = 0) \leq \frac{4\,\mathrm{Var}(X)}{\mathbb{E}[X]^2} \to 0,$$

quando $n \to \infty$, como gostaríamos. □

A função $1/n$ é chamada de *limiar* para a presença de triângulos em $G(n, p)$. Isso significa que, se $p \gg 1/n$, então com alta probabilidade um grafo $G \sim G(n, p)$ contém um triângulo, enquanto que, se $p \ll 1/n$, então com alta probabilidade o grafo não contém nenhum triângulo.

Diversas outras propriedades de grafos apresentam limiares, como conexidade e a presença de qualquer subgrafo fixo. Em particular, Bollobás e Thomason [25] demonstraram que toda propriedade crescente não trivial em grafos, isto é, toda propriedade que é preservada pela adição de arestas, apresenta um limiar no modelo $G(n, p)$.

É evidente a importância do estudo de grafos aleatórios em áreas como combinatória e teoria dos grafos, pois permite demonstrar a existência de grafos com propriedades específicas sem a necessidade construí-los explicitamente. Na ciência da computação, grafos aleatórios são muito relevantes na análise de algoritmos, no estudo da complexidade computacional e na investigação de estruturas em redes e problemas de otimização.

3.2 Análise de caso médio

Uma alternativa frequentemente empregada na análise probabilística de algoritmos é a *análise de caso médio*, na qual assumimos uma distribuição de probabilidade sobre o espaço de possíveis entradas e estimamos o desempenho esperado do algoritmo em relação a essa distribuição.

Definição 9 (*Complexidade de caso médio*)**.** *Seja A um algoritmo para um dado problema Π, e seja Ω_n o conjunto de todas as instâncias de Π de tamanho n. Dada uma distribuição de probabilidade $\mathcal{D}$ sobre Ω_n, a complexidade de caso médio de A é definida por*

$$\texttt{Médio}_A^{\mathcal{D}}(n) := \mathop{\mathbb{E}}_{x \sim D}[T_A(x)],$$

em que $x \in \Omega_n$.

Note que a análise é altamente dependente da distribuição de probabilidade escolhida para o conjunto de entradas possíveis. Quando não dispomos de informações adicionais sobre uma distribuição adequada – o que é muito frequente –, é natural assumirmos uma distribuição uniforme, *i.e.*, uma distribuição na qual toda entrada é equiprovável. No contexto de grafos, por exemplo, isso equivale a considerar a entrada como um grafo aleatório $G \sim G(n, 1/2)$, como discutido na Seção 3.1. Embora essa abordagem possa ser considerada irrealista, ela provavelmente não é mais artificial do que os exemplos extremos usados para demonstrar a ineficiência de algoritmos e, em certo sentido, talvez possa oferecer uma análise mais próxima da realidade.

Por outro lado, poderíamos também demonstrar que um algoritmo quase sempre apresenta um desempenho ruim em instâncias aleatórias, resultando numa crítica mais severa de seu desempenho do que os exemplos patológicos considerados na análise de pior caso.

Na Seção 3.2.1, apresentaremos um algoritmo para resolver um problema $\mathcal{NP}$-completo em tempo esperado polinomial, assumindo uma distribuição uniforme sobre as instâncias.

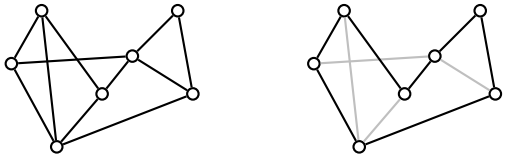
Em seguida, na Seção 3.2.2, exploraremos a relação entre algoritmos aleatorizados e a análise de caso médio por meio da teoria dos jogos, o que resulta no Lema Minimax de Yao.

Recomendamos ao leitor as excelentes revisões introdutórias [26,27] sobre análise de caso médio em algoritmos para grafos e sobre algoritmos em grafos aleatórios.

3.2.1 Ciclo hamiltoniano em tempo esperado polinomial

Inicialmente, definimos o problema de interesse, o *problema do Ciclo Hamiltoniano*[33]:

> **Ciclo hamiltoniano**
>
> Dado um grafo G, encontre um ciclo C que visita cada vértice de G exatamente uma vez.



G $\quad$ $C \subset G$

Figura 4: Exemplo de ciclo hamiltoniano C (em preto) em um grafo G, isto é, um ciclo que visita cada vértice exatamente uma vez.

[33] O problema do Ciclo Hamiltoniano está entre os 21 problemas clássicos que Karp caracterizou como $\mathcal{NP}$-completos, em uma das primeiras demonstrações de que diversos problemas computacionais recorrentes são intrinsecamente difíceis [17].

Apresentaremos dois algoritmos para resolver esse problema: um algoritmo exato, com tempo de execução exponencial no pior caso, e um algoritmo polinomial e determinístico, que pode falhar em encontrar um ciclo hamiltoniano mesmo quando ele existe. Ao combinar ambos, recorrendo ao algoritmo exponencial apenas quando o algoritmo polinomial falhar, obteremos um algoritmo cujo tempo de execução esperado é polinomial sob a distribuição uniforme de grafos.

Para resolver o problema de forma exata, utilizaremos um algoritmo clássico de programação dinâmica, desenvolvido independentemente por Bellman [28] e por Held e Karp [29].

Seja G o grafo de entrada. Sem perda de generalidade, fixamos um vértice inicial $r \in V(G)$ – como buscamos um ciclo, a escolha de r é irrelevante. Dado um subconjunto $S \subseteq V(G) \setminus \{r\}$ e um vértice $v \in S$, definimos a função

$$\mathrm{DP}(S, v) = \begin{cases} 1, \text{ se existe um caminho simples } P \text{ tal que} \\ \quad P \text{ começa em } r, \text{ termina em } v, \text{ e } V(P) = S \cup \{r\}, \\ 0, \text{ caso contrário.} \end{cases}$$

Observe que o valor de $\mathrm{DP}(S, v)$ é trivial para conjuntos S pequenos e que a função pode ser determinada recursivamente. Mais precisamente,

$$\mathrm{DP}(\{v\}, v) = 1 \iff v \in N(r),$$

e

$$\mathrm{DP}(S, v) = 1 \iff \exists u \in S \cap N(v) \text{ tal que } \mathrm{DP}(S \setminus \{v\}, u) = 1,$$

em que $N(v)$ denota a vizinhança de um vértice v em G.

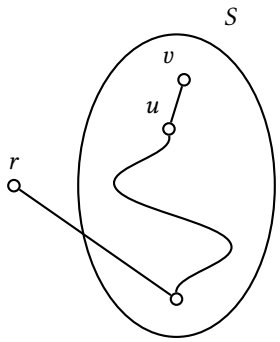


Figura 5: Representação de um caminho considerado ao calcular $\mathrm{DP}(S, v)$, com r como vértice inicial e $u, v \in S$ como vértices finais.

Claramente, o grafo G contém um ciclo hamiltoniano se, e somente se, existe um vértice $v \in N(r)$ tal que $\mathrm{DP}(V(G) \setminus \{r\}, v) = 1$. Nesse caso, o ciclo pode ser recuperado armazenando, para cada estado (S, v) com $\mathrm{DP}(S, v) = 1$, um vértice predecessor $\pi(S, v)$ que justifique a recorrência. Ao final, o ciclo é reconstruído recursivamente. Com isso, obtemos o algoritmo descrito a seguir.

Algoritmo 5 (*BellmanHeldKarp*)

A função BellmanHeldKarp recebe como entrada um grafo G e retorna um ciclo hamiltoniano em G, se existir, e $\emptyset$, caso contrário.

Entrada: Grafo $G = (V, E)$ com n vértices.

Saída: Um ciclo hamiltoniano C em G ou $\emptyset$, caso tal ciclo não exista.

```
BellmanHeldKarp(G)
1.  Escolha um vértice r ∈ V(G)

2.  Para cada vértice v ∈ V(G) \ {r} faça
3.      Se v ∈ N(r) então
4.          DP({v}, v) ← 1
5.          π({v}, v) ← r
6.      Senão
7.          DP({v}, v) ← 0

8.  Para k = 2 até n − 1 faça
9.      Para cada subconjunto S ⊆ V(G) \ {r} com |S| = k faça
10.         Para cada vértice v ∈ S faça
11.             DP(S, v) ← 0
12.             Para cada vértice u ∈ S \ {v} faça
13.                 Se u ∈ N(v) e DP(S \ {v}, u) = 1 então
```

```
                    DP(S, v) ← 1
                    π(S, v) ← u
                    Interrompa

Para cada vértice v ∈ N(r) faça
    Se DP(V(G) \ {r}, v) = 1 então
        C ← {r, v}
        S ← V(G) \ {r}
        Enquanto S ≠ ∅
            u ← π(S, v)
            C ← C ∪ v, u
            S ← S \ {v}
            v ← u
        Retorne C

Retorne ∅
```

A complexidade do algoritmo é claramente dominada pelo laço principal responsável pelo cálculo recursivo da função DP. Observando que são considerados $\Theta(2^n)$ subconjuntos possíveis S e que, para cada um deles, o tempo de processamento é da ordem de $\mathcal{O}(n^2)$, concluímos que o tempo total de execução do algoritmo é $\mathcal{O}(n^2 2^n)$, portanto exponencial[34].

[34] Apesar de exponencial, esse desempenho é substancialmente melhor do que o do algoritmo trivial que testa todas as permutações dos vértices, o qual apresenta custo $\mathcal{O}(n!)$. Além disso, note que o algoritmo requer espaço $\Theta(n2^n)$ para armazenar a tabela de programação dinâmica.

Já o algoritmo polinomial e determinístico que admite falhas, denominado HAM, foi proposto por Bollobás, Fenner e Frieze [30], e baseia-se na técnica de *rotação de caminhos* introduzida por Pósa [31]. Dado um caminho $P = (v_1, v_2, \ldots, v_k)$ e uma aresta $e = \{v_i, v_k\}$, para algum $1 \leq i \leq k-2$, é possível rotacionar P removendo a aresta $\{v_i, v_{i+1}\}$ e adicionando e, obtendo um novo caminho $Q = (v_1, \ldots, v_i, v_k, \ldots, v_{i+1})$ de mesmo comprimento e com o mesmo conjunto de vértices, mas com uma extremidade distinta.

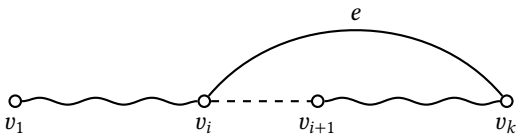


Figura 6: Rotação de Pósa, em que a aresta tracejada é removida e a aresta e é adicionada ao caminho original.

O algoritmo assume que o grafo de entrada G é conexo e possui grau mínimo $\delta(G) \geq 2$, pois, caso contrário, G certamente não contém um ciclo hamiltoniano[35]. A estratégia do algoritmo consiste em analisar caminhos candidatos de comprimento k, tentando estendê-los para obter um caminho de comprimento $k+1$ ou, eventualmente, um ciclo hamiltoniano.

[35] Note que ambas as propriedades podem ser verificadas em tempo $\mathcal{O}(n)$.

Suponha que, no estágio k, o algoritmo analise um caminho P_k. Se existir um vértice fora de P_k adjacente a uma de suas extremidades, o caminho é

estendido diretamente (operação de *extensão*), e o algoritmo avança para o estágio $k+1$. Caso as extremidades de P_k sejam adjacentes, obtém-se um ciclo C_{k+1}, que pode ser convertido em um caminho de comprimento $k+1$ explorando a conectividade de G, isto é, utilizando uma aresta que conecta C_{k+1} a um vértice fora de C_{k+1} (operação de *extensão de ciclo*), permitindo novamente avançar para o estágio $k+1$. Se nenhuma dessas condições for satisfeita, a hipótese $\delta(G) \geq 2$ garante a existência de uma aresta entre uma extremidade de P_k e um vértice interno do caminho, possibilitando a realização de uma rotação de P_k (operação de *rotação*) e a obtenção de um novo caminho candidato Q_k, também de comprimento k. O algoritmo então reaplica os procedimentos anteriores aos novos caminhos gerados, até que seja possível avançar para o estágio $k+1$ ou até que o número de rotações realizadas sobre um mesmo caminho ultrapasse um determinado limite.

Algoritmo 6 (*HAM*)

A função HAM recebe como entrada um grafo conexo G com grau mínimo $\delta(G) \geq 2$ e retorna um ciclo hamiltoniano em G ou a resposta `Erro`. A função auxiliar ROTACIONAR realiza a rotação de um caminho P utilizando uma aresta e. Denotamos por $\mathcal{P}$ o conjunto atual de caminhos candidatos em G, e por $\delta(P_k)$ o número de rotações efetuadas para obter o caminho P_k a partir do caminho inicial de comprimento k. Por fim, denotamos por $N(v)$ a vizinhança de um vértice v em G.

Entrada: Grafo conexo $G = (V, E)$ com n vértices tal que $\delta(G) \geq 2$.

Saída: Um ciclo hamiltoniano C em G ou `Erro`.

HAM(G)

1. Escolha uma aresta $\{u, v\} \in E(G)$
2. $P_1 \leftarrow (u, v)$
3. $\delta(P_1) \leftarrow 0$
4. $\mathcal{P} \leftarrow \{P_1\}$

5. **Enquanto** $\mathcal{P} \neq \emptyset$ **faça**
6. Escolha um caminho $P_k = (v_1, \ldots, v_{k+1}) \in \mathcal{P}$
7. $\mathcal{P} \leftarrow \mathcal{P} \setminus \{P_k\}$
8. **Para cada** vértice extremo $v \in \{v_1, v_{k+1}\}$ **faça**
9. **Para cada** vértice $u \in N(v)$ **faça**
10. **Se** $u \notin P_k$ **então**
11. $P_{k+1} \leftarrow P_k \cup \{u, v\}$ ▷ *Extensão*
12. $\delta(P_{k+1}) \leftarrow 0$
13. $\mathcal{P} \leftarrow \{P_{k+1}\}$ ▷ *Sobrescreve* $\mathcal{P}$
14. **Continue o laço principal**
15. **Senão se** $u \in \{v_1, v_{k+1}\}$ **então**

```
16.              C_{k+1} ← P_k ∪ {v_1, v_{k+1}}
17.              Se V(C_{k+1}) = V(G) então
18.                 Retorne C_{k+1}          ▷ Ciclo hamiltoniano
19.              Para cada vértice v_i ∈ P_k \ {v_1, v_{k+1}} faça
20.                 Para cada vértice w ∈ N(v_i) faça
21.                    Se w ∉ P_k então
22.                       P_{k+1} ← C_{k+1} ∪ {v_i, w} \ {v_{i−1}, v_i}
▷ Extensão
23.                       δ(P_{k+1}) ← 0
24.                       P ← {P_{k+1}}          ▷ Sobrescreve P
25.                       Continue o laço principal
26.           Senão se δ(P_k) ≤ 2 então
27.              Q_k ← ROTACIONAR(P_k, {u, v})          ▷ Rotação
28.              δ(Q_k) ← δ(P_k) + 1
29.              P ← P ∪ {Q_k}

30.  Retorne Erro
```

Observando que um caminho de comprimento k pode gerar no máximo $2(k-2)$ rotações, segue que, no estágio k, o algoritmo analisa no máximo $\mathcal{O}(k^3)$ caminhos. A análise de cada caminho requer tempo $\mathcal{O}(k)$, exceto no caso de uma extensão de ciclo, que demanda tempo $\mathcal{O}(k^2)$, porém, tal extensão ocorre no máximo uma vez por estágio. Assim, o tempo total de execução no estágio k é $\mathcal{O}(k^4)$. Consequentemente, o tempo total de execução do algoritmo é dado por

$$\sum_{k=1}^{n-1} \mathcal{O}(k^4) = \mathcal{O}(n^5).$$

Mostraremos que o Algoritmo 6 falha num grafo $G \sim G(n, 1/2)$ com probabilidade exponencialmente baixa. Incentivamos o leitor a consultar a Seção 3.1 para se familiarizar com as técnicas de análise em grafos aleatórios.

Primeiro, demonstraremos dois resultados auxiliares[36]. Para simplificar os cálculos, ignoramos questões de arredondamento, que não afetam as estimativas assintóticas.

[36] Os resultados poderiam ser formulados em termos de εn, para alguma constante $\varepsilon \in (0, 1/2)$. Por simplicidade, tomamos $\varepsilon = 1/10$.

Para um grafo G, denotamos por $d(v)$ o grau do vértice $v \in V(G)$ no grafo G.

Lema 3. *Seja $G \sim G(n, 1/2)$. Dizemos que um vértice $v \in V(G)$ é pequeno se $d(v) \leq n/10$. Então,*

$$\Pr(G \text{ contém ao menos dois vértices pequenos}) = \mathcal{O}(n^4 2^{-n}).$$

Demonstração. Seja X a variável aleatória que conta o número de vértices pequenos em G.

Suponha que $X \geq 2$. Então, existe um conjunto $S \subseteq V(G)$ com $|S| = 2$ tal que, para todo vértice $v \in S$,

$$d_{V(G)\setminus S}(v) \leq n/10,$$

em que $d_{V(G)\setminus S}$ denota o grau de v restrito a $V(G) \setminus S$.

Consequentemente, pela cota da união sobre todos os pares de vértices $S \subseteq V(G)$, temos

$$\Pr(X \geq 2) \leq \binom{n}{2} \Pr\left(\forall v \in S \,:\, d_{V(G)\setminus S}(v) \leq n/10\right).$$

Fixado um par de vértices S, note que o número de vizinhos de cada vértice de S em $V(G) \setminus S$ é uma variável aleatória com distribuição binomial com parâmetros $n-2$ e $1/2$. Além disso, $d_{V(G)\setminus S}(u)$ e $d_{V(G)\setminus S}(v)$ são independentes para quaisquer $u, v \in S$. Portanto, utilizando a função de distribuição acumulada da variável aleatória binomial, obtemos

$$\Pr(\forall v \in S :\, d_{V(G)\setminus S}(v) \leq n/10) = \left[\Pr(d_{V(G)\setminus S}(v) \leq n/10)\right]^2,$$
$$\Pr(\forall v \in S :\, d_{V(G)\setminus S}(v) \leq n/10) = \left[2^{-(n-2)} \sum_{k=0}^{n/10} \binom{n-2}{k}\right]^2.$$

Observe que o somatório é dominado pelo último termo, de modo que

$$\sum_{k=0}^{n/10} \binom{n-2}{k} \leq (n/10+1)\binom{n-2}{n/10}.$$

Usando $\binom{n}{k} \leq (en/k)^k$, obtemos

$$\Pr(X \geq 2) \leq Cn^4 2^{-2n} (10e)^{n/5} = Cn^4 2^{(-2+\lg(10e)/5)n},$$

para alguma constante absoluta C.

Como $\lg(10e)/5 < 1$, então

$$\Pr(X \geq 2) = \mathcal{O}(n^4 2^{-n}),$$

concluindo a prova. □

Lema 4. *Seja $G \sim G(n, 1/2)$. Então,*

$$\Pr\left(\exists X, Y \subseteq V(G) :\, X \cap Y = \emptyset, |X| = |Y| = n/20 \text{ e } E(X,Y) = \emptyset\right) = \mathcal{O}(2^{-n}),$$

em que $E(X,Y)$ denota o conjunto de arestas entre X e Y.

Demonstração. O número de escolhas para conjuntos disjuntos $X, Y \subseteq V(G)$ com $|X| = |Y| = n/20$ é dado por

$$\binom{n}{n/20}\binom{n-n/20}{n/20} \leq \binom{n}{n/20}^2.$$

Para um dado par X, Y, há $(n/20)^2$ possíveis arestas entre X e Y. Portanto,

$$\Pr(E(X,Y) = \emptyset) = 2^{-(n/20)^2}.$$

Logo, pela cota da união,

$$\Pr\left(\exists X, Y \subseteq V(G) : \ X \cap Y = \emptyset, |X| = |Y| = n/20 \text{ e } E(X,Y) = \emptyset\right) \leq \binom{n}{n/20}^2 2^{-(n/20)^2}.$$

Usando $\binom{n}{k} \leq (en/k)^k$, segue que

$$\binom{n}{n/20}^2 2^{-(n/20)^2} \leq (20e)^{n/10}\, 2^{-(n/20)^2} = 2^{-\Theta(n^2)} = \mathcal{O}(2^{-n}),$$

como desejado. □

Podemos então enunciar o resultado principal:

Teorema 16. *Seja $G \sim G(n, 1/2)$. Então,*

$$\Pr(\text{HAM } \textit{falha em } G) = \mathcal{O}(n^4 2^{-n}).$$

Demonstração. Novamente, chamamos de *pequeno* qualquer vértice $v \in V(G)$ com $d(v) \leq n/10$. Definimos então a variável aleatória

$$K = |\{v \in V(G) \ : \ d(v) \leq n/10\}|,$$

que conta o número de vértices pequenos em G.

Logo, podemos decompor a probabilidade de falha do algoritmo segundo os valores de K, de forma que

$$\begin{aligned}\Pr(\text{HAM falha em } G) &= \Pr(\text{HAM falha em } G \cap K \leq 1) \\ &\quad + \Pr(\text{HAM falha em } G \cap K \geq 2), \\ \Pr(\text{HAM falha em } G) &\leq \Pr(\text{HAM falha em } G \mid K \leq 1) + \Pr(K \geq 2),\end{aligned}$$

em que a desigualdade segue diretamente de desigualdades fundamentais de probabilidade.

Pelo Lema 3, segue que

$$\Pr(K \geq 2) = \mathcal{O}(n^4 2^{-n}).$$

Assim, basta mostrar que a probabilidade de falha do algoritmo quando $K \leq 1$ também é exponencialmente baixa. Mostraremos que a falha de HAM nesse caso implica a existência de dois subconjuntos lineares sem arestas entre si, evento que, pelo Lema 4, tem probabilidade exponencialmente pequena.

Suponha, portanto, que $K \leq 1$ e que o algoritmo falhe em G no estágio k. Seja $P_k = (v_1, \dots, v_k)$ o caminho inicial considerado no estágio k. Perceba que a falha do algoritmo implica que $\{v_1, v_k\} \notin E(G)$ e que todo vizinho de v_1 e de v_k pertence à P_k.

Além disso, podemos supor, sem perda de generalidade, que $d(v_1) > n/10$ e $d(v_k) > n/10$. De fato, como $K \leq 1$, no máximo um dos extremos pode ter grau pequeno. Note que o algoritmo certamente executa uma rotação nesse extremo, obtendo um novo caminho Q_k no qual a suposição se aplica. Os argumentos abaixo permanecem válidos para esse novo caminho.

Seja i o menor índice tal que ocorre uma das seguintes situações:

- Pelo menos metade dos vizinhos de v_1 estão à esquerda de v_i, isto é, estão em $\{v_2, \dots, v_{i-1}\}$;
- Pelo menos metade dos vizinhos de v_k estão à esquerda de v_i, isto é, estão em $\{v_2, \dots, v_{i-1}\}$.

Como todos os vizinhos de v_1 e v_k pertencem a P_k, tal índice certamente existe. Analisamos então os seguintes casos:

- Se metade dos vizinhos de v_1 estão à esquerda de v_i, então existe um conjunto
$$X \subseteq \{v_2, \dots, v_{i-1}\}, \text{ com } |X| \geq n/20,$$
tal que, para todo $x \in X$, é possível obter, por uma rotação de P_k, um caminho $Q_k = (x, \dots, v_i, \dots, v_k)$.

  De fato, para cada vizinho v_j de v_1, com $3 \leq j \leq i-1$, a rotação de P_k através da aresta $\{v_1, v_j\}$ produz um caminho com extremo esquerdo v_{j-1}.

  Observe também que, no caminho Q_k, a ordem dos vértices $\{v_i, \dots, v_k\}$ é preservada em relação à P_k. Como mais da metade dos vizinhos de v_k estão à direita de v_i, existe um conjunto
$$Y \subseteq \{v_{i+1}, \dots, v_{k-1}\}, \text{ com } |Y| \geq n/20,$$

tal que, para todo $y \in Y$ e todo $x \in X$, é possível obter, por uma rotação de Q_k, um caminho $R_k = (x, \dots, v_i, \dots, y)$.

De fato, para cada vizinho v_ℓ de v_k, com $i \leq \ell \leq k-2$, a rotação de Q_k através da aresta $\{v_\ell, v_k\}$ produz um caminho com extremo direito $v_{\ell+1}$.

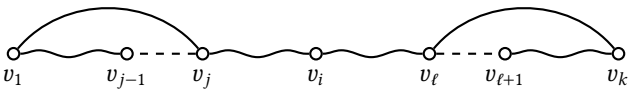


Figura 7: Exemplo de rotações sucessivas no estágio k via vizinhos de v_1 e de v_k, em que as arestas tracejadas são removidas e as curvas adicionadas, produzindo novos extremos.

Finalmente, note que $\{x, y\} \notin E(G)$, para todo $x \in X$ e $y \in Y$. Caso contrário, o algoritmo faria uma extensão de ciclo em $R_k \cup \{x, y\}$, o que contradiz a hipótese de falha.

Concluímos que existem dois subconjuntos disjuntos $X, Y \subseteq V(G)$, ambos de tamanho ao menos $n/20$, sem arestas entre si;

- No caso contrário, em que pelo menos metade dos vizinhos de v_k estão à esquerda de v_i, o argumento é análogo, bastando adaptar a segunda rotação para levar em conta a inversão da ordem de alguns vértices de P_k. Em particular, concluímos novamente a existência de dois subconjuntos disjuntos $X, Y \subseteq V(G)$, ambos de tamanho ao menos $n/20$, sem arestas entre si.

Logo, pelo Lema 4,

$$\Pr(\text{HAM falha em } G \mid K \leq 1) = \mathcal{O}(2^{-n}).$$

Portanto,

$$\Pr(\text{HAM falha em } G) = \mathcal{O}(2^{-n}) + \mathcal{O}(n^4 2^{-n}) = \mathcal{O}(n^4 2^{-n}),$$

como afirmado. □

Podemos então combinar os dois algoritmos apresentados, resultando no algoritmo descrito abaixo.

Algoritmo 7 (*HAMCombinado*)

A função HAMCombinado recebe como entrada um grafo G e retorna um ciclo hamiltoniano em G, se existir, e $\emptyset$, caso contrário.

Entrada: Grafo $G = (V, E)$ com n vértices.

Saída: Um ciclo hamiltoniano C em G ou $\emptyset$, caso tal ciclo não exista.

```
HAMCombinado(G)
1.  Se G não é conexo ou δ(G) < 2 então
2.     Retorne ∅
3.  Senão
4.     R ← HAM(G)
5.     Se R ≠ Erro então
6.        Retorne R
7.     Senão
8.        Retorne BellmanHeldKarp(G)
```

Corolário 5. *Seja G um grafo com n vértices escolhido uniformemente ao acaso. Então, o Algoritmo 7 resolve o problema do Ciclo Hamiltoniano em G em tempo de execução esperado*

$$\mathcal{O}(n^5) + \mathcal{O}(n^4 2^{-n}) \cdot \mathcal{O}(n^2 2^n) = \mathcal{O}(n^6).$$

É importante ressaltar que, no trabalho original, Bollobás, Fenner e Frieze [30] demonstraram, com argumentos mais sofisticados, um resultado ainda mais significativo sobre o algoritmo: a probabilidade assintótica de sucesso de HAM coincide com a probabilidade de que o grafo aleatório contenha um ciclo hamiltoniano.

3.2.2 Lema Minimax de Yao

A teoria dos jogos é um ramo da matemática dedicado à modelagem e análise de interações estratégicas entre agentes racionais, chamadas de *jogos*. Seus conceitos são amplamente aplicados em ciências econômicas, sociais e comportamentais[37]. Apresentamos a seguir alguns conceitos básicos necessários para compreender o resultado de Yao. Para uma introdução completa à teoria dos jogos, veja [32].

Considere o seguinte jogo. Alice esconde um objeto em uma de suas mãos, sem que Bruno observe sua escolha. Em seguida, Bruno tenta adivinhar em qual das mãos o objeto foi escondido. Se Bruno acertar, Alice lhe paga R$ 1. Caso contrário, Alice recebe R$ 1 de Bruno.

Trata-se de um jogo de dois jogadores *com soma nula*, pois o ganho de um jogador coincide com a perda do outro. As possíveis recompensas de

[37] Embora algumas ideias relacionadas já existissem anteriormente, a teoria dos jogos consolidou-se como área de estudo apenas na década de 1920, sobretudo a partir do trabalho de John von Neumann. Esse desenvolvimento culminou na obra *Theory of Games and Economic Behavior* (1944), escrita em coautoria com Oskar Morgenstern, que marcou o estabelecimento formal do campo. A partir de então, a teoria dos jogos expandiu-se significativamente e passou a ser aplicada em diversas áreas, incluindo economia e ciência política a partir da década de 1950, e biologia evolutiva a partir da década de 1970 [32].

Alice podem ser representadas pela seguinte matriz:

$$\mathbf{M} = \begin{array}{c} \begin{matrix} \text{Esq.} & \text{Dir.} \end{matrix} \\ \begin{bmatrix} -1 & 1 \\ 1 & -1 \end{bmatrix} \begin{matrix} \text{Esq.} \\ \text{Dir.} \end{matrix} \end{array}.$$

As linhas correspondem às ações de Alice (esconder o objeto na mão esquerda ou direita), enquanto as colunas correspondem às ações de Bruno (apostar na mão esquerda ou direita). A entrada $\mathbf{M}_{i,j}$ representa o ganho de Alice (resp. perda de Bruno) quando Alice escolhe a ação i e Bruno escolhe a ação j. De modo geral, qualquer jogo de dois jogadores com soma nula pode ser descrito por uma *matriz de recompensas* $\mathbf{M} \in \mathbb{R}^{n \times m}$.

Assumimos que ambos os jogadores desejam maximizar seus próprios ganhos. Como os interesses são opostos, o jogo é inerentemente adversarial.

Chamamos de *estratégia* uma regra que determina a ação escolhida pelo jogador. Quando essa escolha é determinística, a estratégia é dita *pura*. No jogo acima, uma estratégia pura consiste simplesmente em escolher uma das mãos. Assumiremos que este é um jogo de informação nula, isto é, nenhum jogador possui informação prévia sobre a estratégia do adversário.

Se Alice escolher uma estratégia pura i, então, no pior caso, sua recompensa será

$$\min_j \mathbf{M}_{i,j},$$

independentemente da ação de Bruno. Assim, uma estratégia pura ótima para Alice consiste em escolher a linha i que maximiza essa quantidade, garantindo-lhe um ganho de

$$\max_i \min_j \mathbf{M}_{i,j}.$$

De forma análoga, se Bruno escolher uma estratégia pura j, sua maior perda possível é

$$\max_i \mathbf{M}_{i,j},$$

de modo que a menor perda que ele pode garantir é

$$\min_j \max_i \mathbf{M}_{i,j}.$$

Quando esses dois valores coincidem, dizemos que o jogo admite um *equilíbrio puro*: um par de ações (i, j) tal que nenhum dos jogadores tem incentivo a desviar unilateralmente, pois cada um já está obtendo o melhor resultado que pode garantir sem conhecer a escolha do oponente.

Não é difícil verificar que o jogo descrito acima não possui equilíbrio puro. Isso motiva a introdução do conceito de *estratégia mista*. Uma estraté-

gia mista é um vetor $\mathbf{x} = (x_1, \ldots, x_n)$, em que x_i representa a probabilidade do jogador escolher a ação i, com $\sum_{i=1}^{n} x_i = 1$. Assim, o jogador passa a escolher suas ações de forma aleatória, segundo uma distribuição de probabilidade. Observe que estratégias puras são casos particulares de estratégias mistas, nas quais uma ação é escolhida com probabilidade 1 e as demais com probabilidade 0.

Dadas estratégias mistas $\mathbf{x}$ para Alice e $\mathbf{y}$ para Bruno, a *recompensa esperada* de Alice é dada por

$$\mathbf{x}^\top \mathbf{M}\mathbf{y} = \sum_{i=1}^{n} \sum_{j=1}^{m} x_i \mathbf{M}_{i,j} y_j$$

Por argumentos análogos aos anteriores, a maior recompensa esperada que Alice pode garantir, independentemente da estratégia de Bruno, é

$$\max_{\mathbf{x}} \min_{\mathbf{y}} \mathbf{x}^\top \mathbf{M}\mathbf{y},$$

enquanto a menor perda esperada que Bruno pode garantir, independentemente da estratégia de Alice, é

$$\min_{\mathbf{y}} \max_{\mathbf{x}} \mathbf{x}^\top \mathbf{M}\mathbf{y},$$

em que o máximo e o mínimo variam sobre todas as distribuições de probabilidade válidas.

O Teorema Minimax de von Neumann [33], enunciado a seguir, afirma que esses dois valores coincidem para todo jogo de dois jogadores com soma nula, garantindo assim a existência de um *equilíbrio misto*.

Teorema 17. *Para todo jogo de dois jogadores com soma nula especificado por uma matriz de recompensas* $\mathbf{M}$*, vale que*

$$\max_{\mathbf{x}} \min_{\mathbf{y}} \mathbf{x}^\top \mathbf{M}\mathbf{y} = \min_{\mathbf{y}} \max_{\mathbf{x}} \mathbf{x}^\top \mathbf{M}\mathbf{y}$$

Em outras palavras, o maior valor esperado que Alice pode garantir escolhendo uma estratégia mista é igual à menor perda esperada que Bruno pode assegurar ao escolher uma estratégia mista. Esse valor comum é chamado de *valor do jogo*.

A demonstração desse teorema foge ao escopo deste trabalho.

Note que, para uma estratégia mista fixa $\mathbf{x}$, a expressão $\mathbf{x}^\top \mathbf{M}\mathbf{y}$ é uma função linear em $\mathbf{y}$ e, portanto, é minimizada ao concentrar toda a massa de probabilidade em uma coordenada y_j que minimize o coeficiente correspondente. Então, se Bruno conhece a distribuição $\mathbf{x}$ utilizada por Alice, sua melhor estratégia é sempre uma estratégia pura. O mesmo argumento vale

de forma simétrica para Alice.

Essa observação leva a uma versão simplificada do Teorema Minimax, atribuída a Loomis [34].

Corolário 6. *Seja* $\mathbf{e}_k$ *o vetor unitário com* 1 *na* k*-ésima coordenada e* 0 *nas demais.*

Para todo jogo de dois jogadores com soma nula especificado por uma matriz de recompensas $\mathbf{M}$*, vale que*

$$\max_{\mathbf{x}} \min_{j} \mathbf{x}^\top \mathbf{M} \mathbf{e}_j = \min_{\mathbf{y}} \max_{i} \mathbf{e}_i^\top \mathbf{M} \mathbf{y}$$

A seguir, descrevemos como esses resultados de teoria dos jogos podem ser utilizados para estabelecer limitantes inferiores para o desempenho de algoritmos aleatorizados.

3.2.2.1 Analisando algoritmos através de jogos

Considere um problema Π e seja Ω_n o conjunto de todas as instâncias de Π de tamanho n. Seja $\mathcal{A}$ o conjunto de todos os algoritmos determinísticos que resolvem Π. Para que a técnica seja aplicável, supomos que $\mathcal{A}$ e Ω_n sejam finitos.

Definimos então um jogo de soma nula associado a Π, descrito por uma matriz $\mathbf{M}$: cada linha corresponde a uma instância $x \in \Omega_n$ e cada coluna corresponde a um algoritmo $A \in \mathcal{A}$. A entrada $\mathbf{M}_{x,A} = T_A(x)$ representa o tempo de execução[38] do algoritmo A na instância x. Nesse jogo, escolhemos um algoritmo (coluna) com o objetivo de minimizar o custo, enquanto um adversário escolhe uma instância (linha) com o objetivo de maximizá-lo.

[38] Note que poderíamos igualmente considerar outras medidas de custo, como uso de espaço ou qualidade da solução.

Fixado um algoritmo determinístico $A \in \mathcal{A}$, o resultado do jogo corresponde exatamente à análise de pior caso: o adversário seleciona uma instância $x \in \Omega_n$ de forma a maximizar $T_A(x)$. Assim, uma estratégia pura ótima para o jogador das colunas corresponde a um algoritmo determinístico ótimo, e

$$\min_{A \in \mathcal{A}} \max_{x \in \Omega_n} T_A(x)$$

é o tempo de execução no pior caso do melhor algoritmo determinístico para Π, que chamamos de *complexidade determinística* do problema.

A abordagem torna-se mais interessante quando consideramos algoritmos aleatorizados. Um algoritmo Las Vegas $\mathcal{R}$ pode ser visto como uma distribuição de probabilidade sobre o conjunto $\mathcal{A}$ de algoritmos determinísticos. Note que isso corresponde exatamente a uma estratégia mista no jogo

M. Naturalmente, o tempo de execução de $\mathcal{R}$ em uma instância x é dado por

$$T_{\mathcal{R}}(x) = \mathop{\mathbb{E}}_{A \sim \mathcal{R}}[T_A(x)],$$

isto é, o tempo de execução esperado. Uma estratégia mista ótima corresponde, portanto, a um algoritmo Las Vegas ótimo. Definimos a *complexidade aleatorizada* do problema como

$$\min_{\mathcal{R}} \max_{x \in \Omega_n} T_{\mathcal{R}}(x),$$

ou seja, o melhor desempenho esperado no pior caso entre todos os algoritmos Las Vegas.

Analogamente, seja $\mathcal{D}$ uma distribuição de probabilidade sobre Ω_n. Tal distribuição pode ser interpretada como uma estratégia mista do adversário. Para um algoritmo determinístico A, o tempo de execução esperado sob $\mathcal{D}$ é

$$\mathop{\mathbb{E}}_{x \sim D}[T_A(x)] = \texttt{Médio}_A^{\mathcal{D}}(n).$$

Definimos então a *complexidade distribucional* do problema como

$$\max_{\mathcal{D}} \min_{A \in \mathcal{A}} \texttt{Médio}_A^{\mathcal{D}}(n),$$

isto é, o desempenho esperado do melhor algoritmo determinístico contra a pior distribuição de entradas. Observe que essa quantidade é, em geral, menor ou igual à complexidade determinística, pois o algoritmo conhece a distribuição $\mathcal{D}$.

Pelo Teorema 17 a complexidade aleatorizada e a complexidade distribucional coincidem. Esse é precisamente o *Lema Minimax de Yao*, que enunciamos a seguir em uma forma relaxada, obtida a partir do Corolário 6, suficiente para obter limites inferiores.

Corolário 7. *Seja* Π *um problema,* Ω_n *o conjunto (finito) de todas as instâncias de* Π *de tamanho* n*, e* $\mathcal{A}$ *o conjunto (finito) de todos os algoritmos determinísticos que resolvem* Π*. Então, para quaisquer distribuições* $\mathcal{D}$ *sobre* Ω_n *e* $\mathcal{R}$ *sobre* $\mathcal{A}$*, vale que*

$$\min_{A \in \mathcal{A}} \texttt{Médio}_A^{\mathcal{D}}(n) \leq \max_{x \in \Omega_n} T_{\mathcal{R}}(x)$$

Em particular, o desempenho médio do melhor algoritmo determinístico sob uma distribuição arbitrária $\mathcal{D}$ fornece um limitante inferior para o tempo de execução esperado de qualquer algoritmo Las Vegas para o problema Π. Isso é extremamente útil, pois permite escolher uma distribuição conveniente $\mathcal{D}$ e provar que todo algoritmo determinístico[39] tem custo esperado ao menos C. Pelo Lema de Yao, segue então que a complexidade aleatorizada

[39] É importante ressaltar que o algoritmo determinístico é assumido conhecer a distribuição $\mathcal{D}$.

do problema também é ao menos C.

3.3 Análise suavizada

Como vimos na Seção 3.2, a análise de caso médio é a alternativa frequentemente adotada na análise probabilística de algoritmos. Embora tal abordagem evite o pessimismo do pior caso, ela depende fortemente da distribuição de probabilidade atribuída ao espaço de instâncias, o que pode torná-la tão artificial quanto a análise de pior caso. Na prática, é difícil determinar qual distribuição descreve adequadamente as instâncias reais de um problema. Além disso, instâncias aleatórias frequentemente apresentam, com alta probabilidade, certas propriedades estruturais que dominam o comportamento médio, o que pode enviesar a interpretação do desempenho. Assim, instâncias aleatórias não devem ser confundidas com instâncias típicas: instâncias aleatórias são, na verdade, um tipo bem particular de instâncias [35].

Com o objetivo de contornar as limitações de ambos os paradigmas anteriores, Spielman e Teng propuseram a *análise suavizada de algoritmos* [36]. Essa abordagem parte da observação de que, no mundo real, as instâncias de problemas estão frequentemente sujeitas a pequenas perturbações aleatórias, decorrentes de diversas fontes, como erros de medição, restrições de projeto e, de maneira geral, por fatores de aleatoriedade inerentes à ocorrência de instâncias particulares. Dessa forma, as instâncias reais não são completamente adversariais, mas também não são totalmente aleatórias.

A partir dessa ideia, define-se a *medida suavizada* de um algoritmo em uma dada instância como o valor esperado de seu desempenho sob pequenas perturbações aleatórias dessa instância. Já sua *complexidade suavizada* é definida como o máximo dessa medida suavizada sobre todas as instâncias possíveis de entrada.

Definição 10 (σ*–perturbação*)**.** *Dado um grafo G com n vértices e $\sigma \in (0, 1)$, definimos a σ–perturbação de G como o grafo aleatório obtido pela remoção de cada aresta de G com probabilidade σ e pela adição de cada não aresta de G com probabilidade σ. Denotamos por $\mathcal{G}(G, \sigma)$ a distribuição sobre grafos com n vértices assim obtida.*

Equivalentemente, se $G_\sigma \sim \mathcal{G}(G, \sigma)$, então

$$\Pr[e \in E(G_\sigma)] = (1 - \sigma)\mathbb{1}_e + \sigma(1 - \mathbb{1}_e),$$

em que $\mathbb{1}_e = 1$ se $e \in E(G)$ e $\mathbb{1}_e = 0$ caso contrário.

Note também que

$$G_\sigma = G \Delta H,$$

em que $H \sim G(n, \sigma)$ é um grafo aleatório de Erdős–Rényi e $G\Delta H$ denota o grafo sobre o mesmo conjunto de vértices de G, cujas arestas são dadas pela diferença simétrica entre as arestas de G e de H.

Definimos então a complexidade suavizada de um algoritmo em grafos.

Definição 11 (*Complexidade suavizada*)**.** *Seja A um algoritmo e seja $\mathcal{G}_n$ o conjunto de todos os grafos com n vértices. Para cada grafo $G \in \mathcal{G}_n$, seja $G_\sigma \sim \mathcal{G}(G, \sigma)$ uma σ–perturbação de G. Então, a complexidade suavizada de A é definida por*

$$\mathit{Suavizada}_A^\sigma(n) := \max_{G \in \mathcal{G}_n} \mathbb{E}\left[T_A(G_\sigma)\right].$$

Observe que, ao variar o grafo $G \in \mathcal{G}_n$, obtemos distribuições de probabilidade distintas sobre o espaço $\mathcal{G}_n$. A complexidade suavizada considera, portanto, o pior caso, dentre todas essas distribuições, do tempo de execução esperado de A.

Na análise suavizada o parâmetro σ determina o grau de poder do adversário, estabelecendo uma transição contínua entre os paradigmas de pior caso e de caso médio: quando a perturbação tende a zero, voltamos à análise de pior caso; por outro lado, quando a perturbação é suficientemente grande a ponto de dominar as instâncias, a análise aproxima-se do caso médio. Por exemplo, se $\sigma = 1/2$ então $G_\sigma \sim G(n, 1/2)$ para todo grafo G.

Note que o tempo de execução é analisado em função tanto do tamanho da entrada (n) quanto da magnitude da perturbação (σ). Podemos então caracterizar algoritmos que apresentam complexidade suavizada polinomial.

Definição 12 (*Complexidade suavizada polinomial*)**.** *Dizemos que um algoritmo A possui complexidade suavizada polinomial se existem constantes positivas $n_0, \sigma_0, c, k,$ e α tais que, para todo $n \geq n_0$ e $\sigma \geq \sigma_0$,*

$$\mathit{Suavizada}_A^\sigma(n) \leq c \cdot \frac{n^k}{\sigma^\alpha}.$$

Observe que um algoritmo ter complexidade suavizada polinomial indica que as instâncias de desempenho superpolinomial (caso existam) são "instáveis", e, portanto, é razoável esperar que não ocorram em situações práticas.

É importante ressaltar que definimos a análise suavizada restrita ao contexto de algoritmos em grafos, considerando ainda um modelo específico de

perturbação. Entretanto, naturalmente, a noção de complexidade suavizada se estende a diversos outros domínios e, mesmo no contexto de grafos, existem vários modelos distintos de perturbação (veja, por exemplo, [37,38], e os modelos citados em [35]). Optamos por este modelo em particular por sua proximidade com os ideais originais propostos na análise suavizada.

Contudo, note que esse modelo simples pode ser inadequado para certos problemas, pois as entradas são perturbadas aleatoriamente de maneira indiscriminada. Por exemplo, ao analisarmos um algoritmo que decide se um grafo é hamiltoniano, é desejável que a perturbação não altere a hamiltonicidade do grafo original; caso contrário, a análise perde o sentido[40]. Nesses contextos, torna-se necessário impor restrições às perturbações, de modo que certas estruturas da entrada sejam preservadas. Nesse sentido, Spielman e Teng propuseram o estudo de *σ–perturbações que preservam propriedades*, discutidas brevemente em [39].

[40] Isso ocorre porque, ao modificar com alta probabilidade propriedades estruturais relevantes da instância original, a perturbação pode introduzir características que tornam o problema artificialmente mais fácil, descaracterizando a dificuldade intrínseca que se deseja analisar, num efeito análogo ao viés presente na análise de caso médio.

Embora já existam alguns resultados a respeito da complexidade suavizada de algoritmos em grafos (*e.g.*, [40,41,42]), o paradigma também pode ser naturalmente estendido para avaliar outros aspectos do comportamento de algoritmos. Entre suas possíveis aplicações estão a estimativa da probabilidade de erro de algoritmos aleatórios [39], a avaliação da qualidade das soluções obtidas por algoritmos de aproximação [43] e a determinação da menor perturbação necessária para que um grafo qualquer adquira determinada propriedade [46].

Apesar dos avanços existentes na literatura, ainda há amplas oportunidades de investigação, especialmente no contexto de análise de algoritmos em grafos.

Apresentaremos a seguir um resultado recente sobre a análise suavizada de um algoritmo para o *problema do Isomorfismo de Grafos*[41]. As demonstrações dos resultados apresentados a seguir fogem ao escopo deste texto.

[41] O problema do Isomorfismo de Grafos é um dos poucos problemas conhecidos que pertencem a $\mathcal{NP}$, mas cuja $\mathcal{NP}$-dificuldade permanece em aberto. Não se conhecem algoritmos polinomiais para o problema. László Babai [47] reivindicou a existência de um algoritmo quase–polinomial para o problema, isto é, com tempo de execução $\mathcal{O}(e^{(\log n)^{\mathcal{O}(1)}})$, mas sua versão final ainda não foi completamente publicada.

Isomorfismo de grafos

Dado um grafo G e um grafo H, determine se H é isomorfo a G.

Na análise probabilística do Isomorfismo de Grafos, não é ideal lidar com dois grafos aleatórios G e H, pois, com alta probabilidade, eles não serão isomorfos. Para contornar esse problema, uma estratégia comumente adotada consiste em analisar algoritmos de *rotulação canônica* [26]. Uma rotulação canônica associa rótulos aos vértices de um grafo G de forma que, se outro grafo H for rotulado segundo o mesmo procedimento, então os grafos coincidem se, e somente se, são isomorfos. Isso permite testar o isomorfismo de G com qualquer outro grafo.

Um algoritmo polinomial que tenta construir uma rotulação canônica de um grafo G é o *refinamento de cores*. Inicialmente, todos os vértices de G recebem a mesma cor. No primeiro passo, os vértices são coloridos de acordo com seus graus, na tentativa de diferenciá-los; no passo seguinte, essa informação é refinada levando em conta os graus dos vizinhos; e o processo prossegue de maneira semelhante, até que a coloração se estabilize, ou seja, até que não seja mais possível distinguir vértices com base em critérios adicionais.

Se ao final do processo cada vértice de G receber uma cor distinta, a coloração obtida determina uma rotulação canônica de G, e o grafo pode então ser testado quanto ao isomorfismo com qualquer outro em tempo polinomial.

Entretanto, o refinamento de cores nem sempre é bem-sucedido. Considere, por exemplo, um grafo regular: como todos os vértices têm o mesmo grau, o algoritmo não consegue distinguir nenhum vértice dos demais.

Ainda assim, essa situação é "atípica": de acordo com um resultado de Babai, Erdős e Selkow [48], o algoritmo é capaz de produzir uma rotulação canônica para quase todo grafo, como descrito abaixo.

Teorema 18. *Seja G um grafo escolhido uniformemente ao acaso. Então, com alta probabilidade, o algoritmo de refinamento de cores distingue todos os vértices de G. Como consequência, com alta probabilidade, G pode ser testado quanto ao isomorfismo com qualquer outro grafo em tempo polinomial.*

Recentemente, Anastos, Kwan e Moore [40] refinaram esse resultado ao realizar a análise suavizada do algoritmo, obtendo o resultado a seguir.

Teorema 19. *Fixe uma constante $\delta > 0$ e considere $\sigma \in (0,1)$ tal que $\sigma \geq (1+\delta)\log n/n$. Então, para todo grafo G, com alta probabilidade o algoritmo de refinamento de cores rotula canonicamente o grafo $G_\sigma \sim \mathcal{G}(G,\sigma)$. Em particular, com alta probabilidade, uma σ–perturbação de G pode ser testada quanto ao isomorfismo com qualquer outro grafo em tempo polinomial.*

Ou seja, para todo grafo G, a adição e remoção de aproximadamente $n \log n$ arestas aleatórias[42] é suficiente para que o refinamento de cores seja bem-sucedido com alta probabilidade.

[42] Isso decorre do fato de que, para $H \sim G(n,p)$, o número de arestas de H está concentrado em torno de $p\binom{n}{2} \approx n^2 p$.

Note que não analisamos o tempo de execução do algoritmo – que é sempre polinomial –, mas sim sua probabilidade de sucesso. Assim, se essa probabilidade fosse suficientemente alta, poderíamos recorrer a um algoritmo exato apenas nos raros casos de falha do refinamento de cores. Por argumentos semelhantes aos da Seção 3.2.1, isso resultaria em um algoritmo que sempre produz uma rotulação canônica e cujo tempo de execução suavizado seria polinomial. No entanto, a probabilidade de erro do refinamento de cores no contexto suavizado, conforme demonstrado em [40], não parece

ser pequena o bastante para viabilizar esse resultado, sendo necessário um aperfeiçoamento adicional do limitante de erro.

Curiosamente, apesar da origem algorítmica do paradigma da análise suavizada, há hoje mais trabalhos sobre grafos aleatoriamente perturbados na teoria extremal de grafos (veja os exemplos em [40]) do que no contexto algorítmico propriamente dito.

Concluímos a seção com a seguinte questão: seria possível estender os resultados extremamente positivos da Seção 3.2.1 ao contexto da análise suavizada? Observe que o resultado já vale para $\sigma = 1/2$, pois essa perturbação equivale a considerar um grafo aleatório segundo o modelo $G(n, 1/2)$.

Questão 1. *O Algoritmo 7 tem tempo suavizado polinomial para algum* $\sigma < 1/2$*?*

4 Conclusão

A revisão realizada neste trabalho evidencia que a aleatoriedade vai além de uma simples ferramenta algorítmica, constituindo um elemento essencial para compreender mais profundamente o que computadores são capazes de resolver. Seja pelas contribuições conceituais em complexidade computacional resultantes do estudo de algoritmos aleatorizados, seja pelos modelos mais realistas para análise do desempenho algorítmico. Dessa forma, a aleatoriedade oferece novas perspectivas sobre problemas computacionalmente difíceis, e o estudo contínuo desse tema é crucial para aprofundar nossa compreensão de questões fundamentais em ciência da computação.

Referências

[1] CORMEN, T. H. et al. **Introduction to Algorithms, Third Edition**. 3rd. ed. [s.l.] The MIT Press, 2009.

[2] FORTNOW, L. **2023 A.M. Turing Award Laureate**. Disponível em: <https://amturing.acm.org/award_winners/wigderson_3844537.cfm>.

[3] MOTWANI, R.; RAGHAVAN, P. Randomized algorithms. **ACM Comput. Surv.**, v. 28, n. 1, p. 33–37, mar. 1996.

[4] ARORA, S.; BARAK, B. **Computational Complexity: A Modern Approach**. [s.l.] Cambridge University Press, 2009.

[5] WIGDERSON, A. **Mathematics and computation: A theory revolutionizing technology and science**. [s.l.] Princeton University Press, 2019.

[6] METROPOLIS, N. The Beginning of the Monte Carlo Method. **Los Alamos Science Special Issue**, p. 125–130, 1987.

[7] MOTWANI, R.; RAGHAVAN, P. **Randomized Algorithms**. [s.l.] Cambridge University Press, 1995.

[8] DEVROYE, L. **Non-Uniform Random Variate Generation**. 1. ed. New York, NY: Springer, 1986.

[9] GAREY, M. R.; JOHNSON, D. S.; STOCKMEYER, L. Some simplified NP-complete graph problems. **Theoretical Computer Science**, v. 1, n. 3, p. 237–267, 1976.

[10] BOTLER, F. H. et al. **Combinatória**. [s.l.] Impa, 2022.

[11] KARGER, D. R. **Global min-cuts in RNC, and other ramifications of a simple min-cut algorithm**. Proceedings of the Fourth Annual ACM-SIAM Symposium on Discrete Algorithms. **Anais**...: SODA '93.USA: Society for Industrial; Applied Mathematics, 1993.

[12] KARGE, D. R. Random Sampling in Cut, Flow, and Network Design Problems. **Mathematics of Operations Research**, v. 24, n. 2, p. 383–413, 1999.

[13] RAGHAVAN, P.; TOMPSON, C. D. Randomized rounding: A technique for provably good algorithms and algorithmic proofs. **Combinatorica**, v. 7, n. 4, p. 365–374, 1987.

[14] GOEMANS, M. X.; WILLIAMSON, D. P. Improved approximation algorithms for maximum cut and satisfiability problems using semidefinite programming. **J. ACM**, v. 42, n. 6, p. 1115–1145, nov. 1995.

[15] BABAI, L. **Monte-Carlo Algorithms in Graph Isomorphism Testing**. [s.l.] Université de Montréal, Département de Mathématiques et de Statistique, 1979.

[16] ALON, N.; YUSTER, R.; ZWICK, U. **Color-coding: a new method for finding simple paths, cycles and other small subgraphs within large graphs**. Proceedings of the Twenty-Sixth Annual ACM Symposium on Theory of Computing. **Anais**...: STOC '94.New York, NY, USA: Association for Computing Machinery, 1994.

[17] KARP, R. M. Reducibility among Combinatorial Problems. Em: MILLER, R. E.; THATCHER, J. W.; BOHLINGER, J. D. (Eds.). **Complexity of Computer Computations: Proceedings of a symposium on the Complexity of Computer Computations, held March 20–22, 1972, at the IBM Thomas J. Watson Research Center, Yorktown Heights, New York, and sponsored by the Office of Naval Research, Mathematics Program, IBM World Trade Corporation, and the IBM Research Mathematical Sciences Department**. Boston, MA: Springer US, 1972. p. 85–103.

[18] ALT, H. et al. A method for obtaining randomized algorithms with small tail probabilities. **Algorithmica**, v. 16, n. 4, p. 543–547, 1996.

[19] LUBY, M.; SINCLAIR, A.; ZUCKERMAN, D. Optimal speedup of Las Vegas algorithms. **Information Processing Letters**, v. 47, n. 4, p. 173–180, 1993.

[20] LUBY, M.; ERTEL, W. **Optimal parallelization of Las Vegas algorithms**. (P. Enjalbert, E. W. Mayr, K. W. Wagner, Eds.)STACS 94. **Anais**...Berlin, Heidelberg: Springer Berlin Heidelberg, 1994.

[21] RUAN, Y.; HORVITZ, E.; KAUTZ, H. **Restart Policies with Dependence among Runs: A Dynamic Programming Approach**. (P. Van Hentenryck, Ed.)Principles and Practice of Constraint Programming - CP 2002. **Anais**...Berlin, Heidelberg: Springer Berlin Heidelberg, 2002.

[22] GOMES, C. P.; SELMAN, B. Algorithm portfolios. **Artificial Intelligence**, v. 126, n. 1, p. 43–62, 2001.

[23] GOMES, C. P. Randomized Backtrack Search. Em: MILANO, M. (Ed.). **Constraint and Integer Programming: Toward a Unified Methodology**. Boston, MA: Springer US, 2004. p. 233–291.

[24] BOLLOBÁS, B. Random Graphs. Em: Cambridge Studies em Advanced Mathematics. 2. ed. [s.l.] Cambridge University Press, 2001.

[25] BOLLOBÁS, B.; THOMASON, A. G. Threshold functions. **Combinatorica**, v. 7, n. 1, p. 35–38, 1987.

[26] FRIEZE, A. M. Probabilistic Analysis of Graph Algorithms. Em: TINHOFER, G. et al. (Eds.). **Computational Graph Theory**. Vienna: Springer Vienna, 1990. p. 209–233.

[27] FRIEZE, A.; MCDIARMID, C. Algorithmic theory of random graphs. **Random Structures & Algorithms**, v. 10, n. 1-2, p. 5–42, 1997.

[28] BELLMAN, R. Dynamic Programming Treatment of the Travelling Salesman Problem. **J. ACM**, v. 9, n. 1, p. 61–63, jan. 1962.

[29] HELD, M.; KARP, R. M. **A dynamic programming approach to sequencing problems**. Proceedings of the 1961 16th ACM National Meeting. **Anais**...: ACM '61.New York, NY, USA: Association for Computing Machinery, 1961.

[30] BOLLOBÁS, B.; FENNER, T. I.; FRIEZE, A. M. An algorithm for finding hamilton paths and cycles in random graphs. **Combinatorica**, v. 7, n. 4, p. 327–341, 1987.

[31] PÓSA, L. Hamiltonian circuits in random graphs. **Discrete Mathematics**, v. 14, n. 4, p. 359–364, 1976.

[32] OSBORNE, M. J. **An Introduction to Game Theory**. [s.l.] Oxford University Press, 2004.

[33] NEUMANN, J. VON; MORGENSTERN, O. **Theory of Games and Economic Behavior**. Princeton: Princeton University Press, 1944.

[34] LOOMIS, L. H. On A Theorem of von Neumann. **Proceedings of the National Academy of Sciences**, v. 32, n. 8, p. 213–215, 1946.

[35] SPIELMAN, D. A.; TENG, S.-H. Smoothed analysis: an attempt to explain the behavior of algorithms in practice. **Commun. ACM**, v. 52, n. 10, p. 76–84, out. 2009.

[36] SPIELMAN, D.; TENG, S.-H. **Smoothed analysis of algorithms: why the simplex algorithm usually takes polynomial time**. Proceedings of the Thirty-Third Annual ACM Symposium on Theory of Computing. **Anais**...: STOC '01.New York, NY, USA: Association for Computing Machinery, 2001.

[37] ETSCHEID, M.; RÖGLIN, H. Smoothed Analysis of Local Search for the Maximum-Cut Problem. **ACM Trans. Algorithms**, v. 13, n. 2, mar. 2017.

[38] ENGLERT, M.; RÖGLIN, H.; VÖCKING, B. Smoothed Analysis of the 2-Opt Algorithm for the General TSP. v. 13, n. 1, set. 2016.

[39] SPIELMAN, D. A.; TENG, S.-H. **Smoothed Analysis**. (F. Dehne, J.-R. Sack, M. Smid, Eds.)Algorithms and Data Structures. **Anais**...Berlin, Heidelberg: Springer Berlin Heidelberg, 2003.

[40] ANASTOS, M.; KWAN, M.; MOORE, B. **Smoothed Analysis for Graph Isomorphism**. Proceedings of the 57th Annual ACM Symposium on Theory of Computing. **Anais**...: STOC '25.New York, NY, USA: Association for Computing Machinery, 2025.

[41] DINITZ, M. et al. Smoothed analysis of dynamic networks. **Distributed Computing**, v. 31, n. 4, p. 273–287, 2018.

[42] FLAXMAN, A. D.; FRIEZE, A. M. The diameter of randomly perturbed digraphs and some applications. **Random Structures & Algorithms**, v. 30, n. 4, p. 484–504, 2007.

[43] MANTHEY, B.; PLOCIENNIK, K. Approximating independent set in perturbed graphs. **Discrete Applied Mathematics**, v. 161, n. 12, p. 1761–1768, 2013.

[44] KRIVELEVICH, M.; SUDAKOV, B.; TETALI, P. On smoothed analysis in dense graphs and formulas. **Random Structures & Algorithms**, v. 29, n. 2, p. 180–193, 2006.

[45] BÖTTCHER, J. et al. Triangles in randomly perturbed graphs. **Combinatorics, Probability and Computing**, v. 32, n. 1, p. 91–121, 2023.

[46] KRIVELEVICH, M.; REICHMAN, D.; SAMOTIJ, W. Smoothed Analysis on Connected Graphs. **SIAM Journal on Discrete Mathematics**, v. 29, n. 3, p. 1654–1669, 2015.

[47] BABAI, L. **Graph Isomorphism in Quasipolynomial Time**., 2016. Disponível em: <https://arxiv.org/abs/1512.03547>

[48] BABAI, L.; ERDŐS, P.; SELKOW, S. M. Random Graph Isomorphism. **SIAM Journal on Computing**, v. 9, n. 3, p. 628–635, 1980.