Carlos Eduardo Meira Tavares

DESALEATORIZAÇÃO DE ALGORITMOS BPP

Curitiba - PR, Brasil

2008

Carlos Eduardo Meira Tavares

DESALEATORIZAÇÃO DE ALGORITMOS BPP

Monografia apresentada como parte dos requisitos para obtenção do Grau de Bacharel em Ciência da Computação pela Universidade Federal do Paraná.

Orientador:
Jair Donadelli Jr.

UNIVERSIDADE FEDERAL DO PARANÁ

Curitiba - PR, Brasil

2008

Agradecimentos

Agradeço primeiramente a Deus, por ter me dado força e não deixado desistir, à minha noiva, Valquiria pela ajuda com a revisão gramatical e compreensão nos momentos em que estava longe concentrado no trabalho. Aos meus pais, pelo grande apoio e incentivo que recebi. Ao meu orientador professor Jair Donadelli Jr. por ter acreditado em meu trabalho e ter dedicado seu tempo. Aos demais, sou grato por terem me apoiado com carinho e incentivando-me em meus objetivos. Cada palavra foi de extrema importância para meu desenvolvimento.

Sumário

1 Introdução p. 5

2 Computação Aleatorizada p. 7

2.1 Máquinas de Turing Aleatorizadas . . . . . p. 7

2.2 Classes de Complexidade Aleatorizadas . . . . . p. 8

2.2.1 As Classes *RP* e *coRP* . . . . . p. 8

2.2.2 O problema da Multiplicação de Matrizes . . . . . p. 10

2.2.3 A Classe *PP* . . . . . p. 13

2.2.4 A Classe *ZPP* . . . . . p. 13

2.2.5 O problema da Coloração de Conjuntos . . . . . p. 15

2.2.6 A Classe *BPP* . . . . . p. 17

2.2.7 Problema do Corte Mínimo . . . . . p. 21

3 Fontes Aleatórias p. 23

3.1 Indistinguibilidade Computacional . . . . . p. 24

3.2 Geradores Pseudoaleatórios . . . . . p. 24

3.2.1 Amplificando a função de expansão . . . . . p. 26

3.3 Funções Unidirecionais (*One-Way Functions*) . . . . . p. 26

3.4 A Relação de Funções unidirecionais com Geradores Pseudoaleatórios . . . . p. 27

4 Desaleatorização p. 29

4.1 Nova noção de gerador pseudoaleatório . . . . . p. 29

4.1.1 Desaleatorização usando gerador pseudoaleatório . . . . . p. 30

4.2 Desaleatorização usando grafos expansores . . . . . p. 32

4.2.1 Passeios Aleatórios em Grafos . . . . . p. 34

4.2.2 Amplificação de Probabilidade por Passeios Aleatórios em Expansores p. 34

5 Conclusão p. 37

Referências Bibliográficas p. 38

Apêndice A p. 39

A.1 Desigualdade de Markov . . . . . p. 39

A.2 Valor Esperado de uma variável aleatória . . . . . p. 39

A.3 Variáveis aleatórias de Bernoulli . . . . . p. 40

A.4 Conceitos de Algebra Linear . . . . . p. 41

A.4.1 Autovalores e Autovetores . . . . . p. 41

A.4.2 Norma de Vetores . . . . . p. 42

1 Introdução

"Não se vendem dois pardais por um asse? E nenhum deles cairá em terra sem o consentimento de vosso Pai."

Mateus 10:29

"E até os cabelos da vossa cabeça estão todos contados (...)"

Lucas 12:7

Em um algoritmo aleatorizado além da parte prática, há também uma parte filosófica associada.

A parte filosófica pode ser bem representada pelo texto do apóstolo Mateus. Segundo ele, Deus não permite que nenhum pardal caia sem o seu consentimento. Daí vem a pergunta: Existem eventos realmente aleatórios, ou o mundo é de fato determinístico? A parte prática tem a ver com a geração de bits aleatórios, assumindo que eles existam. Um algoritmo aleatório pode necessitar de bilhões de bits aleatórios a cada segundo. Mesmo se encontrarmos no mundo um pouco de aleatoriedade, esta pode não ser suficiente para prover a quantidade necessária de bits não relacionados a um algoritmo aleatório. Atualmente os ambientes computacionais recorrem a outros recursos, tais como, pegar o intervalo de tempo entre uma tecla pressionada no teclado e outra (medindo em microsegundos), aplicar um procedimento determinístico e tornar a sequência de bits longa e "aparentemente" aleatória.

Sabe-se que em geral a aleatoriedade nos ajuda a criar algoritmos mais eficientes. Contudo não temos fontes aleatórias sempre disponíveis, então gostaríamos de minimizar ao máximo o uso de aleatoriedade, fazendo uma simulação eficiente de algoritmos aleatórios, esse processo se chama desaleatorização e as ferramentas base para a derandomização são os geradores pseudoaleatórios. Uma área de grande interesse tanto na Ciência da Computação como na Matemática é o estudo de objetos pseudoaleatórios. Esses estão intimamente ligados com a noção de complexidade computacional.

Porém um problema não resolvido nesta área: se é válida a afirmação de que algoritmos baseados em sorteios são realmente mais poderosos que algoritmos determinísticos. Para muitos casos o que podemos afirmar é que algoritmos aleatorizados são os mais simples disponíveis, ou mais rápidos, ou ambos. Esta afirmação se baseia na construtibilidade eficiente de conjuntos pseudoaleatórios pequenos que possam simular grandes conjuntos evitando assim o uso de uma quantidade muito grande de aleatoriedade. Esta simulação se chama desaleatorização.

O interesse deste trabalho é promover o estudo das classes de complexidades aleatórias e investigar algumas ferramentas, conceitos necessários e técnicas para entender o funcionamento da desaleatorização, focando principalmente na classe de algoritmos *BPP*.

2 *Computação Aleatorizada*

Para compreender as classes de complexidade aleatórias, é necessário introduzir uma nova noção de máquina de Turing, adicionando a ela a capacidade de "jogar moedas", isto é, tomar decisões aleatórias baseadas em "lançamentos de moeda".

2.1 Máquinas de Turing Aleatorizadas

A idéia fundamental deste estudo é a associação de *computação eficiente* com máquinas de Turing aleatorizadas de tempo polinomial. Consideraremos *eficiente* somente máquinas que executam em um tempo que é no máximo polinomial do tamanho da entrada.

Um problema de decisão é uma questão sobre um sistema formal com uma resposta do tipo *sim* ou *não*, ou seja, um problema de determinar se um determinado elemento de algum universo pertence ou não a um determinado conjunto (ou equivalentemente, se satisfaz uma determinada propriedade). Por exemplo, dada uma *string x* e uma linguagem $L, L \subseteq \{0,1\}^*$, a pergunta, $x \in L$ ou $x \notin L$ é um problema de decisão.

Uma máquina de Turing não-determinística é uma máquina que admite mais de uma transição partindo de um estado sob um certo símbolo. Assim, podem existir várias computações possíveis para o processamento de uma *string*. Uma *string* é aceita quando existe uma computação para a qual a máquina pára em um estado final. Se existir tal máquina que receba como entrada uma *string x* e retorne como saída *sim*, caso *x* pertença a linguagem L, ou *não*, caso contrário, então diz-se que o problema de decisão para a linguagem *L* é decidível. Do contrário, se não existir tal máquina capaz de fazer essa avaliação, então diz-se que o problema de decisão para a linguagem *L* é indecidível.

Na máquina de Turing aleatorizada consideramos também a probabilidade de aceitação ao invés de somente perguntar se a máquina tem uma computação de aceitação [1]. Informalmente, uma máquina de Turing aleatorizada é uma máquina de Turing não-determinística onde a cada decisão não-determinística é definida uma probabilidade. Sem perda de generalidade

podemos assumir que em cada passo a máquina tem exatamente duas escolhas, cada uma delas com probabilidade $\frac{1}{2}$. Todas as computações possíveis, ou "ramos" da árvore de computação são eventos equiprováveis, e para uma computação de tamanho k, cada uma ocorre com probabilidade 2^{-k} [6]. A soma das probabilidades das computações da máquina M com entrada $x \in \{0,1\}^*$ e que resulta em $y \in \{0,1\}$ será denotada por $Prob[M(x) = y]$.

2.2 Classes de Complexidade Aleatorizadas

2.2.1 As Classes *RP* e *coRP*

As primeiras duas classes que surgem da computação aleatorizada são as linguagens computáveis em tempo polinomial com erro unilateral. Se existe uma máquina que pode decidir uma linguagem com alta probabilidade em tempo polinomial, é razoável considerar que o problema é relativamente fácil. Alta probabilidade, neste contexto, significa que a máquina tem certeza somente em um caso e retorna a reposta correta no outro caso somente com boa probabilidade (os casos são quando $x \in L$ e quando $x \notin L$). Não pode haver respostas que sejam falsos positivos, desde então, para $x \notin L$, a rejeição é certa. Também a probabilidade de falsos negativos é no máximo $\frac{1}{2}$. A razão é por que, se em cada passo da máquina está envolvido uma escolha aleatória entre duas alternativas, com probabilidade $\frac{1}{2}$ para cada, então todas as computações são eventos igualmente prováveis, cada um com probabilidade $2^{-p(|x|)}$ dado que todas as computações tem tamanho $p(|x|)$.[1]

Daqui em diante, uma máquina de Turing aleatória de tempo polinomial significa uma máquina que sempre pára após um número de passos polinomial no tamanho da entrada.

Definição 2.1 (Random Polynomial-time - *RP*[1, 2])**.** *A classe de complexidade RP é a classe de todas as linguagens L, para as quais existe uma máquina de Turing aleatória M de tempo polinomial, tal que*

$$x \in L \Rightarrow Prob[M(x) = 1] \geq \tfrac{1}{2}$$

$$x \notin L \Rightarrow Prob[M(x) = 1] = 0\,.$$

Definição 2.2 (Complementary Random Polynomial-time - *coRP*[1, 2])**.** *A classe de complexidade coRP é a classe de todas as linguagens, para as quais existe uma máquina de Turing aleatória M de tempo polinomial, tal que*

[1] $p(|x|)$ onde p é um polinômio e $|x|$ é o número de símbolos de x.

$$x \in L \Rightarrow Prob[M(x) = 1] = 1$$

$$x \notin L \Rightarrow Prob[M(x) = 0] \geq \frac{1}{2}$$

A constante $\frac{1}{2}$ na definição de RP é arbitrária. Podemos substituí-la por qualquer outra estritamente entre 0 e 1, e obter a mesma classe de complexidade.

Mostraremos que substituindo a constante $\frac{1}{2}$ por $\frac{1}{p(|x|)}$ ou $1 - 2^{-p(|x|)}$, onde p é qualquer polinomio fixo, obteremos a mesma classe de linguagens. Considere as seguintes definições:

Definição 2.3. *A classe de complexidade $RP1$ é classe a de todas as linguagens L, para as quais existe uma máquina de Turing aleatorizada M de tempo polinomial, e um polinomio $p(.)$, tal que*

$$x \in L \Rightarrow Prob[M(x) = 1] \geq \frac{1}{p(|x|)}$$

$$x \notin L \Rightarrow Prob[M(x) = 0] = 1.$$

Definição 2.4. *A classe de complexidade $RP2$ é a classe de todas as linguagens L, para as quais existe uma máquina de Turing aleatorizada M de tempo polinomial, e um polinomio $p(.)$, tal que*

$$x \in L \Rightarrow Prob[M(x) = 1] \geq 1 - 2^{-p(|x|)}$$

$$x \notin L \Rightarrow Prob[M(x) = 0] = 1$$

Estas definições parecem bastante diferentes das anteriores, pois em $RP1$ temos um *algoritmo probabilístico* que responde corretamente com uma probabilidade muito pequena, enquanto em $RP2$ temos um *algoritmo eficiente* que quase ignora a probabilidade de errar. Contudo, estas duas definições definem a mesma classe de linguagens como provaremos a seguir. Isto implica que a existência de um algoritmo com uma probabilidade de sucesso considerável implica a existência de um *algoritmo eficiente* com uma probabilidade de erro desprezível.

PROPOSIÇÃO 2.5:
$RP1 = RP2$.

Demonstração. Vamos mostrar $RP1 \supseteq RP2$. Esta direção é trivial pois se $|x|$ é grande o bastante então o limite na definição de $RP1$ é menor que o limite na definição de $RP2$. Assim, estar em $RP2$ implica estar em $RP2$ para quase todas as entradas. As entradas para as quais isto não ocorre podem ser incorporadas na máquina $RP1$. Assim $RP1 \supseteq RP2$.

Vamos mostrar $RP1 \subseteq RP2$. Usaremos um método chamado *amplificação*. Repetimos a execução da máquina $RP1$ muitas vezes para que a probabilidade dela retornar uma resposta incorreta seja bastante pequena. Assumindo a existência de uma máquina M_1 tal que:

$$\forall x \in L : Prob[M_1(x) = 1] \geq \frac{1}{p(|x|)}.$$

Definiremos uma nova máquina M_2, dada uma função $t(|x|)$, do seguinte modo:

$$M_2(x) = \begin{cases} \text{Executar } M_1(x)\ t(|x|) \text{ vezes} \\ \text{Se alguma chamada retornar 1, então retorne 1} \\ \text{Senão retorne 0,} \end{cases}$$

tome $t = t(|x|)$. Então para todo $x \in L$

$$Prob[M_2(x) = 0] = (Prob[M_1(x) = 0])^{t(|x|)} \leq (1 - \frac{1}{p(|x|)})^{t(|x|)}$$

Para encontrar então o $t(|x|)$ desejado resolvemos a seguinte equação:

$$(1 - \frac{1}{p(|x|)})^{t(|x|)} \leq 2^{-p(|x|)},$$

e obtemos:

$$t(|x|) \geq p(|x|) \log_2 (1 - \frac{1}{p(|x|)})^{-1} = \frac{p(|x|)^2}{\log_2 e},$$

onde $e \approx 2.7182818\ldots$ é a base do logaritmo natural.

Assim tomando $t(|x|) = p(|x|)^2$ na definição de M_2 temos uma máquina que executa em tempo polinomial e decide L com probabilidade maior que $1 - 2^{-p(|x|)}$ para obter a resposta correta para $x \in L$ e sempre correta quando $x \notin L$. □

Veremos agora um exemplo de algoritmo da classe RP.

2.2.2 O problema da Multiplicação de Matrizes

Suponha que A, B e C sejam 3 matrizes de ordem $n \times n$. No seguinte problema de decisão associado à multiplicação de matrizes, deseja-se decidir simplesmente se o produto de A por B é igual a C. O melhor algoritmo determinístico para esse problema, multiplica A por B utilizando o algoritmo de Coppersmith e Winograd (de complexidade $O(n^{2.376})$) e compara o

resultado obtido com C. O algoritmo de Coppersmith e Winograd é uma melhoria do algoritmo de Strassen que tem complexidade $O(n^{2.807}))$.

O algoritmo aleatorizado apresentado a seguir, é devido a Freivalds e tem um tempo de processamento da ordem de $O(\log(\frac{1}{\phi})n^2))$, onde $\phi > 0$, representa a probabilidade de falha definida a priori. Como será visto mais adiante, a constante k presente no algoritmo, representa o número total de repetições até que se tenha probabilidade de falha inferior a ϕ.

Dados : Matrizes A, B, C e constante $\varepsilon > 0$
Devolve: $AB = C$
continua $\leftarrow$ true;
$k \leftarrow 1$;
while $k \neq \log(\lceil \frac{1}{\phi} \rceil)$ *e continua* $\neq$ *false* **do**
    escolhe $r \in \{0,1\}^n$ aleatoriamente;
    if $ABr \neq Cr$ **then**
        continua $\leftarrow$ false;
    end
    if $continua = true$ **then**
        return $AB = C$
    else
        return $AB \neq C$
    end
end

Algoritmo 1: Multiplicação de matrizes (A, B, C, ε)

Observe que a comparação de $ABr = Cr$ pode ser implementada em tempo $O(n^2)$ computando-se inicialmente Br seguido de $A(Br)$ e Cr respectivamente.

Uma vez definidos A, B e C na entrada de dados note que, se $AB = C$ então $ABr = Cr$, $\forall r \in \{0,1\}^n$ ou seja, $Prob[ABr = Cr] = 1$. Nesta situação o algoritmo de Freivalds nunca retorna uma resposta incorreta. O que ocorre, entretanto, se $AB \neq C$? Neste caso, se para algum $r \in \{0,1\}^n$ a igualdade $ABr = Cr$ se verifica, isto não implicará necessariamente que $AB = C$.

Para ilustrar essa situação considere o exemplo seguinte onde são dadas 3 matrizes A, B e C de ordem 2×2 e o vetor $r = e$, (onde e é um vetor com todas as coordenadas iguais a 1):

$$A = \begin{pmatrix} 2 & 4 \\ 1 & 3 \end{pmatrix}; B = \begin{pmatrix} 3 & 4 \\ 1 & 1 \end{pmatrix}; C = \begin{pmatrix} 11 & 11 \\ 7 & 6 \end{pmatrix}; r = e = \begin{pmatrix} 1 \\ 1 \end{pmatrix};$$

Calculando $A(Be)$ e Ce separadamente conclui-se diretamente que $ABe = Ce$. Entretanto,

$AB \neq C$.

O resultado seguinte garante que, se $AB \neq C$ então $Prob[ABr = Cr] \leq \frac{1}{2}$ para r selecionado aleatoriamente. Isto é equivalente a afirmar que a probabilidade de um falso negativo em uma única repetição do algoritmo é menor ou igual a $\frac{1}{2}$. A probabilidade de erro é minimizada aumentando-se o número de repetições.

PROPOSIÇÃO 2.6:
Se $r \in \{0,1\}^n$ é escolhido aleatoriamente com distribuição uniforme e $AB \neq C$ então $Prob[ABr = Cr] \leq \frac{1}{2}$.

Demonstração. Seja $D = AB - C$. Como $AB \neq C$ (hipótese) então $D \neq 0$. Assim, existirão índices $i, j \in \{1, \ldots, n\}$ tais que $d_{ij} \neq 0$ (onde $d_{ij} \in D$). Além disso, $Prob[ABr = Cr] = Prob[Dr = 0]$. Se $d_i = (d_{i1}, \ldots, d_{ij}, \ldots, d_{in})$ representa a i-ésima linha de D então:

$$Prob[Dr = 0] \leq Prob[\textstyle\sum_{k=1}^{n} d_{ik} r_k = 0], \text{ para algum } i \in \{1, \ldots, n\} \text{ e } d_{ij} \neq 0$$

Isolando-se r_j no segundo membro da desigualdade (já que $d_{ij} \neq 0$) conclui-se diretamente que:

$$Prob[\textstyle\sum_{k=1}^{n} d_{ik} r_k = 0] = Prob[r_j = v], \text{ onde } v = -(\textstyle\sum_{k \neq j} d_{ik} r_k)/d_{ij}$$

Assumindo-se que todos os demais elementos r_k tal que $(k \neq j)$ sejam escolhidos anteriormente a r_k, tem-se que a probabilidade de r_j ser igual a um valor fixo v será menor ou igual a $\frac{1}{2}$. Lembre-se que $r_j \in \{0,1\}$ é escolhido aleatoriamente satisfazendo uma distribuição uniforme. Portanto, $Prob[Dr = 0] \leq \frac{1}{2}$.

□

Como consequência deste resultado, se $AB \neq C$ então $Prob[ABr \neq Cr] \leq \frac{1}{2}$. Contudo, se $ABr \neq Cr$ para algum $r \in \{0,1\}^n$ escolhido arbitrariamente, então $(AB - C)r \neq 0$ e portanto $AB \neq C$ (pois $r \neq 0$). Logo, a cada iteração, o algoritmo retorna uma resposta correta com probabilidade de acerto superior a $\frac{1}{2}$. A probabilidade de fracasso pode ser minimizada repetindo-se o processo um número pré-determinado de vezes. Se k repetições forem realizadas então a probabilidade de falha ϕ no procedimento será $\leq \frac{1}{2^k}$. Se, em algumas destas iterações ocorrer $ABr \neq Cr$ para algum $r \in \{0,1\}^n$, o processo é interrompido imediatamente retornando $AB \neq C$.

Como discutido anteriormente, o algoritmo de Freivalds tem complexidade polinomial e igual a $O(\log(\frac{1}{\phi})n^2)$ para $\phi > 0$, selecionado a priori. Note que, uma vez escolhido ϕ , o valor

$\log(\frac{1}{\phi})$ se torna constante. A determinação de um algoritmo determinístico $O(n^2)$ para este problema continua um problema em aberto.

2.2.3 A Classe *PP*

Uma linguagem L está na classe *PP* se houver uma máquina de Turing aleatorizada M de tempo polinomial, tal que para toda entrada x, $x \in L$ se e somente se mais que a metade das computações de M para a entrada x aceitam, M decide L "pela maioria". [1, 2]

Definição 2.7 (Probabilistic Polynomial-time)**.** *A classe de complexidade PP é a classe de todas as linguagens, para as quais existe uma máquina de Turing aleatorizada M de tempo polinomial, tal que*

$$x \in L \Rightarrow Prob[M(x) = 1] > \tfrac{1}{2}$$
$$x \notin L \Rightarrow Prob[M(x) = 0] > \tfrac{1}{2}.$$

2.2.4 A Classe *ZPP*

Agora será interessante examinar as propriedades de $RP \bigcap coRP$. Parece que problemas que estão em $RP \bigcap coRP$ podem se beneficiar do resultado exato da máquina de Turing *RP* (quando $x \notin L$) e da máquina de Turing *coRP* (quando $x \in L$).

Outro ponto interessante é permitir que a máquina responda "Não sei" para algumas entradas. Será mostrado que estas duas idéias se concentram na mesma classe de linguagens, isto é, máquinas que podem responder "Não sei" ou de outro modo responder corretamente.

Definição 2.8 (Zero-error Probabilistic Polynomial-time[1, 2])**.** *A classe de complexidade ZPP é a classe de todas as linguagens, para as quais existe uma máquina de Turing aleatorizada M de tempo polinomial, tal que*

$$\forall x \in \{0,1\}^*, Prob[M(x) = \perp] \leq \tfrac{1}{2}$$
$$\forall x \in \{0,1\}^*, Prob[M(x) = \mathcal{X}_{L(x)} \text{ ou } M(x) = \perp] = 1,$$

onde a resposta "Não sei" é denotada por $\perp$ *e* $\mathcal{X}_{L(x)}$ *denota*

$$\mathcal{X}_{L(x)} = \begin{cases} 1, se\ x \in L \\ 0, se\ x \notin L. \end{cases}$$

PROPOSIÇÃO 2.9:
$ZPP = RP \bigcap coRP$.

Demonstração. Tome $L \in ZPP$. Considere M a máquina definida anteriormente. Construiremos uma nova máquina M' que decide L de acordo com a definição de *RP*, (isto implica que $ZPP \subseteq RP$).

Considere o seguinte algoritmo:

Dados : *string* $x \in \{0,1\}^*$
Devolve: $M'(x)$
$b \leftarrow M(x)$;
if $(b == \perp)$ **then**
    return 0;
else
    return b;
end

Algoritmo 2: M'(x)

Assim, se $x \notin L$ e retornando 0 quando $M(x) = \perp$ sempre responderá corretamente, pois neste caso $M(x) \neq \perp$ implica em $M'(x) = \mathcal{X}_{L(x)}$ implica em $M'(x) = 0$.

Se $x \in L$, a probabilidade de obter a resposta correta com M' é maior que $\frac{1}{2}$, porque M retorna uma resposta definida $(M(x) \neq \perp)$ com probabilidade maior que $\frac{1}{2}$ e respostas definidas de M são sempre corretas, pois retorna $\perp$ quando está incerto.

Do mesmo modo podemos ver que $ZPP \subseteq coRP$ (a máquina que será construída retorna "sim" quando está incerta). Obtemos $ZPP \subseteq RP \bigcap coRP$.

Assuma $L \in RP \bigcap coRP$. Tome M_{RP} a máquina *RP* e M_{coRP} a máquina *coRP* que decide L de acordo com as definições 1 e 2, respectivamente. Redefinimos $M'(x)$ usando M_{RP} e M_{coRP} a seguir:

Dados : *string* $x \in \{0,1\}^*$

Devolve: $M'(x)$

if $(M_{RP}(x) == 1)$ **then**
    return 1;

end

if $(M_{coRP}(x) == 0)$ **then**
    return 0;

end

return $\perp$;

Algoritmo 3: M'(x)

Se M_{RP} responder "sim", então pela definição 1 garantimos que $x \in L$. Similarmente, se M_{coRP} responder "não", então pela definição 2 garantimos que $x \notin L$. Assim nunca obtemos uma resposta errada. Nos outros casos poderemos ter certeza que M' retorna uma resposta definida e correta com probabilidade maior que $\frac{1}{2}$. □

A conclusão é esta: M' é de fato uma máquina *ZPP*, assim, $RP \bigcap coRP \subseteq ZPP$ e, junto com a parte anterior, concluímos que $RP \bigcap coRP = ZPP$.

Veremos agora um exemplo de algoritmo da classe *ZPP*.

2.2.5 O problema da Coloração de Conjuntos

Suponha $S = \{x_1, x_2, \ldots, x_n\}$ um conjunto de n elementos quaisquer. Seja $S_1, S_2, \ldots, S_k$ uma coleção de subconjuntos distintos de S cada um com $r \geq 2$ elementos e tal que $k \leq 2^{r-2}$. Os elementos de S deverão ser pintados com branco ou preto de forma que cada um dos subconjuntos S_i para $i = 1, \ldots, k$ contenha pelo menos dois elementos com cores distintas.

Um algoritmo aleatorizado para este problema pode ser obtido, simplesmente, colorindo-se cada elemento de S aleatoriamente com probabilidade $\frac{1}{2}$. Caso a coloração resultante contenha um ou mais subconjuntos com elementos de mesma cor, repete-se o processo até que uma coloração viável seja obtida. O seguinte algoritmo sintetiza as principais etapas do problema da coloração de conjuntos. A variável *viavel* (presente no algoritmo) indica se uma coloração desejada foi ou não encontrada:

Devolve: *viavel*
Dados $(S_1, S_2, \ldots, S_k)$
$viavel \leftarrow false$;
while $viavel \neq true$ **do**
  for $i \leftarrow 1$ **to** n **do**
    pinte x_i com cor $\xi_r\{branco, preto\}$, $\xi_r \equiv$ Escolha Aleatoria
  end
  viavel $\leftarrow$ checa_viabilidade$(x, S_1, \ldots, S_k)$
end

Algoritmo 4: Coloração de conjuntos

Para mostrar que o Algoritmo acima sempre encontra uma solução desejada deve-se mostrar inicialmente que uma coloração válida é obtida com probabilidade de sucesso p estritamente positiva. Para comprovar este fato, considere Y uma variável aleatória indicando o número total de iterações até que uma configuração viável seja obtida. Observe que Y define uma distribuição geométrica com parâmetro $p > 0$ e valor esperado $\varepsilon(Y)$ igual a $\frac{1}{p}$. Isto garante um número finito de passos mesmo que a probabilidade de sucesso em uma dada iteração seja bastante pequena.

Suponha que um evento A ocorra sempre que uma coloração viável for obtida. Caso contrário, a ocorrência do evento complementar $\overline{A}$ irá indicar que pelo menos um dos subconjuntos S_i para $i \in \{1, \ldots, k\}$ foi colorido com mesma cor (branco ou preto). Analogamente, sejam B_i para $i = 1, \ldots, k$, eventos indicando que os elementos de S_i foram pintados com cores distintas. Logo, os eventos complementares $\overline{B_i}$ irão indicar que apenas uma cor foi utilizada na coloração de S_i.

Como $|S_i| = r$ para $i = 1, \ldots, k$, tem-se que $Prob[\overline{B_i}] = \frac{2}{2^r}$. Note que cada S_i pode ser colorido só com branco ou só com preto. A probabilidade de fracasso $Prob[\overline{A}]$ será dada por:

$$Prob[\overline{A}] = Prob[\bigcup_{i=1}^{k} \overline{B_i}] \leq \sum_{i=1}^{k} Prob[\overline{B_i}] = \frac{2k}{2^r}.$$

Como $k \leq 2^{r-2}$, conclui-se diretamente que $Prob[\overline{A}] \leq \frac{1}{2}$. Isto garante que a probabilidade de sucesso em uma iteração do algoritmo aleatório será superior ou igual a $\frac{1}{2}$, ou seja, $Prob[A] \geq \frac{1}{2}$. Repete-se o processo sempre que alguma falha tenha sido detectada pelo procedimento Checa-viabilidade.

Observe neste exemplo que, fazendo a probabilidade de sucesso igual a $\frac{1}{2}$, o número esperado de iterações $\varepsilon(Y)$ para obtenção de uma coloração válida será apenas 2. Informalmente falando, é provável que, em aproximadamente 2 repetições do algoritmo, já se tenha uma coloração viável desejada.

Se a probabilidade de sucesso for muito pequena, ter-se-á provavelmente, um longo período de processamento. O algoritmo será eficiente se o valor esperado para o tempo de processamento for polinomial no tamanho da entrada. No problema da coloração de conjuntos serão necessárias $O(n)$ passos para coloração completa de S e $O(kr)$ passos, no pior caso, para checar se a coloração é ou não viável (procedimento checa-viabilidade). Note que, por hipótese, os k subconjuntos de S (de tamanho r) são distintos entre si, logo $k \leq \frac{n!}{r!(n-r)!}$. Como $\varepsilon(Y) = 2$, o tempo esperado de processamento será polinomial e igual a $O(2(n+kr))$. O procedimento será polinomial no tamanho da entrada.

2.2.6 A Classe *BPP*

A classe *BPP* é uma classe de problemas bastante prática [5, 1, 2]. A maioria dos problemas de interesse em *BPP* têm algoritmos probabilísticos bastante *eficientes*. Por este motivo, é de grande interesse que problemas e classes de problemas estejam em *BPP*. A máquina *BPP* reconhece uma linguagem com alta probabilidade independentemente se $x \in L$ ou $x \notin L$.

Definição 2.10 (Bounded-error Probabilistic Polynomial-time - *BPP*)**.** *A classe de complexidade BPP é a classe de todas as linguagens L, para as quais existe uma máquina de Turing aleatorizada M de tempo polinomial, tal que*

$$x \in L \Rightarrow Prob[M(x) = 1] \geq \tfrac{2}{3},$$

$$x \notin L \Rightarrow Prob[M(x) = 1] < \tfrac{1}{3}.$$

A máquina *BPP* é uma máquina que comete erros nas duas respostas, porém, executando mais de uma vez, retorna a resposta correta a maior parte do tempo. Imagine a seguinte situação: temos uma moeda viciada, isto é, ao fazer um lançamento, um de seus lados tem maior probabilidade de aparecer que o outro. Sabemos que um lado tem probabilidade ϕ, para algum $\phi > 0$, e o outro $1-\phi$, porém não sabemos qual é qual. Um experimento óbvio para descobrir qual lado da moeda é viciado seria lançá-la muitas vezes, e tomar o lado que aparecer mais com probabilidade ϕ. Contudo, quantas vezes é preciso lançar a moeda para ser capaz de descobrir corretamente com uma razoavel alta probabilidade? O seguinte Lema nos mostrará:

LEMA 2.11 (Desigualdade de Chernoff [4, 5]):
Suponha que $X_1, X_2, \ldots, X_n$ são variáveis aleatórias de Bernoulli que assumem os valores 1 *e* 0 *com probabilidades p e $1-p$, respectivamente, e considere a soma $X = \sum_{i=1}^{n} X_i$. Então, para todo $0 \leq \theta \leq 1$,*

$$Prob[X \geq (1+\theta)pn] \leq [\frac{e^{\theta}}{(1+\theta)^{(1+\theta)}}]^{pn}, \text{ onde } pn = \sum_{i=1}^{n} p_i = \varepsilon(X) \quad (1)$$

Demonstração. Se t é qualquer número real positivo temos trivialmente que

$$Prob[X \geq (1+\theta)pn] = Prob[e^{tX} \geq e^{t(1+\theta)pn}]$$

Aplicando a desigualdade de Markov[2] no lado direito, obtemos,

$$Prob[X \geq (1+\theta)pn] \leq \frac{\varepsilon(e^{tx})}{e^{t(1+\theta)pn}} \quad (2)$$

Observe que: $\varepsilon(e^{tx}) = \varepsilon(e^{t\sum_{i=1}^{n} x_i}) = \varepsilon(\prod_{i=1}^{n} e^{tx_i})$.

Desde que x_i seja independente, as variáveis aleatórias e^{tx_i} também serão, segue que:

$$\varepsilon(\prod_{i=1}^{n} e^{tx_i}) = \prod_{i=1}^{n} \varepsilon(e^{tx_i}).$$

Usando isto em (2), temos que:

$$Prob[X \geq (1+\theta)pn] \leq \frac{\prod_{i=1}^{n} \varepsilon(e^{tx_i})}{e^{t(1+\theta)pn}}.$$

A variável aleatória e^{tx_i} assume o valor e^t com probabilidade p_i e o valor (1) com probabilidade $1-p_i$. Computando $\varepsilon(e^{tx_i})$, destas obervações temos que:

$$Prob[X \geq (1+\theta)pn] \leq \frac{\prod_{i=1}^{n}((1-p_i)e^0+p_ie^t)}{e^{t(1+\theta)pn}} = \frac{\prod_{i=1}^{n}(1+p_i(e^t-1))}{e^{t(1+\theta)pn}}.$$

Agora usaremos a desigualdade $1+x < e^x$ com $x = p_i(e^t-1)$, para obtermos:

$$\begin{aligned} Prob[X \geq (1+\theta)pn] &\leq \frac{\prod_{i=1}^{n} e^{p_i(e^t-1)}}{e^{t(1-\theta)pn}} \\ &= \frac{e^{\sum_{i=1}^{n} p_i(e^t-1)}}{e^{t(1+\theta)pn}} \\ &= \frac{e^{(e^t-1)\sum_{i=1}^{n} p_i}}{e^{t(1+\theta)pn}} \\ &= \frac{e^{(e^t-1)pn}}{e^{t(1+\theta)pn}} \\ &= \frac{e^{(e^t-1)}\,^{pn}}{e^{t(1+\theta)}} \\ &= \frac{e^{\theta}}{(1+\theta)^{(1+\theta)}}\,^{pn}, \end{aligned}$$

se considerarmos $t = \ln(1+\theta)$. □

[2]Veja Apêndice.

Este resultado pode ser especializado como segue.

Corolário 2.12. *Se* $p = \frac{1}{2} + \varepsilon$ *para algum* $\varepsilon > 0$*, então a probabilidade de* $\sum_{i=1}^{n} x_i \leq \frac{n}{2}$ *é no máximo* $e^{-\frac{\varepsilon^2 n}{6}}$.

Demonstração. Tome: $\theta = \frac{\varepsilon}{\frac{1}{2}+\varepsilon}$.

Assim podemos detectar em nossa moeda um vício ε com razoável certeza tomando a maioria de $\frac{1}{\varepsilon^2}$ experimentos. □

A classe *BPP* contém todas as linguagens L, para as quais existe uma máquina de Turing aleatorizada M de tempo polinomial, tal que:

Para todas as entradas x, se $x \in L$ então pelo menos $\frac{2}{3}$ das computações de $M(x)$ respondem "sim"; e se $x \notin L$ então pelo menos $\frac{2}{3}$ delas respondem "não".

Assim sendo, precisamos que M aceite pela "maioria", ou rejeite pela "maioria" dentre várias rodadas. O $\frac{2}{3}$ da definição é uma constante arbitrária. Substituindo o $\frac{2}{3}$ da definição por qualquer outra estritamente entre $\frac{1}{2}$ e 1 resulta na mesma classe. Se, por exemplo tivermos uma máquina M, que reconhece alguma linguagem L com probabilidade $p > \frac{1}{2}$, podemos facilmente construir uma máquina com uma probabilidade qualquer $q > p$ e executando esta máquina um número polinomial de vezes e retornando a maioria das respostas. Isto claramente incrementa a probabilidade de obtermos a resposta correta dentro do limiar buscado, além de executar em tempo polinomial. A notação *BPP* significa "tempo polinomial probabilístico com erro-limitado", devido ao fato de a classe *PP* ("tempo polinomial probabilístico") referir-se à definição em que as entradas na linguagem são aceitas com probabilidade maior que $\frac{1}{2}$ e as entradas que não são da linguagem são aceitas com probabildade no máximo $\frac{1}{2}$. Apesar deste nome, a classe *PP* não é um bom modelo de máquina aleatorizada, porque gasta um número de repetições exponencial para reduzir a probabilidade de erro. *BPP* é considerada a classe padrão associada com máquinas de Turing aleatorizadas de tempo polinomial.

Agora podemos reescrever a definição de *BPP* como:

$$x \in L \Rightarrow Prob[M(x) = 1] \geq \frac{1}{2} + \frac{1}{6}$$
$$x \notin L \Rightarrow Prob[M(x) = 1] < \frac{1}{2} - \frac{1}{6},$$

Poderíamos considerar a seguinte mudança nas constantes:

$$x \in L \Rightarrow Prob[M(x) = 1] \geq p + \varepsilon$$

$$x \notin L \Rightarrow Prob[M(x) = 1] < p - \varepsilon,$$

Para constantes $p \in (0,1)$ e $0 < \varepsilon < \min\{p, 1-p\}$.

Porém, é possível fazer mais que isto e usar um limiar que depende do tamanho da entrada, como mostra-se a seguir.

Usando o quadro acima veremos que para qualquer função computável em tempo polinomial e qualquer margem perceptível (representada pelo inverso de um polinômio) é possível recuperar o limiar padrão – de $\frac{1}{2}$ – e a margem "segura" – de $\frac{1}{6}$.

Definição 2.13. *$L \in BPP$ se e somente se existe uma função computável em tempo polinomial $f : N \to [0,1]$, um polinômio positivo $p(.)$ e uma máquina de Turing aleatorizada M de tempo polinomial, tal que:*

$$\forall x \in L, Prob[M(x) = 1] \geq f(|x|) + \frac{1}{p(|x|)}$$
$$\forall x \notin L, Prob[M(x) = 1] < f(|x|) - \frac{1}{p(|x|)}.$$

Demonstração. É fácil vermos que escolhendo $f(|x|) \equiv \frac{1}{2}$ e $p(|x|) \equiv 6$ obtemos a definição original do *BPP*, disto temos que qualquer linguagem *BPP* satizfaz a condição acima.

Assuma que temos uma máquina de Turing aleatória M de tempo polinomial com esses limites na probabilidade de obter uma resposta "sim". Então para qualquer entrada x, olhamos para a variável aleatória $M(x)$, que é uma variável aleatória de Bernoulli, com parâmetro desconhecido $p = \varepsilon[M(x)]$. Aproveitando o fato de a esperança de uma variável de Bernoulli ser exatamente a probabilidade de se obter "sim", temos que $p = Prob[M(x) = \text{"sim"}]$. Assim, estimando p, podemos concluir algo sobre se $x \in L$ ou $x \notin L$. A estimativa mais natural é pegar a média de n amostras da variável aleatória, isto é, as respostas de n chamadas independentes de $M(x)$.

Definimos a seguinte máquina de Turing.

$$M'(x) = \begin{cases} \text{Executar } M(x) \text{ } n \text{ vezes (chamar o resultado da i-ésima execução } t_i) \\ \text{Computar } p'' \leftarrow \frac{1}{n} . \sum_{i=1}^{n} t_i \\ \text{Se } p'' > f(x) \text{ , responda "sim", senão responda "não"} \end{cases}$$

Note que p'' é exatamente igual a média de uma amostra de tamanho n obtida da variável aleatória $M(x)$. Se pudermos mostrar que com um n apropriado a estimativa não cairá longe do

valor real com boa probabilidade, segue que a máquina responde corretamente com a mesma probabilidade.

Para escolher n usaremos a desigualdade de Chernoff, esta afirma que, para qualquer conjunto de variáveis aleatórias de Bernoulli $\{X_1, X_2, \ldots, X_n\}$ com as mesmas esperanças $p \leq \frac{1}{2}$ e para todo $\delta, 0 < \delta \leq p(p-1)$, temos:

$$Prob[|\frac{\sum_{i=1}^{n} X_i}{n} - p| > \delta] < 2e^{\frac{-\delta^2}{2p(1-p)}n}$$
$$\leq 2e^{\frac{-\delta^2}{2\frac{1}{4}}n} = 2e^{-2n\delta^2}.$$

Tomando $\delta = \frac{1}{p(|x|)}$ e $n = \frac{-\ln\frac{1}{6}}{2\delta^2}$ temos que a máquina de Turing M' decide $L(M)$ com probabilidade maior que $\frac{2}{3}$ sugerindo que $L \in BPP$.

□

Veremos agora um exemplo de algoritmo da classe *BPP*.

2.2.7 Problema do Corte Mínimo

Seja G um grafo conexo, não dirigido. Um corte em G é um conjunto de arestas cuja remoção resulta em G ser quebrado em dois ou mais componentes. O corte mínimo em G, é um corte de cardinalidade mínima..

Seja G um grafo qualquer com peso positivo nas arestas, e k, um inteiro positivo. Deseja-se responder se o Corte Mínimo em G tem valor *min_corte* exatamente igual a k. Seja w, uma solução do problema do Corte Mínimo gerada após sucessivas repetições do algoritmo com probabilidade de sucesso maior ou igual a $\frac{2}{3}$.

Considere agora o seguinte algoritmo:

Dados : Grafo G com peso positivo nas arestas e $k \in Z_+$
Devolve: {verdadeiro, falso}
$w \leftarrow$ Valor aleatório do Corte Mínimo gerado após sucessivas repetições;
if $k = w$ **then**
    return $min_corte = k$
else
    return $min_corte \neq k$
end

Algoritmo 5: Corte-Mínimo

Observe neste caso que, se $x \in L$, ou seja, se o corte mínimo *min_corte* é de fato igual a k então o Algoritmo acima aceita x com probabilidade maior ou igual a $\frac{2}{3}$ ($Prob[k = w] \geq \frac{2}{3}$). Por outro lado, se $x \notin L$, ou seja, se $min_corte < k$ então o Algoritmo rejeita x com probabilidade maior ou igual a $\frac{2}{3}$. Assim, $Prob[k \neq w] \geq \frac{2}{3}$. Se $min_corte > k$, o algoritmo sempre rejeita x, já que a solução w nunca será inferior a k. Logo, da definição de *BPP* conclui-se que o problema do Corte-Mínimo em Grafos pertence a BPP.

3 *Fontes Aleatórias*

Estabelecemos até o momento algumas classes de linguagens que são definidas por máquinas de Turing com a capacidade de utilizar números aleatórios em seus cálculos. Sua relevância prática e importância apoia-se firmemente na hipótese de que podemos implementar algoritmos aletorizados, em outras palavras, que tenhamos uma fonte de bits aleatórios.

Para formalizar a idéia, definimos uma fonte aleatória perfeita [6] como uma variável aleatória com valores que são seqüências infinitas $(x_1, x_2, \ldots)$ de bits tal que para todo $n > 0$ e para todo $(y_1, y_2, \ldots, y_n) \in \{0,1\}^n$ temos que $Prob[x_i = y_i, i = 1, \ldots, n] = 2^{-n}$, isto é, os x_i's são os resultados de experimentos independentes em que cada x_i é 1 com probabilidade $p = 0$ (esta idéia será extendida à frente, pelo conceito de indistinguibilidade computacional).

Se tivermos uma fonte aleatória perfeita (um dispositivo físico que, pressionando um botão, começaria a gerar uma seqüência $x_1, x_2, x_3, \ldots$), então é possível implementar um algoritmo aleatorizado correspondente a uma máquina de Turing aleatorizada, escolhendo a transição apropriada no i-ésimo passo de acordo com x_i. Assim, com uma fonte aleatória perfeita, problemas nestas classes de complexidade podem ser realmente resolvidos de forma satisfatória. Porém há uma dúvida se é possível obter fisicamente alguma fonte perfeita. Existem algumas fontes físicas de bits aleatórios de boa qualidade, contudo são falhas como fontes aleatórias perfeitas.

O problema real ao implementar fisicamente fontes aleatórias perfeitas é que qualquer processo físico tende a ser afetado pelos resultados anteriores.

Dadas as dificuldades ao implementar fisicamente fontes perfeitas de números aleatórios, tentaremos encontrar aleatoriedade não mais em processos físicos, mas em matemáticos e computacionais. Isto nos leva aos chamados geradores de números pseudoaleatórios.

Tecnicamente um processo computacional que retorna o que parece ser um número aleatório, na verdade, executa um algoritimo específico, que pode ser simulado, mesmo sendo quase impossível ver um padrão na seqüência gerada. Um exemplo simples de tal processo seria tomar o inteiro anterior na seqüência, elevá-lo ao quadrado e tomar os bits intermediários do produto. Números produzidos por esse processo mecânico complexo são chamados de números

pseudoaleatórios.

3.1 Indistinguibilidade Computacional

O conceito de computação eficiente naturalmente nos guia a uma nova espécie de equivalência entre objetos. Dois objetos são considerados computacionalmente equivalentes se não podem ser diferenciados por um *algoritmo eficiente*.[1]

Definição 3.1 (Agrupamentos)**.** *Um agrupamento indexado pelos naturais* $\mathbb{N}$ *é uma sequência de variáveis aleatórias indexadas por* $\mathbb{N}$*. A saber* $X = \{X_i\}_{i\in\mathbb{N}}$*, em que os* $X_{i's}$ *são variáveis aleatórias, é um agrupamento indexado por* $\mathbb{N}$*.*

Definição 3.2 (Agrupamentos aleatórios)**.** *Um agrupamento aleatório* X *é uma família* $X = \{X_n\}_{n\geq 1}$ *tal que* X_n *é uma distribuição aleatória em algum domínio finito.*

Definição 3.3 (Funções desprezíveis)**.** *Uma função* $f : \mathbb{N} \longmapsto [0,1]$ *é desprezível, se e somente se é não negativa e seu resultado tende a zero com o inverso de um polinômio e para n suficientemente grande,*

$$f(n) < \frac{1}{p(n)}.$$

Informalmente, diz-se que dois agrupamentos podem ser tão próximos se suas diferenças são consideradas desprezíveis, ou seja, eles são indistinguíveis.

Definição 3.4 (Indistinguibilidade Computacional)**.** *Dois agrupamentos aleatórios* $\{X_n\}_{n\in\mathbb{N}}$ *e* $\{Y_n\}_{n\in\mathbb{N}}$*, são indistinguíveis computacionalmente se para algum algoritmo aleatorizado A de tempo polinomial, para qualquer polinômio não-negativo p e para n suficientemente grande*

$$|Prob_{x\backsim X_n}[A(x) = 1] - Prob_{y\backsim Y_n}[A(y) = 1]| < \frac{1}{p(n)}$$

Onde $x \backsim X_n$ significa x selecionado de acordo com a distribuição X_n.

3.2 Geradores Pseudoaleatórios

Geradores pseudoaleatórios[1, 2] são definidos como algoritmos determinísticos eficientes que são capazes de gerar sequências longas de bits pseudoaleatórios a partir de uma semente curta de bits aleatórios genuínos. Um princípio fundamental ao examinar estruturas pseudoaleatórias é considerar objetos equivalentes se eles não podem ser distinguidos por algoritmos

eficientes. Assim, a idéia é investigar como gerar bits pseudoaleatórios de forma determinística, tendo como critério principal a impossibilidade computacional de distinguir objetos gerados pseudoaleatoriamente de objetos genuinamente aleatórios, através de procedimentos de tempo polinomial. Isto se refere à noção de indistinguibilidade computacional.

Um gerador pseudoaleatório usa uma pequena string aleatória verdadeira, chamada semente aleatória e a expande em uma longa sequência pseudoaleatória que deve "se parecer" com uma sequência aleatória verdadeira. Há 3 aspectos fundamentais que devem ser ressaltados com respeito à geradores pseudoaleatórios.

1. Eficiência: Computação eficiente é associada a computação conduzida em tempo polinomial com relação ao tamanho da entrada. Assim, é necessário que o gerador seja implementado por um algoritmo determinístico de tempo polinomial, ou seja, o algoritmo recebe uma semente aleatória e aplica algum procedimento determinístico que gere uma sequência pseudoaleatória.

2. Expansão: O gerador deve expandir a semente de entrada de n bits em uma sequência pseudoaleatória de $l(n)$ bits. A função l é chamada função de expansão do gerador.

3. Pseudoaleatoriedade: A saída do gerador deve parecer aleatória, isto é, qualquer procedimento eficiente deve falhar ao tentar distinguir a saída do gerador de uma sequência aleatória de mesmo comprimento. Isto se refere à noção de indistinguibilidade computacional como já discutido anteriormente.

Definição 3.5 (Gerador Pseudoaleatório)**.** *Um algoritmo, G, com função de expansão $l(n)$, é um gerador pseudoaleatório, se obedecer as seguintes condições:*

- *G é um algoritmo determinístico de tempo polinomial*
- *Existe uma função $l : \mathbb{N} \longmapsto \mathbb{N}$, tal que $l(n) > n$ para todo $n \in \mathbb{N}$ e $|G(s)| = l(|s|) > |s|$ para todo $s \in \{0,1\}^*$.*
- *$\{G(U_n)n \in \mathbb{N}\}$ e $\{U_{l(n)}n \in \mathbb{N}\}$ são indistinguíveis computacionalmente, em que U_m denota uma distribuição uniforme sobre $\{0,1\}^m$.*

A noção de gerador pseudoaleatório tem grande importância na prática. Se tivermos a garantia de que um algoritmo é de fato um gerador pseudoaleatório, então podemos usá-lo em qualquer aplicação eficiente que necessite de sequências aleatórias sem mesmo testar o desempenho do gerador na nova aplicação.

3.2.1 Amplificando a função de expansão

Dado um gerador pseudoaleatório com fator de expansão $l_1 = n+1$ construiremos um gerador pseudoaleatório com fator de expansão polinomial. O novo gerador será chamado G.

Construção 3.6. *[5] Tome G_1 um algoritmo determinístico de tempo polinomial que expande sementes de comprimento n em strings de comprimento $n+1$. Tomamos a semente $s(|s| = n)$ e usamos com entrada em G_1. Salvamos o primeiro bit da saída de G_1 (denotado por x_1) e usamos o resto dos n bits em uma nova chamada de G_1. A operação deve ser repetida $l(n)$ vezes. No i-ésimo passo chamamos G_1 com os últimos n bits do passo $i-1$, salvamos o primeiro bit da nova saída com x_i e chamamos novamente G_1 com os últimos n bits. A saída de G é $x = x_1x_2x_3\ldots x_{l(n)}$.*

3.3 Funções Unidirecionais (*One-Way Functions*)

Nossa exposição segue [1].

Há várias definições de funções unidirecionais. As mais conhecidas são as funções unidirecionais fortes (*Strong One-Way Functions*), ou simplesmente funções unidirecionais (*One-Way Functions*).

Basicamente, funções unidirecionais fortes são funções fáceis de computar, entretanto, difíceis de inverter. Explicando mais claramente, a primeira condição trata que, dada uma função f, existe um algoritmo de tempo polinomial que recebe uma entrada x e retorna $f(x)$. A segunda condição trata que, dado qualquer algoritmo aleatorizado de tempo polinomial e uma entrada y, o algoritmo tenta encontrar uma inversa de y sobre f e obtém sucesso com uma probabilidade desprezível.

Definição 3.7 (Funções Undirecionais Fortes)**.** *Uma função $f : \{0,1\}^* \longmapsto \{0,1\}^*$ é chamada unidirecional se obedece às seguintes condições:*

- *Fácil de computar: Dado um algoritmo determinístico de tempo polinomial A, em uma dada entrada x, A retorna $f(x)$, ou seja, $A(x) = f(x)$.*
- *Difícil de inverter: Para qualquer algoritmo aleatorizado de tempo polinomial A', algum polinômio p e n suficientemente grande:*

$$Prob[A'(f(U_n)) = f^{-1}f(U_n)] < \frac{1}{p(n)}$$

Uma função muito popular e candidata a ser unidirecional é baseada no fato conhecido da intratabilidade da fatoração de números inteiros.

Tome $x, y > 1$ inteiros. Definimos

$$f(x,y) = x \cdot y.$$

Quando x e y são números primos, sabe-se que encontrar x e y a partir de $x \cdot y$ é computacionalmente difícil.

Definição 3.8 (Predicado Hardcore)**.** *Um predicado computável em tempo polinomial* $b : \{0,1\}^* \mapsto \{0,1\}$ *é chamado de* Hardcore *de uma função* f *se para todo algoritmo* A' *de tempo polinomial, todo polinomio positivo* $p(.)$*, e n suficientemente grande*

$$Prob_{x\ U_n}[A'(f(x)) = b(x)] < \tfrac{1}{2} + \tfrac{1}{p(n)}$$

3.4 A Relação de Funções unidirecionais com Geradores Pseudoaleatórios

Há uma relação muito estreita entre funções unidirecionais e geradores pseudoaleatórios. Uma propriedade importante de um gerador pseudoaleatório G é que ele transforma uma semente s em uma sequência $x = G(s)$ em tempo polinomial. Porém a operação inversa – encontrar a semente s a partir de x – precisa ser difícil (caso contrário a pseudoaleatoriedade seria violada).

Um gerador pseudoaleatório não é nada mais do que uma função difícil de inverter e que expande uma semente em uma longa semente que parece aleatória. Isto nos leva ao fato de que a existência de um gerador pseudoaleatório implica a existência de uma função unidirecional. Mostraremos que essa condição além de necessária é também suficiente. Assim, um gerador pseudoaleatório existe se e somente se existe uma função unidirecional.

Teorema 3.9. *Geradores pseudoaleatórios existem se e somente se funções unidirecionais existem.*

Demonstração. Considere um gerador pseudoaleatório $G : \{0,1\}^n \longmapsto \{0,1\}^{2n}$ e definimos uma função $f : \{0,1\}^{2n} \longmapsto \{0,1\}^{2n}$ como:

$$f(xy) = G(x) \text{tal que}(|x| = |y| = n)$$

Mostraremos que f é unidirecional provando por contradição.

Suponha a existência de um algoritmo aleatorizado A' de tempo polinomial que é capaz de inverter f com uma probabilidade de sucesso maior que $\frac{1}{p(n)}$, em que p é um polinômio.

Considere o seguinte algoritmo D chamado diferenciador:

Dados : $\alpha, \alpha \in \{0,1\}^{2n}$
$xy \leftarrow A'(x)$;
if $(xy == \alpha)$ **then**
    return 1;
else
    return 0;
end

Algoritmo 6: Diferenciador

$$Prob[D(G(U_n)) = 1] = Prob[D(f(U_n)) = 1]$$
$$= Prob[f(A'(f(U_n))) = f(U_n)]$$
$$= Prob[A'(f(U_n)) = f^{-1}f(U_n)]$$
$$> \frac{1}{p(n)}$$

A última desigualdade ocorre devido à hipótese assumida.

Em contrapartida, há no máximo 2^n strings de comprimento $2n$ que tem uma pré-imagem de G (e de f). Assim, uma string aleatória selecionada uniformemente de comprimento $2n$ tem uma pré-imagem sobre f com probabilidade no máximo $\frac{2^n}{2^{2n}}$. Logo

$$Prob[D(U_{2n}) = 1] = Prob[f(A'(U_{2n})) = U_{2n}]$$
$$\leq Prob[U_{2n} \text{ estar na imagem de } f]$$
$$\leq \frac{2^n}{2^{2n}} = 2^{-n}.$$

Assim,

$$Prob[D(G(U_n)) = 1] - Prob[D(U_{2n}) = 1] > \frac{1}{p(n)} - \frac{1}{2^n} > \frac{1}{q(n)}$$

para algum polinômio q.

□

4 Desaleatorização

A noção de geradores pseudoaleatórios apresentada no capítulo anterior precisa ser estendida, pois há uma diferença entre os geradores que foram apresentados e os geradores que podem ser usados para desaleatorizar um algoritmo. A diferença principal é o tempo de execução. Estes têm um tempo de execução mais lento, porém rápido o suficiente para a simulação.

4.1 Nova noção de gerador pseudoaleatório

Para permitir a desaleatorização precisamos definir uma nova noção de gerador pseudoaleatório. Um gerador que possa enganar qualquer circuito de tamanho polinomial.

Definição 4.1 (*$DTime_i(t(.))$*)**.** *É a classe de linguagens que são decidíveis por uma máuina de Turing determinística de i fitas em $t(.)$ passos. Isto é, $L \in DTime_i(t(.))$ se existe uma máquina de Turing determinística M, com i fitas que decide L e para qualquer $x \in \{0,1\}^*$, e com a entrada x a máquina executa no máximo $t(|x|)$ passos.*

Definição 4.2 (Família de Circuitos Booleanos)**.** *O tamanho de um circuito é o número de portas lógicas que há nele. Uma família de circuitos é uma sequencia infinita $C = (C_0, C_1, ...)$ de circuitos booleanos em que C_n tem n variáveis de entrada. Dizemos que uma linguagem $L \subseteq \{0,1\}^*$ é decidida por um circuito de tamanho polinomial se houver uma família de circuitos $C = (C_0, C_1, ...)$ tal que o seguinte é verdadeiro: Primeiro, o tamanho de C_n é no máximo $p(n)$ para algum polinomio fixo p. E segundo, para todo $x \in \{0,1\}^*, x \in L$ se e somente se a saida de $C_{|x|}(x)$ é verdadeira. [6]*

Definição 4.3 (Geradores Pseudoaleatórios)**.** *Um algoritmo determinístico de tempo polinomial, é chamado de um gerador pseudoaleatório se existe uma função de expansão $l : \mathbb{N} \longmapsto \mathbb{N}$, uma família $\{C_k\}_{k \in L}$ de circuitos de tamanho polinomial, para qualquer polinômio positivo p, e k suficientemente grande.*

$$|Prob_{x\ G(U_k)}[C_k(x) = 1] - Prob_{y\ U_{l(k)}}[C_k(y) = 1]| < \frac{1}{p(k)}$$

PROPOSIÇÃO 4.4:
Se $L \in P$ é uma algoritmo aleatorizado que executa em tempo $t(n)$ e usa $r(n)$ bits aleatórios, então $L \in Dtime(t(n).2^{r(n)})$. Em particular, se $t(n) = poly(n)$ e $r(n) = O(log\ n)$, então $L \in P$ (pode ser decidido em tempo polinomial).

Demonstração. $L \in BPP$ implica que existe algum algoritmo aleatorizado A executando em tempo $t(n)$ para algum polinômio t. Como um limite superior, A usa no máximo $t(n)$ bits aleatórios. Podemos ver A como um algoritmo determinístico com duas entradas (sua entrada regular x e a sequência aleatória r). Escreveremos $A(x,r)$ para denotar a saída de A.

$$Prob[A(x,r) = 1] = \frac{1}{2^{t(n)}} \sum_{r \in \{0,1\}^{t(n)}} A(x,r)$$

Podemos executar $A(x,r)$ para todo r em tempo determinístico $2^{t(n)}$. $t(n)$ e devolver a resposta majoritária.

□

Este método de desaleatorização pode ser aplicado em geral para qualquer algoritmo *BPP*, porém é inexequível (toma tempo exponencial). Contudo, se o algoritmo apenas usa uma quantidade pequena de bits aleatórios ele pode tornar-se exequível.[5]

Usaremos duas ferramentas para construir esse novo tipo de gerador pseudoaleatório, um é chamado *predicado imprevisível* e o outro é uma estrutura combinatória chamada *Design*.

4.1.1 Desaleatorização usando gerador pseudoaleatório

A primeira ferramenta que necessitamos para construir o novo gerador é um predicado que não pode ser aproximado por circuitos de tamanho polinomial.

Definição 4.5 (Predicado Imprevisível)**.** *Dizemos que um predicado computável em tempo exponencial $b : \{0,1\}^l \longmapsto \{0,1\}$ é imprevisível por circuitos de tamanho polinomial, se para qualquer polinômio $p(.)$, para l suficientemente grande e para qualquer circuito C de tamanho $p(l)$.*

$$Prob[C(U_l) = b(U_l)] < \frac{1}{2} + \frac{1}{p(l)}$$

Para validar a veracidade dessa definição, será provado a seguir que a existência de um predicado imprevisível é garantida se existir uma função unidirecional. O inverso não é necessariamente verdadeiro.

Teorema 4.6. *Se f_0 é uma função unidirecional e b_0 é um Hardcore de f_0, então $b(x) = b_0(f_0^{-1}(x))$ é um predicado imprevisível.*

Demonstração. Note que b é computável em tempo exponencial, pois precisamos de tempo exponencial para inverter f e junto mais a computação de b_0, o tempo total não é mais do que exponencial no tamanho de x.

A segunda propriedade que precisamos mostrar é que é impossível de predizer b. Para provar essa propriedade usamos a variável $y = f_0^{-1}(x)$ para obter

$$b(f_0(y)) = b_0(y)$$

Assumimos por contradição que b é previsível. Isto significa que existe um algoritmo A e um polinômio $p(.)$ tal que para infinitos n

$$Prob_{x\ U_n}[A(x) = b(x)] \geq \frac{1}{2} + \frac{1}{p(n)}$$

A é um algoritmo de tempo polinomial que pode prever b com um desvio perceptível. Mas f é uma função então podemos escrever

$$Prob_{x\ U_n}[A(f_0(x)) = b(f_0(x))] \geq \frac{1}{2} + \frac{1}{p(n)}$$

Assim da definição de b temos

$$Prob_{x\ U_n}[A(f_0(x)) = b_0(x)] \geq \frac{1}{2} + \frac{1}{p(n)}$$

que é uma contradição a definição de b_0 como *Hardcore*. □

Gerar um único bit a partir de bits aleatórios parece uma tarefa fácil se tivermos o predicado que definimos anteriormente, pois a saída do predicado deve parecer aleatória para qualquer circuito de tamanho polinomial. O problema é gerar mais de 1 bit. Faremos isso usando uma coleção de conjuntos quase disjuntos que são mutuamente quase independentes.

Definição 4.7. *Uma coleção de m subconjuntos $\{I_1, I_2, \ldots, I_m\}$ é um {k,m,l}-design se as seguintes propriedades valem.*

- Para todo $i \in \{1, \ldots, m\}, |I_i| = l$
- Para todo $i \neq j \in \{1, \ldots, m\}, |I_i \bigcap I_j| = o(\log\ k)$

- A coleção é construtível em tempo exponencial em k.

Em nossa aplicação os conjuntos $\{1,\ldots,k\}$ são as localizações de todos os bits na semente, e os subconjuntos $\{I_1, I_2, \ldots, I_m\}$ correspondem a diferentes subsequências extraídas da semente. Por exemplo, tomando uma semente com 10 bits

$$S = \langle 1010110100 \rangle$$

O subconjunto $I = \{1,3,8\} \subset \{1,\ldots,10\}$ corresponde ao primeiro, o terceiro e o oitavo bit da semente

Neste exemplo,

$$S[I] = \langle 111 \rangle$$

Em geral. para $S = \langle \sigma_1 \sigma_2 \ldots \sigma_k \rangle$, e $I = \{i_1, i_2, \ldots, i_l\} \subset \{1,\ldots,k\}$ usaremos a notação:

$$S = \langle \sigma_{i_1} \sigma_{i_2} \ldots \sigma_{i_k} \rangle$$

PROPOSIÇÃO 4.8:
Tome $b : \{0,1\}^{\sqrt{k}} \longmapsto \{0,1\}$ *um predicado imprevisível, e* $\{I_1, I_2, \ldots, I_m\}$ *como um* $(k, m, \sqrt{k})-$ *design, então a seguinte função é um gerador pseudoaleatório:*

$$G(S) = \langle b(S[I_1])\ b(S[I_2])\ \ldots\ b(S[I_m]) \rangle$$

Demonstração. Esta prova será omitida. □

4.2 Desaleatorização usando grafos expansores

Uma Cadeia de Markov[5] M é um processo estocástico sobre um conjunto de estados S em termos de uma matriz de probabilidades de transição P. A matriz P tem uma linha e uma coluna para cada estado em S. A cadeia está em algum estado em algum tempo, fazendo transições de estado em passos de tempo $t = 1,2,3,\ldots$. A entrada P_{ij} na matriz de probabilidades de transição se refere a probabilidade do próximo estado ser j, dado que o estado atual é i. Assim, para todo $i, j \in S$, temos que $0 \leq P_{ij} \leq 1$, e $\sum_j P_{ij} = 1$

Uma propriedade importante em uma cadeia de Markov é desta não possuir memória, ou seja, o próximo passo na cadeia depende exclusivamente do estado atual e não como ela chegou

ao estado atual. Denotamos por X_t o estado da cadeia de Markov no tempo t. E a sequência $\{X_t\}$ especifica o histórico da cadeia de Markov. A propriedade sem memória de uma cadeia de Markov pode ser afirmada do seguinte modo:

$$Prob[X_{i+1} = j|\ X_0 = i_0, X_1 = i_1, \ldots, X_t = i] = Prob[X_{t+1} = j|\ X_t = i] = P_{ij}.$$

Uma cadeia de Markov não necessita ter um estado inicial pré-definido. Em geral o estado X_0 pode ser escolhido de acordo com alguma distribuição de probabilidades.

Para os estados $i, j \in S$ definimos a probabilidade de transição do passo t como

$$P_{ij}^{(t)} = Prob[X_t = j|X_0 = i].$$

Dado um estado inicial $X_0 = i$, a probabilidade que a primeira transição ao estado j ocorra no tempo t é denotado por $r_{ij}^{(t)}$ e é dado por

$$r_{ij}^{(t)} = Prob[X_t = j, \text{ e para } 1 \leq\ s \leq\ t-1, X_s \neq j|\ X_0 = i].$$

Também, para $X_0 = i$, a probabilidade de ocorrer uma transição a um estado j em algum tempo $t > 0$ é denotada por f_{ij}, e dada por

$$f_{ij} = \sum_{t>0} r_{ij}^{(t)}.$$

O número esperado de passos para atingir o estado j iniciando a partir do estado i é denotado por h_{ij} e dado por:

$$h_{ij} = \sum_{t>0} t\ r_{ij}^{(t)}.$$

Se $f_{ij} < 1$ então $h_{ij} = \infty$.

Definição 4.9. *Definimos $q^{(t)} = (q_1^{(t)}, q_{(2)}^{(t)}, \ldots, q_n^{(t)})$ como a distribuição da cadeia no tempo t. como um vetor linha onde a i-ésima componente é a probabilidade da cadeia estar no estado i e no tempo t.*

É fácil ver que $q^{(t+1)} = q^{(t)}\ P$, temos por indução $q^{(t)} = q^{(0)}\ P$. Segue que o comportamento da cadeia de Markov é dado pela sua distribuição inicial e sua matriz de transição P.

4.2.1 Passeios Aleatórios em Grafos

Tome $G = (V,E)$ um grafo conexo, não bipartido, não dirigido onde $|V| = n$ e $|E| = m$. Um passeio nesse grafo induz uma cadeia de Markov[5] M_G do seguinte modo: Os estados de M_G são os estados de G, e para quaisquer dois vértices $u, v \in G$,

$$P_{uv} = \begin{cases} \frac{1}{d(u)} & \text{se } (u,v) \in E \\ 0 & \text{caso contrário} \end{cases}$$

Onde $d(w)$ é o grau do vértice w.

4.2.2 Amplificação de Probabilidade por Passeios Aleatórios em Expansores

Grafos expansores são grafos esparsos e bem conexos. A cadeia de Markov induzida por um passeio aleatório em um grafo expansor é de grande interesse, pois converge para uma distribuição estacionária em um pequeno número de passos.[5]

Sem perda de generalidade, modificaremos a definição do *BPP* tal que a probabilidade de erro seja $\frac{1}{100}$.

Definição 4.10 (*BPP*)**.** *A classe BPP consiste em todas as linguagens L para as quais existe um algoritmo aleatório A de tempo polinomial, tal que para qualquer* $x \in \Sigma^*$*, dada uma string aleatória r apropriadamente longa*

$$x \in L \Longrightarrow Prob[A(x,r) = 0] \leq \frac{1}{100}$$
$$x \notin L \Longrightarrow Prob[A(x,r) = 1] \leq \frac{1}{100}$$

Fixando uma entrada x, considerando um algoritmo *BPP* A que usa n bits aleatórios em entradas de tamanho $|x|$, supondo que escolhemos k strings aleatórias independentes de n bits $r_1, r_2, \ldots, r_k$ e computando $A(x,r_1), A(x,r_2), \ldots, A(x,r_k)$, pela desigualdade de Chernoff a probabilidade que a maioria das respostas esteja errada é $\frac{1}{2^{\Omega(k)}}$, fazemos a probabilidade de erro exponencialmente pequena em k usando nk bits aleatórios. O problema de amplificação de probabilidade é conseguir esta mesma probabilidade de erro usando um número mínimo de bits aleatórios.

Considere uma família $(N, 7, 2\alpha)$-expansor. Assumindo que N é ímpar. Escolha $m = 2^{\frac{n-1}{2}}$ e $N = 2m^2 = 2^n$, e nomeie cada vértice com uma sequência distinta de bits a partir de $\{0,1\}^n$.

Tome A a matriz de adjacências do expansor. Tome $Q = \frac{(I+\frac{A}{7})}{2}$ a matriz de probabilidade de transição da cadeia de Markov induzida por um passeio aleatório neste grafo, com um *self-loop* de probabilidade $\frac{1}{2}$ em cada vértice.

Escolha um inteiro positivo β tal que $\lambda_2^\beta \leq \frac{1}{10}$, onde λ_i é o i-ésimo maior autovalor de Q.

Obtendo a saída do passeio aleatório descrito acima, a amplificação de probabilidade funciona do seguinte modo. Para $0 \leq i \leq 7k$, tome $r_i = X_{i\beta}$. Execute o algoritmo $A(x,*)$ usando as $7k$ entradas aleatórias diferentes. declare a maioria das decisões como a decisão final. Note que o número total de bits usados é $n+O(k)$: precisamos de n bits para escolher o vértice inicial do passeio aleatório, e 4 bits para cada um dos $7k\beta$ passos seguintes do passeio.

Denotamos o vetor de probabilidades $r_i = X_{i\beta}$ como $p^{(i)}$. Defina $B = Q^\beta$. Esta é a matriz de transição para a cadeia de Markov correspondente a sequência de $r_i's$. Temos $p^{(i)} = p^{(0)}\ B^i$, em que $p^{(0)}$ é a distribuição uniforme a qual usaremos de início.

$\mathbb{W}$ denota o conjunto de testemunhas para a entrada x. Em outras palavras, $\mathbb{W} = \{r \in \{0,1\}^n |\ A(x,r) = 1\}$. Garantimos que $|\mathbb{W}|\ \geq 0.99N$. O conjunto de não testemunhas tem cardinalidade $|\overline{\mathbb{W}}|\ \leq 0.01N$. Definimos a matriz diagonal $0-1NxN\ W$ tal que $W_{ii} = 1$ se e somente se o i-ésimo vértice corresponde a uma string que é testemunha de x; similarmente a matriz diagonal $0-1NxN\ \overline{W} = I - W$.

Considere uma sequência de strings $r_1, \ldots, r_{7k}$. Tome as matrizes de sequência de eventos $S = (S_1, \ldots, S_{7k}) \in \{W, \overline{W}\}^{7k}$ sendo tal que $S_i = W$ se e somente se $r_i \in \mathbb{W}$. Assim S codifica o padrão de erros em várias execuções do algoritmo.

LEMA 4.11:
Para qualquer sequência de eventos fixa S,

$$Prob[S\ ocorrer] = \|p^{(0)}(BS_1)(BS_2)\ldots(BS_{7k-1})(BS_{7k})\|_1$$

LEMA 4.12:
Para todos os vetores $p \in \mathbb{R}^{\mathbb{N}}$,

- $\|pBW\| \leq \|p\|$.
- $\|pBW\| \leq \frac{1}{3}\|p\|$.

Onde:

$\|w\|_1 = \sum_{i=1}^n |w_i|$ representa norma-1, e

$$\|w\| = \sqrt{w^T w} = \sqrt{\sum_{i=1}^{n} w_i^2} \text{ representa norma-2.}$$

Teorema 4.13. *A probabilidade da maioria das saídas $A(x,r_1),\ldots,A(x,r_{7k})$ estar incorreta é no máximo $\frac{1}{2^k}$.*

Demonstração. Note que a maioria das saídas é incorreta somente se a sequência de eventos S ter mais da metade de seus elementos igual a $\overline{W}$. Fixando algum S cujos elementos contém a maioria de $\overline{W}'s$, concluimos que $\kappa \geq \frac{7k}{2}$.

$$\begin{aligned}
Prob[S \text{ ocorrer}] &= \|p^{(0)}(BS_1)(BS_2)\ldots(BS_{7k-1})(BS_{7k})\|_1 \\
&\leq \sqrt{N}\|p^{(0)}(BS_1)(BS_2)\ldots(BS_{7k-1})(BS_{7k})\| \\
&\leq \sqrt{N}(\tfrac{1}{5})^{\kappa}\ \|p^{(0)}\| \\
&\leq \sqrt{N}(\tfrac{1}{5})^{\frac{7k}{2}}\ \|p^{(0)}\| \\
Prob[\text{Maioria dos votos estarem incorretos}] &\leq 2^{7k}\sqrt{N}(\tfrac{1}{5})^{\frac{7k}{2}}\ \|p^{(0)}\| \\
&\leq \frac{2^{7k}}{5^{\frac{7k}{2}}} \\
&\leq \tfrac{1}{2^k}.
\end{aligned}$$

□

5 Conclusão

Este trabalho mostrou de forma simplificada e resumida alguns dos principais conceitos matemáticos e computacionais envolvidos por trás da derandomização, assim como um estudo preliminar das classes de complexidade aleatórias. Por ser um assunto muito extenso e com muitas ramificações alguns aspectos foram tratados apenas de maneira superficial, porém isso não causou um prejuízo no entendimento do trabalho.

Existem várias técnicas que podem ser usadas para este fim, porém a derandomização utilizando passeio aleatório em grafos expansores tende a ser a mais poderosa, e por consequência a mais interessante.

Referências Bibliográficas

[1] Oded Goldreich. *Lecture Notes for a Two-Semester course*. Weizmann Institute of Science, 1999.

[2] Oded Goldreich. *Foundations of cryptography*. Cambridge University Press, Cambridge, 2001. Basic tools.

[3] Steven J. Leon. *Linear Algebra with Applications*. Prentice Hall, Cambridge University, Dartmouth, 1998.

[4] Michael Mitzenmacher and Eli Upfal. *Probability and computing*. Cambridge University Press, Cambridge, 2005. Randomized algorithms and probabilistic analysis.

[5] Rajeev Motwani and Prabhakar Raghavan. *Randomized algorithms*. Cambridge University Press, Cambridge, 1995.

[6] Christos H. Papadimitriou. *Computational Complexity*. Addison Wesley, University of California at Berkeley, 1994.

APÊNDICE A

A.1 Desigualdade de Markov

Teorema A.1 (Desigualdade de Markov)**.** *Tome X uma variável aleatória que assume valores inteiros não-negativos. Então para qualquer* $k > 0$.

Demonstração. Tome p_i como a probabilidade de $X = i$

$$\varepsilon(x) = \sum_i ip_i = \sum_{i<k} ip_i + \sum_{i\geq k} ip_i \geq \sum_{i\geq k} ip_i \geq \sum_{i\geq k} kp_i$$

$$= k\sum_{i\geq k} p_i = kProb[x \geq k]$$

$$\varepsilon(x) \geq kProb[x \geq k]$$

$$Prob[x \geq k] \leq \frac{\varepsilon(x)}{k}$$

□

A.2 Valor Esperado de uma variável aleatória

Em teoria das probabilidades, o valor esperado (ou *esperança*, ou *expectância*) de uma variável aleatória é a soma das probabilidades de cada possibilidade de saída da experiéncia multiplicada pelo seu valor. Isto é, representa o valor médio *esperado* de uma experiéncia se ela for repetida muitas vezes. Note-se que o valor em si pode não ser esperado no sentido geral; pode ser improvável ou impossível. Se todos os eventos tiverem igual probabilidade o valor esperado é a média aritmética.

Para uma variável aleatória discreta X com valores possíveis $x_1, x_2, x_3, \ldots$ e com as suas probabilidades representadas pela função $p(x_i)$, o valor esperado calcula-se pela série:

$$\varepsilon[X] = \sum_{i=1}^{\infty} x_i p(x_i)$$

desde que a série seja convergente.

O valor esperado de uma combinação linear de variáveis aleatórias é a combinação linear dos seus valores esperados:

$$\varepsilon[aX + bY] = a\varepsilon[X] + b\varepsilon[Y]$$

Por esse motivo, a função $\varepsilon[]$ que associa a cada variável aleatória o seu valor esperado é um operador linear, chamado de operador esperança

A.3 Variáveis aleatórias de Bernoulli

Suponha que executaremos um experimento que obtém sucesso com probabilidade p e falha com probabilidade $1-p$.

Tome Y uma variável aleatória, tal que:

$$Y = \begin{cases} 1 \textit{ em caso de sucesso} \\ 0 \textit{ caso} \text{ contrário} \end{cases}$$

A variável Y é chamada *variável aleatória de Bernoulli*, note que para uma *variável aleatória de Bernoulli*:

$$\varepsilon[Y] = p1 + (1-p)0 = p = Prob[Y = 1]$$

Por exemplo, se lançarmos uma moeda e considerarmos o resultado "cara" como sucesso, então a esperança da *variável aleatória de Bernoulli* é $\frac{1}{2}$.

Considere agora uma sequéncia de n lançamentos de moeda independentes. Qual é a distribuição do número de caras na sequéncia toda? Generalizando, considere uma sequéncia de n experimentos. Cada um obtém sucesso com probabilidade p. Se tomarmos X como a representação de sucesso em n experimentos, então X é uma distribuição binomial.

Definição A.2. *Uma variável binomial aleatória X com parâmetros n e p, é definida pela seguinte distribuição probabilística, com $j = 0, 1, \ldots, n$.*

$$Prob[X = j] = \binom{n}{j} p^j (1-p)^{n-j}$$

Isto é, a variável binomial aleatória X igual a j, havendo j sucessos e $n-j$ falhas em n experimentos independentes, cada um obtém sucesso com probabilidade p.

Qual é a *esperança* de uma variável binomial aleatória X? Vemos diretamente da definição que:

$$\begin{aligned}
\varepsilon[X] &= \textstyle\sum_{j=0}^{n} j\binom{n}{j} p_j (1-p)^{n-j} \\
&= \textstyle\sum_{j=0}^{n} \frac{(n-1)!}{j!(n-j)!} p^j (1-p)^{n-j} \\
&= \textstyle\sum_{j=1}^{n} \frac{n!}{(j-1)!(n-j)!} p^j (1-p)^{n-j} \\
&= np \textstyle\sum_{j=1}^{n} \frac{(n-1)!}{(j-1)!((n-1)(j-1))!} p^{j-1} (1-p)^{(n-1)-(j-1)} \\
&= np \textstyle\sum_{k=0}^{n-1} \frac{(n-1)!}{k!((n-1)-k)!} p^k (1-p)^{(n-1)-k} \\
&= np \textstyle\sum_{k=0}^{n-1} \binom{n-1}{k} p^k (1-p)^{(n-1)-k} = np
\end{aligned}$$

e a última equação usa a identidade binomial,

$$(x+1)^n = \textstyle\sum_{k=0}^{n} \binom{n}{k} x^k y^{n-k}$$

A linearidade das esperanças permite um argumento simples. Se X é uma variável binomial aleatória com parâmetros n e p, então X é o número de sucessos em n tentativas onde cada tentativa obtém sucesso com probabilidade p.

Definimos um conjunto de n variáveis aleatórias $X_1, X_2, \ldots, X_n$, onde $X_i = 1$ se a tentativa i obtém sucesso e 0, caso contrário.

Claramente, $\varepsilon[X_i] = p$ e $X = \sum_{i=1}^{n} X_i$, e também pela linearidade das esperanças:

$$\varepsilon[X] = \varepsilon[\textstyle\sum_{i=1}^{n} X_i] = \sum \varepsilon[X_i] = np$$

A.4 Conceitos de Algebra Linear

A.4.1 Autovalores e Autovetores

Definição A.3. *Seja A uma matriz quadrada $n \times n$. Um escalar λ é* autovalor *ou* valor característico *de A se existe um vetor não-nulo tal que $Ax = \lambda x$. O vetor x é um* autovetor *ou* vetor característico *associado a λ.*

A.4.2 Norma de Vetores

[3]

Definição A.4. *A Norma 1 é definida como:* $\|X\|_1 = \sum_{i=1}^{n} |x_i|$

Definição A.5. *A Norma 2 é definida como:* $\|X\|_2 = (\sum_{i=1}^{n} |{x_i}^2|)^{\frac{1}{2}}$