CI2612: Algoritmos y Estructuras de Datos II

Blai Bonet

Universidad Simón Bolívar, Caracas, Venezuela

Algoritmos randomizados

Objetivos

- Hacer análisis probabilístico para calcular **tiempo promedio** de algoritmos determinísticos
- Utilizar **randomización** en el diseño de algoritmos
- Hacer análisis probabilístico para calcular **tiempo esperado** de algoritmos randomizados
- Diferencia entre tiempo promedio y tiempo esperado
- Algoritmos tipo Monte Carlo y Las Vegas

Problema de empleo (hiring problem)

Considere una compañia que desea contratar un nuevo asistente. La compañia contrata una agencia de empleo la cual envia un candidato cada día

La compañia entrevista al candidato y debe decidir si contratarlo o no después de la entrevista. La agencia de empleo cobra un monto pequeño por cada entrevista y un monto bastante mayor si se contrata

La compañia decide probar n días la estrategia de siempre contratar si el candidato es mejor que el asistente actual

¿Cuál es el costo promedio de esta estrategia de empleo?

Algoritmo de empleo

La estrategia de empleo se puede resumir con el siguiente algoritmo

```
Hire-Assistant(int n)
    best = 0                          % candidato 0 es un sentinela
    for i = 1 to n do
        Entrevistar al candidato i
        if i es mejor a best then
            best = i
            Contratar al candidato i
```

El **costo promedio** basicamente depende del número promedio de candidatos contratados

Análisis del algoritmo de empleo

Para analizar el costo promedio, asumimos que todos los candidatos tienen una **calificación diferente** y que son presentados a la compañia de forma **aleatoria**

Calculamos el **costo promedio** en que incurre el algoritmo utilizando variables aleatorias indicadoras

Sea $X_i = \mathbb{I}\{\text{candidato } i \text{ es contratado}\}$ y $X = \sum_{i=1}^{n} X_i$ el número total de candidatos contratados. Queremos calcular $\mathbb{E}[X]$ con respecto a la distribución sobre las posibles entradas

$$\begin{aligned} \mathbb{E}[X_i] &= \mathbb{P}(\text{candidato } i \text{ es contratado}) \\ &= \mathbb{P}(\text{candidato } i \text{ es mejor que los candidatos } 1, \ldots, i-1) \\ &= \tfrac{(n-1)!}{n!} \;=\; \tfrac{1}{n} \end{aligned}$$

[los primeros i candidatos aparecen en cualquier orden]

Por lo tanto, $\mathbb{E}[X] = \sum_{i=1}^{n} \mathbb{E}[X_i] = \sum_{i=1}^{n} \frac{1}{i} = \ln n + O(1)$

Algoritmos randomizados (aleatorizados)

Se dice que un algoritmo es randomizado cuando su ejecución no solo depende de la entrada sino también de los valores producidos por un **generador de números aleatorios**

Asumiremos que todo algoritmo tiene a su disposición una rutina `Random(a,b)` para generar números aleatorios

Una llamada a `Random(a,b)` retorna un número en $\{a, a+1, \ldots, b\}$ seleccionado de forma **aleatoriamente uniforme**

`Random(0,1)` retorna 0 con probabilidad $\frac{1}{2}$ y 1 con probabilidad $\frac{1}{2}$. `Random(4,9)` retorna un entero en $\{4,5,6,7,8,9\}$ cada uno con probabilidad $\frac{1}{6}$

Cada entero generado por `Random` es **independiente de todos los enteros generados anteriormente**

Espacio de probabilidad para algoritmos randomizados

El espacio de probabilidad $(\Omega, \mathcal{F}, \mathbb{P})$ asociado a **una ejecución** de un algoritmo randomizado es el espacio donde cada par de eventos A y B asociados a **llamadas distintas** a `Random` son independientes:

$$\mathbb{P}(A \cap B) \;=\; \mathbb{P}(A) \times \mathbb{P}(B)$$

E.g., si el algoritmo realiza n llamadas a `Random(0,1)` y solo esas llamadas a `Random` durante su ejecución, el algoritmo basicamente "lanza n monedas insesgadas e independientes"

En dicho caso, $\Omega = \{H, T\}^n$ y la medida satisface (entre otras cosas):

$$\mathbb{P}(\{\omega \in \Omega : \omega_i = H\}) \;=\; \tfrac{1}{2} \qquad \text{para todo } i = 1, 2, \ldots, n$$

Análisis de algoritmos randomizados

Al hacer el aňalisis de algoritmos randomizados se debe considerar todas los posibles valores retornados por las llamadas al generador

El algoritmo es correcto ssi es correcto para cualquier valor de los enteros aleatorios

Para el desempeño, calculamos el **tiempo esperado de ejecución** con respecto a la distribución de probabilidad sobre los números aleatorios generados

Algoritmo randomizado para empleo

Para randomizar el algoritmo cambiamos el modelo y asumimos que la agencia envia a la compaῆia la lista de los n candidatos a ser entrevistados

Una manera fácil de randomizar el algoritmo es **permutar aleatoriamente** la lista de candidatos antes de iniciar las entrevistas

Input: arreglo A con candidatos a entrevistar

```
Randomized-Hire-Assistant(array A)
    Permutar de forma aleatoria el arreglo A
    best = 0                               % candidato 0 es un sentinela
    for i = 1 to A.length do
        Entrevistar al candidato A[i]
        if A[i] es mejor a best then
            best = A[i]
            Contratar al candidato A[i]
```

Por la randomización, el **costo esperado** es igual al **costo promedio** de Hire-Assistant; i.e. esperamos realizar $O(\log n)$ contrataciones

Sobre la generación de enteros aleatorios

Hasta ahora hemos supuesto que tenemos a nuestra disposición el generador Random(a,b)

¿Qué hacemos si no lo tenemos o tenemos algo mas débil?

- Si no tenemos un generador genuino de aleatoridad, podemos utilizar un **generador de números pseudo-aleatorios**
- Si solo tenemos monedas sesgadas con Biased-Random(0,1), utilizamos Biased-Random(0,1) para implementar Random(0,1) (Ejercicio 5.1-2)
- Si solo tenemos el generador Random(0,1) (i.e. solo podemos "lanzar monedas insesgadas"), utilizamos Random(0,1) para implementar Random(a,b) (Ejercicio 5.1-3)

Solo podemos lanzar monedas sesgadas

Considere la rutina Biased-Random(0,1) que retorna 0 con probabilidad p y retorna 1 con probabilidad $1-p$ donde p es fijo

```
Random(0,1)
    while true do
        x = Biased-Random(0,1)
        y = Biased-Random(0,1)
        if x == 0 & y == 1 then return 0
        if x == 1 & y == 0 then return 1
```

En una iteración, Random(0,1) retorna 0 con probabilidad $p(1-p)$ y retorna 1 con probabilidad $(1-p)p$. Ambas probabilidades son iguales por lo que Random(0,1) es un generador uniforme sobre $\{0,1\}$

Tiempo esperado de ejecución

Sea T la v.a. para el tiempo de ejecución (en unidades de tiempo) de Random(0,1). Queremos calcular $\mathbb{E}[T]$

Como T es entero y $T \geq 0$, utilizamos $\mathbb{E}[T] = \sum_{t \geq 1} \mathbb{P}(T \geq t)$

Una iteración **falla** si $x = y$ (ambos 1 o ambos 0). La probabilidad de que eso suceda es $r = p^2 + (1-p)^2$

$T \geq t$ ssi existen $t-1$ fallas sucesivas desde el inicio. Como cada iteración es independiente de las otras, $\mathbb{P}(T \geq t) = r^{t-1}$

$$\mathbb{E}[T] = \sum_{t \geq 1} \mathbb{P}(T \geq t) = \sum_{t \geq 1} r^{t-1} = \sum_{t \geq 0} r^t = \frac{1}{1-r}$$

E.g., si $p = \frac{1}{2}$ entonces $r = \frac{1}{2}$ y $\mathbb{E}[T] = 2$. Si $p = 0.99$ entonces $r = 0.9802$ y $\mathbb{E}[T] = 50.505$

Solo podemos lanzar monedas

Mostramos como implementar Random(0,b) utilizando Random(0,1) (implementar Random(a,b) usando Random(0,1) se deja como ejercicio)

La idea es **generar los bits** del número aleatorio con Random(0,1)

```
Random(0,b)
  n = ⌈log(b+1)⌉
  while true do
      x = 0
      for i = 1 to n do
          x = 2 * x + Random(0,1)
      if x <= b then return x
```

En una iteración, Random(0,b) retorna un $x \in [0, b]$ si $x \leq b$. En otro caso, itera. Como la probabilidad es igual para cualquier x retornado, la generación es uniforme

Permutaciones aleatorias de arreglos

Muchos algoritmos randomizados **permutan la entrada** de forma uniforme. Si la entrada es un arreglo, esto equivale a permutar el arreglo de forma tal que cualquier permutación es **equiprobable**

Veremos dos formas de permutar un arreglo A:

– permutación por ordenamiento

– permutación por intercambio

1er Método: Permutación por ordenamiento

La idea es generar una **prioridad** para cada elemento del arreglo y luego ordenar el arreglo utilizando las prioridades como **claves para el ordenamiento**

Input: arreglo A con n elementos

Output: arreglo A permutado

```
Permute-By-Sorting(array A)
    n = A.length
    let P be new array[n]
    for i = 1 to n do
        P[i] = Random(1,n^3)
    Ordenar A usando P como las claves de los elementos en A
```

Si todas las prioridades son distintas, las claves son una permutación uniforme de prioridades y los elementos en A son permutados de forma uniforme

Permutación por ordenamiento

¿Por qué las prioridades (claves) se generan en el rango $\{1, 2, \ldots, n^3\}$?

Porque con gran probabilidad todas las prioridades serán distintas (serán distintas con **probabilidad al menos** $1 - \frac{1}{n}$)

Sea p_i la v.a. que denota la prioridad del elemento $A[i]$

$$
\begin{aligned}
&\mathbb{P}(\text{las prioridades no son únicas}) \\
&\quad = \mathbb{P}(\exists i, j : p_i = p_j) \\
&\quad \le \textstyle\sum_{1 \le i < j \le n} \mathbb{P}(p_i = p_j) \quad \text{(cota de unión)} \\
&\quad = \textstyle\sum_{1 \le i < j \le n} \sum_{k=1}^{n^3} \mathbb{P}(p_i = k, p_j = k) \\
&\quad = \textstyle\sum_{1 \le i < j \le n} \sum_{k=1}^{n^3} \mathbb{P}(p_i = k) \times \mathbb{P}(p_j = k) \\
&\quad = \textstyle\sum_{1 \le i < j \le n} \sum_{k=1}^{n^3} \frac{1}{n^3} \times \frac{1}{n^3} \\
&\quad = \textstyle\sum_{1 \le i < j \le n} \frac{1}{n^3} \;=\; \binom{n}{2}\frac{1}{n^3} \;<\; \frac{1}{n}
\end{aligned}
$$

2do Método: Permutación por intercambio

El segundo método es más simple. Consiste en intercambiar cada elemento $A[i]$ con un elemento al azar en $A[i \ldots n]$

Input: arreglo A con n elementos

Output: arreglo A permutado

```
Permute-By-Swap(array A)
    n = A.length
    for i = 1 to n do
        Intercambiar A[i] con A[Random(i,n)]
```

Claramente el algoritmo corre en tiempo $\Theta(n)$

Correctitud de permutación por intercambio

Invariante:

Al comienzo de cada iteración, el subarreglo $A[1 \ldots i-1]$ contiene una $(i-1)$-permutación con probabilidad uniforme

Inicialización: en la primera iteración, $i = 1$ y el invariante dice que $A[1 \ldots 0]$ contiene una 0-permutación con probabilidad uniforme. En dicha iteración, $A[1 \ldots 0]$ es la única 0-permutación válida, la permutación vacía

Correctitud de permutación por intercambio

Invariante:

Al comienzo de cada iteración, el subarreglo $A[1 \ldots i-1]$ contiene una $(i-1)$-permutación con probabilidad uniforme

Mantenimiento: asuma que estamos por comenzar la i-ésima iteración y que el invariante es cierto. Sea $\langle x_1, x_2 \ldots, x_{i-1} \rangle$ la permutación en $A[1 \ldots i-1]$ y E_{i-1} el evento $A[1 \ldots i-1] = \langle x_1, \ldots, x_{i-1} \rangle$. Por el invariante, $\mathbb{P}(E_{i-1}) = 1/\binom{n}{i-1}(i-1)! = (n-i+1)!/n!$

La iteración intercambia $A[i]$ con $A[j]$ para $j = \texttt{Random(i,n)}$. Denote $A[1 \ldots i] = \langle x_1, \ldots, x_i \rangle$ al final de la iteración y sea E_i el evento de que el algoritmo construye $\langle x_1, \ldots, x_i \rangle$. Sea S el evento de escoger el índice j para intercambiar con $A[i]$

$$\mathbb{P}(E_i) = \mathbb{P}(S \cap E_{i-1}) = \mathbb{P}(S|E_{i-1}) \times \mathbb{P}(E_{i-1}) = \frac{1}{n-i+1} \times \frac{(n-i+1)!}{n!} = \frac{(n-i)!}{n!}$$

Correctitud de permutación por intercambio

Invariante:

Al comienzo de cada iteración, el subarreglo $A[1 \ldots i-1]$ *contiene una* $(i-1)$*-permutación con probabilidad uniforme*

Terminación: el lazo termina cuando $i = n+1$. Al finalizar el invariante dice que la permutación en $A[1 \ldots n]$ es una n-permutación cualquiera con probabilidad $\frac{1}{n!}$

Entonces podemos concluir que el **algoritmo es correcto**

Verificación probabilística de productos de matrices

Queremos diseñar un algoritmo que dadas tres matrices cuadradas con coeficientes en los enteros A, B y C, verifique si $A \times B = C$

(Nota: la restricción a matrices cuadradas no es necesaria)

Una forma es calcular el producto $A \times B$ y comparar con C. Esto toma $O(n^3)$ o $O(n^{2.81})$ si usamos Strassen. Queremos un algoritmo más eficiente

Veremos que la verificación puede hacerse en tiempo $O(n^2)$ con gran precisión utilizando un **algoritmo randomizado**

IDEA: generar un vector aleatorio z, computar $A \times (B \times z)$ y $C \times z$, y comparar los resultados. Si $A \times B = C$, ambos resultados son iguales. Si $A \times B \neq C$, **esperamos** que ambos resultados sean diferentes

Algoritmo de Freivalds (1977)

Input: matrices A, B y C de dimensión $n \times n$

Output: **true** si $A \times B = C$, **false** en otro caso

```
Verify-Product(matrix A, matrix B, matrix C)
    n = A.nrows
    let z be new matrix[n][1]      % vector columna de dimensión n×1
    for i = 1 to n do
        z[i] = Random(0,1)         % construimos vector aleatorio
    y = B×z
    x1 = A×y
    x2 = C×z
    return x1 == x2                % chequea si A×(B×z) = C×z
```

- El algoritmo corre en tiempo $\Theta(n^2)$
- Si $A \times B = C$, `Verify-Product(A,B,C)` retorna **true**
- Si $A \times B \neq C$, `Verify-Product(A,B,C)` puede o no retornar **false** **(i.e. el algoritmo puede cometer errores)**

Las Vegas vs. Monte Carlo

Algoritmo tipo Las Vegas:
- algoritmo corre en **tiempo esperado polinomial**
- algoritmo es **siempre correcto**
- "el algoritmo juega con el tiempo de ejecución"

Algoritmo tipo Monte Carlo:
- algoritmo corre en **tiempo determinístico polinomial**
- algoritmo **puede incurrir en errores** pero con probabilidad acotada
- "el algoritmo juega con el resultado"

Conversión:
- Todo algoritmo Las Vegas puede ser convertido a Monte Carlo
- La otra dirección no es cierta

Error en algoritmo de Freivalds

Si $A \times B = C$, el algoritmo siempre retorna **true** y no existe error

Si $A \times B \neq C$, el algoritmo erra cuando $(A \times B) \times z = C \times z$ donde $z \in \{0,1\}^n$ es el vector aleatorio construido por el algoritmo

Sea $D = A \times B - C = (d_{ij})$. Como $A \times B \neq C$, debe existir un coeficiente en D distinto de 0. Sea $d_{ij} \neq 0$ y defina $x = D \times z$ dado por

$$x_i \ = \ \sum_{k=1}^{n} d_{ik} z_k \ = \ d_{i1}z_1 + d_{i2}z_2 + \cdots + d_{in}z_n \ = \ d_{ij}z_j + y$$

Calculamos,

$$\mathbb{P}(x_i = 0 | y = 0) \ = \ \mathbb{P}(d_{ij}z_j = 0) \ = \ \mathbb{P}(z_j = 0) \ = \ \tfrac{1}{2}$$

$$\mathbb{P}(x_i = 0 | y \neq 0) \ = \ \mathbb{P}(z_j = 1 \wedge d_{ij} = -y) \ \leq \ \mathbb{P}(z_j = 1) \ = \ \tfrac{1}{2}$$

Error en algoritmo de Freivalds

Si $A \times B = C$, el algoritmo siempre retorna **true** y no existe error

Si $A \times B \neq C$, el algoritmo erra cuando $(A \times B) \times z = C \times z$ donde $z \in \{0,1\}^n$ es el vector aleatorio construido por el algoritmo

Sea $D = A \times B - C = (d_{ij})$. Como $A \times B \neq C$, debe existir un coeficiente en D distinto de 0. Sea $d_{ij} \neq 0$ y defina $x = D \times z$ dado por

$$x_i \ = \ \sum_{k=1}^{n} d_{ik} z_k \ = \ d_{i1}z_1 + d_{i2}z_2 + \cdots + d_{in}z_n \ = \ d_{ij}z_j + y$$

Finalmente,

$$\begin{aligned}
\mathbb{P}(x_i = 0) \ &= \ \mathbb{P}(x_i = 0, y = 0) + \mathbb{P}(x_i = 0, y \neq 0) \\
&= \ \mathbb{P}(x_i = 0 | y = 0) \times \mathbb{P}(y = 0) + \mathbb{P}(x_i = 0 | y \neq 0) \times \mathbb{P}(y \neq 0) \\
&\leq \ \tfrac{1}{2}[\mathbb{P}(y = 0) + \mathbb{P}(y \neq 0)] \ = \ \tfrac{1}{2} \\
\mathbb{P}(error) \ &= \ \mathbb{P}(x_1 = 0 \wedge x_2 = 0 \wedge \cdots \wedge x_n = 0) \ \leq \ \mathbb{P}(x_i = 0) \ \leq \ \tfrac{1}{2}
\end{aligned}$$

Amplificación mediante ejecuciones independientes

La **garantía** $\mathbb{P}(error) \leq \frac{1}{2}$ para el algoritmo de Freivalds parece pobre. Sin embargo, ella es **independiente** de la dimensión $n \times n$

Podemos **amplificar** esta garantía corriendo el algoritmo de Freivalds múltiples veces

Si corremos el algoritmo de Freivalds k veces sobre la misma entrada y retornamos **true** ssi todas las corridas devuelven **true**, entonces obtenemos un algoritmo cuya probabilidad de error es $\leq \frac{1}{2^k}$

Para **constante** k moderada, e.g. $k = 50$, obtenemos un **algoritmo confiable de verificación** que corre en tiempo $\Theta(kn^2) = \Theta(n^2)$

No se conoce ningún algoritmo determinístico que realice la verificación del producto en tiempo $O(n^2)$

Resumen

- Un algoritmo es randomizado cuando hace llamadas a un generados de números aleatorios: `Random(p,q)`
- Diferencia entre tiempo promedio de un algoritmo determinístico y tiempo esperado de un algoritmo randomizado (también podemos hablar de tiempo promedio de un algoritmo randomizado y aqui hablamos del tiempo esperado cuando promediamos sobre la aleatorización dentro del algoritmo y la aleatorización de las entradas)
- Análisis probabilístico para calcular tiempos promedio y esperado de algoritmos
- Algoritmo de Freivalds y poder de la randomización

Ejercicios (1 de 4)

1. (5.1-2) Implemente el procedimiento `Random(a,b)` utilizando el procedimiento `Random(0,1)`. Calcule el tiempo esperado de ejecución de `Random(a,b)` en función de a y b.

2. (5.2-1) Para `Hire-Assistant`, asumiendo que los candidatos se presentan en orden aleatorio, ¿cuál es la probabilidad de que solo se contrate a un candidato? ¿cuál es la probabilidad que se contraten los n candidatos?

3. (5.2-1) Para `Hire-Assistant`, asumiendo que los candidatos se presentan en orden aleatorio, ¿cuál es la probabilidad de que solo se contraten a dos candidatos?

4. Obtenga una versión Las Vegas del algoritmo `Permute-By-Sorting` para permutar arreglos

Ejercicios (2 de 4)

5. (5.3-2) ¿Cuál es el problema con el siguiente algoritmo para permutar de forma aleatoria un arreglo A?

```
Permute-Without-Identity(array A)
    n = A.length
    for i = 1 to n-1 do
        Intercambiar A[i] con A[Random(i+1,n)]
```

6. (5.3-3) ¿Produce el siguiente algoritmo una permutación aleatoria uniforme?

```
Permute-With-All(array A)
    n = A.length
    for i = 1 to n do
        Intercambiar A[i] con A[Random(1,n)]
```


Ejercicios (3 de 4)

7. (5.3-7) Suponga que queremos crear un subconjunto aleatorio uniforme de tamaño m a partir de un conjunto de m elementos (cada subconjunto debe tener probabilidad de ser retornado igual a $1/\binom{n}{m}$). Una forma de hacerlo es generar una permutación aleatoria uniforme de los n elementos y retornar los primeros m elementos. Esto requiere generar n números aleatorios y puede ser ineficiente cuando $m \ll n$. Muestre que el siguiente algoritmo recursivo construye un m-subconjunto aleatorio uniforme de $\{1, 2, \ldots, n\}$ generando solo m números aleatorios

```
Random-Sample(int m, int n)
    if m == 0 then
        return ∅
    else
        S = Random-Sample(m-1, n-1)
        i = Random(1,n)
        if i ∈ S then
            return S ∪ {n}
        else
            return S ∪ {i}
```


Ejercicios (4 de 4)

8. Solucionar problema 5-2 del libro