Universidad Nacional Autónoma de México
Licenciatura en Ciencias de la Computación, Facultad de Ciencias
Computación Distribuida
Examen 3
Información General

Profesor:	Miguel Angel Piña Avelino
Ayudantes:	Diego Estrada Mejía, Daniela Susana Vega Monroy
Laboratoristas:	Luis Fernando Fong Baeza, Pablo Gerardo González López
Alumno:	Ricardo Montiel Manriquez
Num. de Cuenta:	314332662
Fecha:	14 de enero del 2022
Fecha de Entrega	15 de enero del 2022 a las 12:00

Problemas

1. (2 puntos) Considera un sistema eventualmente síncrono con las siguientes características: (1) la gráfica de comunicación G es arbitraria, (2) el número máxima de fallas, t, de tipo paro es menor a la conexidad por vértices de G, $\mathcal{K}(G)$, (3) existe una constante desconocida Δ tal que, eventualmente, todo mensaje que manda un proceso p a uno de sus vecinos q en G, toma Δ unidades de tiempo en llegar a q y ser procesado.

   Muestra que el algoritmo 1, basado en la técnica de relojes vectoriales, cumple las propiedades del detector de fallas $\diamond P$:

   - Eventualmente, la variable *suspect* de todo proceso correcto incluye a todos los procesos incorrectos.
   - Eventualmente, la variable *suspect* de todo procesos no incluye a procesos correctos.

2. (2 puntos) Considera el siguiente problema definido por las siguientes dos propiedades, en el que cada proceso inicia con una propuesta:

   - Existen dos n-vectores distintos I_1, I_2, con propuestas en sus entradas, y dos valores distintos, d_1, d_2, tales que en toda ejecución del algoritmo en la que los procesos inician con propuestas en I_j (es decir, cada p_i empieza con propuesta $I_j[i]$), la decisión de todo proceso que decide en esa ejecución es d_j, donde $j \in \{1, 2\}$.
   - Todo proceso correcto eventualmente toma una decisión.

   Considera también un sistema asíncrono en el que la gráfica de comunicación es la completa, hay a lo más $t < \frac{n}{2}$ fallas de tipo paro y los procesos tienen acceso a un detector de fallas de tipo Ω[1]. ¿Es el problema descrito arriba soluble en este sistema? Justifica tu respuesta.

   Supongamos dos vectores $I1$ y $I2$ cuyos valores de entrada son $I1 = [1, 2], I2 = [2, 2]$, observemos que por la definición de los vectores ocurre que $I1 \neq 2$

[1] La definición de Ω se encuentra en la pág. 273 de Distributed Computing de Attiya y Welch

Algoritmo 1 Algoritmo de $\diamond P$ para topología arbitraria

Algoritmo $\diamond P$ TopoArbitraria:

T = 1
suspect $= \varnothing$
C = entero positivo
neighbors = mis vecinos en G
ID = mi identificador, en el espacio $\{1, \ldots, n\}$
$VC[1, \ldots, n] = [0, \ldots, 0]$
$timeout[1, \ldots, n] = [1, \ldots, 1]$

Ejecuta al iniciar y después cada C unidades de tiempo:

VC[ID] = VC[ID] + 1 *%% anuncio mi nuevo evento en mi reloj vectorial VC*
send $\langle VC \rangle$ a todos en *neighbors*

Al recibir un mensaje $\langle VC' \rangle$ **de** $q \in$ **neighbors:**

for each $i \in \{1, \ldots, n\}$ **do**
if $VC'[i] > VC[i]$ **then** *%% VC' tiene información más reciente de i*
if $i \in suspect$ **then**
$suspect = suspect \backslash \{i\}$
$T = 2T$
end if
$timeout[i] = clock()$
end if
$VC[i] = max(VC[i], VC'[i])$
end for

Al cumplirse $(clock() - timeout[i]) > T$ **para algún** $i \in \{1, \ldots, n\} \backslash \{ID\}$**:**

$suspect = suspect \cup \{i\}$

Por definición de Ω, sabemos que el detector de fallos elegirá un líder; entonces si el líder elegido es un proceso correcto p', tenemos que el proceso líder puede distinguir que la propuesta 2 proviene del vector $I1$. Sin embargo, tenemos que la información obtenida en el proceso p' s insuficiente para hacer una distinción entre los vectores y determinar de cual llego la propuesta.

Por lo tanto tenemos que nos es posible resolver el problema de elección de líder y por consiguiente el problema de consenso en el sistema dado.

3. (2 puntos) Muestra un algoritmo aleatorio y centralizado tipo Monte Carlo que solucione el siguiente problema que recibe como entrada un entero $p > 0$: si p es primo, entonces devuelve `true`, de otra forma forma, devuelve `false` con probabilidad mayor a cero.

   La fuente de información aleatoria que puede usar tu algoritmo es una cadena aleatoria de bits, modelada de la siguiente forma: existe una función `random_bit()` que devuelve un bit elegido con probabilidad uniforme, la cual puede ser invocada varias veces y el resultado de la invocación actual es independiente de las invocaciones anteriores. Tu algoritmo debe invocar $o(\log p)$ veces la función `random_bit()` y correr en tiempo $o(\log p)$ (en ambos casos es o pequeña). Argumenta que tu algoritmo es correcto, además, argumenta como la repetición de tu algoritmo amplifica la probabilidad de éxito.

   Consideramos el siguiente algoritmo aleatorio de tipo Monte Carlo, con la cualidad de terminación, pero con probabilidad de error en su resultado:

```
1    si p = 1 entonces
2        regresa false;
3    fin
4    si p = 2 entonces
5        regresa true;
6    fin
7
8    prime ← true
9    for i ← 1 to k do
10       r ← ⊥
11       l ← log_2(p)
12       randomArray ← [l]
13
14       for j ← 0 to l-1 do
15           randomArray[j] ← random_bit()
16       fin
17
18       r ← integer(randomArray)
19       si r ≥ p entonces
20           randomArray[0] ← 0 //Se reduce el n mero para ser menor a p;
21           r ← integer(randomArray);
22       fin
23
24       if r < 2 entonces
25           randomArray[l-1] ← 0 //Se modifica el bit 2^0
26           randomArray[l-2] ← 1 //Se modifica el bit 2^1
27           r ← integer(randomArray)
28       fin
29       m ← r^(p-1) mod p;
30       si m != 1 entonces
31           prime ← false
32       fin
```

```
    fin
    return primo


```

Notamos que el entero r aleatorio generado es tal que $r2, ...p1$ [*], y que dado que el algoritmo esta a lo mas acotado por tantas iteraciones como k, es de tipo Monte Carlo, considerando además el siguiente argumento de correctez:

El algoritmo se basa en el peque˜no teorema de Fermat (Fermat's Little Theorem [FLT]), el cual enuncia:

Si p es primo y a no es divisible por p, entonces:

$$a^{p-1} \equiv 1 mod p$$

Esto equivale a: Si p es primo, entonces para todo a ¡p:

$$a^{p-1} \equiv 1 mod p$$

Así, se sigue que el algoritmo es correcto, pues:

- 1 no es primo y se devuelve false
- 2 es primo y se devuelve true
- Se tienen dos casos:
  - Todos los enteros i, con 2 ¡i ¡p cumplen

    $$a^{p-1} \equiv 1 mod p$$

    y por lo tanto, p es primo. Si esto ocurre, cualquier elección aleatoria de r por [*] cumplir´a con

    $$r^{p-1} \equiv 1 mod p$$

    por lo tanto, nunca se cambiar´a prime a false, y por lo tanto, se devolverá true, concordando con la hipótesis de que p era primo.
  - Existe al menos un entero i, con 2 ¡i ¡p tal que

    $$a^{p-1} \neq 1 mod p$$

    Por lo tanto, p no es primo. y pueden pasar dos situaciones:
    - Se elige r = i, en cuyo caso, se detecta

      $$r^{p-1} \neq 1 mod p$$

      así, el algoritmo cambia la variable prime a false, la cual se termina devolviendo, y por lo tanto, en esta opción, false coincide con que p no era primo.

- Sin perdida de generalidad, se elige r $\neq$ i, en cuyo caso, se detecta

$$r^{p-1} \equiv 1 mod p$$

así, el algoritmo no cambia la variable prime a false, por lo cual se termina devolviendo true, lo cual no coincide con que p no era primo.

Por lo tanto, si p se supone no primo, se devuelve false sujeto a la probabilidad de elección de r, la cual es mayor a 0 por [*] a 0 en el siguiente argumento.

Por lo anterior, como para todo entero p se cumple con la especificación del problema, el algoritmo de muestra correcto

Mostramos ahora un argumento por el cual la repetición k veces de la rutina aumenta la probabilidad de éxito.

Existen números no primos n, llamados números de Carmichael tales que cumplen

$$a^{n-1} \equiv 1 mod n$$

para todo a que es primo relativo a p, así, la probabilidad de que nuestro algoritmo devuelva true para dichos números esta sujeta a la probabilidad de elegir a tal que es primo relativo de n.

Tenemos que hallar un primo relativo a para un n dado es tal como:

$$\frac{c}{n}$$

con c ¡n dado que existe para un numero n, una cota menor c ¡n con la propiedad de ser primos relativos a n, así, para k iteraciones donde se elige un a con la misma probabilidad, dado que cada llamada a random bit() es independiente entre si corresponde a:

$$(\frac{c}{n})^k$$

la cual tiende a reducirse cuando k crece, pues $(\frac{c}{n})^k < 1$, y en particular

$$lim_{k\to\infty}(\frac{c}{n})^k = 0$$

por lo tanto, la probabilidad de éxito aumenta conforme a k que indexa las repeticiones del algoritmo. Finalmente, mostramos que el algoritmo cumple con la complejidad establecida (consideraremos la complejidad de cada iteración, o esto es, cuando k = 1).

Notemos que se hace una llamada a random bit() tantas veces como log2(p), por lo tanto, el orden del n´umero de llamadas a la función en el algoritmo está acotado de la forma O(log(p)), por lo cual, se satisface que esta acotado mediante la notación opequeña por o(p), ya que p es acota asintóticamente a log(p) , siendo $p > log(p)$.

Luego, como la complejidad de nuestro algoritmo esta dada cuando k = 1, tenemos las siguientes observaciones:

- Encontrar blog2(p)c toma tiempo O(log(p))
- Por lo anterior, se hacen llamadas a random bit() con orden O(log(p))

- La exponenciación modular realizada en la linea 25 tiene complejidad O(log(p - 1)) [1], esto es, el exponente al cual se eleva r
- Todas las demás operaciones (accesos a memoria, comparaciones y asignaciones) se realizan en tiempo constante.

De lo anterior, el máximo de las complejidades es O(log(p)) en el algoritmo, y dado que asintóticamente, p ¿log(p), la complejidad del algoritmo queda acotada en la forma o(p). Argumentamos además que las lineas 10 - 24 aseguran que el entero aleatorio finalmente elegido corresponde a un entero que vive en el conjunto 2, ..., p - 1 por los siguientes motivos:

- Por construcción, (10 - 15) se genera un binario, al que luego se le realiza un casting con tantos bits como los mínimos que necesita p para ser representado en sistema binario.
- Decimos que r no puede elegirse tal que $r \geq p$, ya que por las lineas (16 - 19), si detectamos que r era mayor o igual que p, apagamos el bit mas significativo, con lo cual, el binario resultante puede estar constituido por a lo más blog2(p)c - 1 posiciones binarias, así, el máximo número representable sería

  $$2^{log2(p)} - 1$$

  notemos que al ser representado p con al menos blog2(p)c posiciones decimales, necesariamente:

  $$p \geq 2^{log2(p)}$$

  y de allí, concluimos, después de asignar a r el casting del binario con el bit m´as significativo apagado, se cumple:
- Finalmente, decimos que el mínimo número aleatorio que se devuelve es 2, ya que si el entero p es 1 o 0, se devuelve el correspondiente valor binario, y no se realiza el algoritmo, así, p entra al algoritmo si $p \geq 3$, pero como esto sucede, podemos afirmar que

  $$log2(p) \geq 2$$

  por lo tanto, siempre podemos alterar los últimos dos bits del arreglo, para tener al menos en r un valor tal que

  $$r \geq 2$$

4. (4 puntos) Considera el algoritmo 2 aleatorio y distribuido tipo Las Vegas que corre en un sistema síncrono, sin fallas, sin identificadores y la gráfica de comunicación G tiene una topología arbitraria. Responde lo siguiente:

   a) El algoritmo nunca calcula una coloración inválida, es decir, nunca es el caso que dos procesos vecinos en G deciden el mismo color.

   Tenemos que la proposición se cumple. Ya que al ser un algoritmo aleatorio tipo las Vegas, por propiedades de este tipo de algoritmos, sabemos que estos se pueden ejecutar

Algoritmo 2 Algoritmo de coloración en un sistema síncrono, sin fallas, sin identificadores y G con topología arbitraria

Algoritmo ColoreaProba:
color = cadena de bits, inicializada a la cadena vacía \\ el color de cada proceso se codifica en binario
activos = conjunto de mis vecinos en G

Ejecuta cada tiempo $t \geq 0$**:**
b = bit aleatorio elegido con probabilidad uniforme
$color = color \cdot b$ \\ el símbolo $\cdot$ denota la concatenación
send ⟨color⟩ a todos mis vecinos en *activos*

Al recibir un mensaje $\langle color' \rangle$ **de un vecino** v**:**
if $color \neq color'$ **then**
$activos = activos \backslash \{v\}$
if $activos == \varnothing$ **then**
return *color*

varias veces hasta funcionar. Entonces tenemos que en cada tiempo t $\geq$ 0 el algoritmo se estará ejecutando varias veces hasta que dos vértices adyacentes cualesquiera tengan diferente color.

b) Describe una ejecución en la que por lo menos un par de procesos nunca terminan. ¿Existe una ejecución en la que todos los procesos, a excepción de uno, terminan?

Vamos a proponer una ejecución del algoritmo sobre la gráfica completa con dos vértices K2, donde los dos procesos, en cada una de las rondas, generan el mismo bit aleatorio, por lo que sus cadenas de color son iguales en cada una de las rondas, y la condición del si en la línea 9 del algoritmo nunca se cumple, haciendo que nunca se eliminen mutuamente de su lista de activos. Esto se podría dar si ambos procesos recibieran la misma semilla en su generador pseudoaleatorio de bits. Por la naturaleza del algoritmo tipo Las Vegas podemos ver que puede nunca terminar.

¿Existe una ejecución en la que todos los procesos, a excepción de uno, terminan?

No existe una ejecución en la que todos los procesos, a excepción de uno, terminan. Si dicha ejecución existiera, implicaría que dados dos procesos vecinos u y v, u saca a v de su lista de activos, pero v mantiene a u en su lista, por el inciso anterior podemos ver que esto no puede suceder ya que cuando un proceso u elimina de su lista activos a otro proceso v, entonces, en la misma ronda, v elimina a u de sus activos.

c) Dado un entero $K > 1$, describe una ejecución en la que todos los procesos terminan la coloración resultante usa exactamente K colores distintos.

Para la asignación del bit aleatorio utilizaremos una función conocida como la delta de Kroneker: La delta de Kronecker es una función de dos variables, que regresa 1 si son iguales, y 0 si son diferentes. Trivialmente se puede notar que la función es continua, por

lo que el bit será elegido con probabilidad uniforme, así que podemos usar la función sin problema.

Consideremos a la gráfica completa con K vértices, cuyo conjunto de vértices es v1, ..., vK. Supongamos que en cada tiempo t, la función que obtiene el bit aleatorio del proceso vi con i 1, .., K está dada por d(i, t) (la delta de Kronecker).

Entonces, en el tiempo i, el bit aleatorio que todos los procesos generan es 0, excepto el proceso vi, que genera 1. Por lo tanto, en el tiempo i, para cualesquiera j ¿i, n ¿i, el bit generado aleatoriamente por ambos procesos es 0, por lo que color del proceso vj es igual al color del proceso vn (ya que sus cadenas de colores son ambos una concatenación de i ceros), por lo que no se eliminan de sus correspondientes listas activos. Por otro lado, como el bit generado aleatoriamente para vi es 1, va a diferir de todos sus vecinos, por lo que los borra a todos de su lista activos, y todos lo borran a ´el de su correspondiente lista. Así en el tiempo i el proceso vi decide su color, que es distinto al de todos los que han decidido anteriormente, y es el único que decide. Siguiendo lo anterior, para la ronda K 1 solo tendremos dos procesos restantes con el mismo color, una vez que aplicamos la función y obtenemos el color del proceso vK-1 como 00..,1 y el del proceso vK como 00..,0, tendremos que sus cadenas de bits de color tendrán la misma longitud, pero van a diferir en valor, por lo que se borrarían mutuamente de sus listas de activos, vaciándoles y regresando así sus colores.

Los colores con los que el algoritmo terminará decidiendo son: 1, 01, 001, 0001, ..., 00..,01, 0..,00, donde las últimas dos cadenas son de longitud K - 1. Notemos que toda ejecución que termine sobre la gráfica completa de K vértices tiene la propiedad deseada, esta ejecución correrá por K - 1 rondas, lo que nos dará K colores distintos.

d) Decimos que una arista e de G está en *conflicto* en el tiempo $t \geq 0$ si los procesos en los extremos de e tienen el mismo color en ese tiempo, es decir, las cadenas de bits en sus variables locales *color* son iguales, en el tiempo t. Entonces, todas las aristas de G están en conflicto en tiempo $t = 0$. Sea C_t el conjunto de aristas en conflicto en tiempo t. Demuestra lo siguiente:

1) $C_0 \supseteq C_1 \supseteq C_2 \supseteq \ldots$

   Sean Ci y Ci+1. Quiero demostrar que Ci $\supseteq$ Ci+1. Por contradicción, supongo que no es cierto esto. Por lo tanto existe e $\notin$ Ci pero e $\in$ Ci+1. Es decir, en el tiempo C_{i+1}, e estaba en conflicto, pero en Ci no lo estaba. Por lo que en algún punto en el algoritmo se tuvo que haber agregado la arista al conjunto de activos. Pero esto es imposible pues sólo se quitan aristas en el algoritmo, nunca se agregan. De esta forma, Ci $\supseteq$ C_{i+1}. Y como fueron cualquiera, se cumple que $C0 \supseteq C1 \supseteq C2 \supseteq$. . . .

2) Para todo $t \geq 0$, el tamaño esperado de $C_{t+1} = |C_t|/2$.

   Para cada t, se escoge con probabilidad uniforme un bit nuevo. Como este bit nuevo sólo puede escogerse entre 0 y 1, hay $\frac{1}{2}$ de probabilidad de escoger 0, y el resto a 1. Por lo tanto, se espera que la mitad de C_t escoja un 0, y la otra mitad escoja un 1. Entonces, si comparamos cuáles fueron los resultados, esperamos que $\frac{1}{2}$ de los

vecinos de cada nodo tenga un 0 y la otra un 1. Por lo que la mitad de los vecinos de cada nodo dejarán de estar en conflicto.

Que la mitad de los vecinos dejen de estar en conflicto quiere decir que la mitad de las aristas dejarán de estar en conflicto. De esta manera, es esperado que $\frac{|C_t|}{2}$ dejen de estar en conflicto. Por lo que se espera que $|Ct+1| = \frac{|Ct|}{2}$

3) Usa el resultado anterior para (a) mostrar que el tiempo esperado de ejecución del algoritmo es $O(\log n)$, (b) calcular el número esperado de mensajes que se mandan en una ejecución y (c) el número máximo esperado de colores con que se colorean los procesos en una ejecución.

Para (a), tengo que ver que para que el algoritmo termine se necesita que no existan aristas en conflicto. Como es esperado que en cada iteración se reduzca a la mitad las aristas en conflicto, hay que hacer esto la cantidad de veces hasta que $C_t = 1$. Es decir, es esperado que en cada iteración se reduzca a la mitad. Y una vez que sabemos que se reduce a la mitad C_t en cada iteración, puedo ver que a lo más tengo $O(n^2)$ aristas. Esto es tomando en cuenta la gráfica completa. Si hay k iteraciones, entonces tarda:

$$\frac{n^2}{n^k} = 1$$
$$n^2 = 2^k$$
$$log(n^2) = k$$
$$2log(n) = k$$

Verificar que dos nodos tengan la misma coloración es constante. Por lo que se espera que la ejecución tarde $O(2c * log(n)) = O(log(n))$

Para (b), en cada iteración se mandan el número de vecinos activos de cada nodo. Al principio hay n nodos activos, y se manda un mensaje para cada nodo. Esto es la cantidad de aristas por dos, uno de cada lado de la arista. Por lo que toma $2 * |E|$. Después se espera que se manden la mitad de mensajes, porque se redujo a la mitad las aristas en conflicto. Por lo tanto, se mandan $|E|$ mensajes. Esto se repite log(n) veces. Que es:

$$\sum_{i=0}^{log(n)} \frac{2 * |E|}{2^i}$$
$$= 2 * |E| \sum_{i=0}^{log(n)} \frac{1}{2^i}$$
$$\leq 2 * |E| * 2$$
$$= 4 * |E|$$

Por lo que se espera que se manden a lo más $4 * |E|$ mensajes.

Para (c), en la primera iteración se espera que haya $\frac{|E|}{2}$ aristas que no se repitieron. Estos corresponden con el color 0, y color 1. Después hay color 0, 1, 2 y 3, porque se espera que la mitad tenga color diferente.

Así, hasta que se repita log(n) veces, la cantidad de veces que se repite el algoritmo. Por lo tanto, hay $2^log(n)$ colores diferentes. El doble en cada iteración. Y $2^log(n) = n$, por lo que en el peor caso se esperan n colores.