Universidad Simón Bolívar
CI-5651 - Diseño de Algoritmos I
Prof. Ricardo Monascal

Resuelto por:
Christopher Gómez

Tarea 9: Algoritmos probabilísticos y aproximados

1. Considere su número de carné (sin el guión) restando 1 si es par, concatenado tres veces, como un número entero. Por ejemplo, si su carné es `12-34567`, entonces el entero a considerar sería:

   $$12345671234567123456712345671234567$$

   Muestre la ejecución del algoritmo de Miller–Rabin repetido, paso a paso (a nivel del ciclo principal de `MillerRabinRep`), para ver si el número es primo o compuesto, usando $k = 10$. ¿En cuántas iteraciones obtiene el resultado esperado? ¿Ejecutó las $k = 10$ iteraciones?

   Nota: Puede usar el generador de números aleatorios que viene con su lenguaje de elección.

   En este caso, el carné es `18-10892`, por lo que el número a considerar es $18108911810891181089118108911$.

   Notemos que, por construcción, para cualquier carnet el número resultante de concatenarlo 3 veces será compuesto, de la siguiente manera:

   Sea $c > 1$, $c \in \mathbb{N}$ un carnet dado, concatenar c 3 veces será equivalente a:

   $$\begin{aligned} c + c * 10000000 + c * 100000000000000 &= c * (1 + 10000000 + 100000000000000) \\ &= c * 100000010000001 \end{aligned}$$

   Veamos entonces cuál es el resultado de la ejecución del algoritmo de Miller-Rabin:

   - El número aleatorio a resulta ser $a = 106681647674259816197$ (generado por un dado de 6 caras).
   - El primer ciclo itera 1 vez:
     - $t = 90544559054455905445$.
   - $x = a^t \bmod 18108911810891181089 = 87323191318267542098$.
   - El resultado no es 1 ni $n - 1 = 18108911810891181089$.
   - Como $s = 1$ el segundo ciclo no itera.
   - Se concluye que el número es compuesto, y no hace falta seguir con las iteraciones restantes.

2. Sea A y B dos matrices $n \times n$, para algún entero $n > 0$.

   Sospechamos que $B = A^{-1}$. Esto es, que B es la matriz inversa de A.

   Diseñe un algoritmo de Monte Carlo que permita confirmar esta sospecha, con un cierto error permitido ε, usando tiempo $O\left(n^2 \log \frac{1}{\varepsilon}\right)$.

   Nota: Puede usar el generador de números aleatorios que viene con su lenguaje de elección.

   Queremos saber si $AB = I_n$, sea I_n la matriz identidad de grado n. Así, un algoritmo de Monte Carlo que nos permite confirmar esta sospecha con un error arbitrario $\varepsilon > 0$ y tiempo $O\left(n^2 \log \frac{1}{\varepsilon}\right)$ es ejecutar el método de Freivalds k veces, con $k = \lg\left(\frac{1}{\varepsilon}\right)$:

```
def freivalds(A: Matriz, B: Matriz, C: Matriz) → bool:
    n = A.tamaño
    x = vector_aleatorio(0, 1, n)

    return x*A*B == x*C

def es_inversa(A: Matriz, B: Matriz, epsilon: Real) → bool:
    n = A.tamaño
    k = ceil(log(1/epsilon))

    I = matriz_identidad(n)
    i = 0
    while i < k:
        if not freivalds(A, B, I):
            return False

        i = i + 1
    return True
```

   Una implementación de este algoritmo en Java se puede encontrar aquí.

3. Sea un grafo $G = (N, C)$, decimos que $V \subseteq N$ es un cubrimiento de vértices para G si todas las conexiones tienen alguno de sus extremos en V.

$$(\forall a, b \in N : \{a, b\} \in C \Rightarrow \{a, b\} \cap V \neq \emptyset)$$

Sea MIN−COVER el problema de hallar un cubrimiento de vértices de cardinalidad mínima. Sabemos que MIN−COVER es NP–completo.

Diseñe un algoritmo para el problema 1–relativo–MIN−COVER asociado. Esto es, diseñe un algoritmo eficiente (en tiempo polinomial) que resuelve el problema del mínimo cubrimiento de vértices, produciendo una respuesta que es a lo sumo el doble de la solución óptima. Debe demostrar que esto último es cierto para su algoritmo propuesto.

Para resolver aproximadamente el problema del mínimo cubrimiento de vértices, basta con ir tomando aristas A cualesquiera del grafo y eliminar las aristas incidentes a los vértices de A, pues ya estarán cubiertas por los vértices de A, que se agregarán al cubrimiento. Si repetimos esto hasta considerar todas las aristas, obtendremos un cubrimiento de vértices. Veamos el algoritmo:

```
def min_cover(G: Grafo) → Conjunto:
    C = {}
    for u, v in G.aristas:
        C.unir({u, v})
        G.eliminar_vertice(u)
        G.eliminar_vertice(v)

    return C
```

Notemos que, como en cada iteración escogemos una arista distinta y eliminamos sus vértices y respectivas aristas incidentes, el cubrimiento C obtenido será equivalente a un *emparejamiento*, y de hecho, uno maximal (si en vez de guardar los vértices guardáramos las aristas). Llamemos M a este emparejamiento.

Por otro lado, tenemos también que un cubrimiento de vértices de cardinalidad mínima debe contener por definición al menos un vértice por cada arista del emparejamiento maximal (de no ser así, no cubriría todas las aristas), es decir, sea C^* la solución óptima, entonces $|M| \leqslant |C^*|$.

Pero el algoritmo propuesto no guarda aristas, sino los dos vértices de la misma, por lo que $|C| = 2|M|$, y por lo tanto $\frac{|C|}{2} \leqslant |C^*|$, o lo que es lo mismo, $|C| \leqslant 2|C^*|$.

Se concluye así que el algoritmo propuesto produce una respuesta que es a lo sumo el doble de la solución óptima.

Una implementación de este algoritmo en Typescript se puede encontrar aquí.