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

Resuelto por:
Christopher Gómez

Tarea 5: Grafos

1. Considere la concatenación de su nombre con su apellido, llevado todo a minúscula y eliminando todos los caracteres repetidos, menos la primera vez que ocurran. Si su nombre es `Fulano Mengano`, entonces debe considerar la cadena `fulanomeg`.

   Se desea que:

   (a) Construya un árbol binario de búsqueda considerando los caracteres en el orden en que aparecen (puede suponer que el orden es lexicográfico).

   (b) Realice un etiquetado en pre-order del árbol resultante.

   (c) Realice un recorrido de Euler sobre el árbol resultante.

   (d) Muestre el cálculo del ancestro común más bajo entre los dos últimos caracteres de su apellido, usando el método de precondicionamiento visto en clase.

   Para el ejemplo de `fulanomeg`, debe ver el ancestro común más bajo entre `e` y `g`.

   En este caso, la cadena a considerar es `christopegmz`. En la Figura 1 se puede ver el árbol binario resultante, con el etiquetado en pre-order a la izquierda de cada nodo y el etiquetado por niveles a la derecha, necesario para el recorrido de Euler.

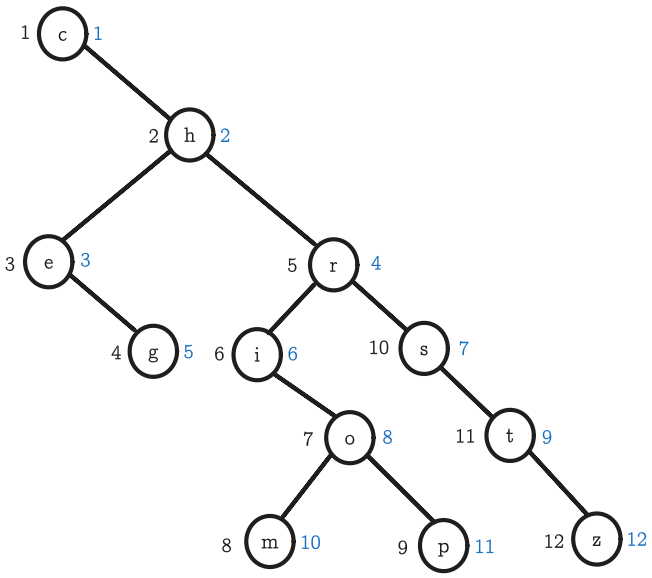


Figura 1: Árbol binario para `christopegmz`

Luego, el recorrido de Euler tomando en cuenta dichas etiquetas será:

1	2	3	5	3	2	4	6	8	10	8	11	8	6	4	7	9	12	9	7	4	2	1

Y la tabla resultante del precondicionamiento es:

Nodo	1	2	3	4	5	6	7	8	9	10	11	12
Primera etiqueta	0	1	2	6	3	7	15	8	16	9	11	17

Como se puede concluir con la tabla y el recorrido de Euler, y como se indica con los colores de las celdas, la primera aparición de `m` (nodo 10) es en la posición 9 y la primera aparición de `z` (nodo 12) es en la posición 17, y la etiqueta más pequeña que se puede encontrar en el recoorrido de Euler en el rango $[9..17]$ del arreglo es 4, que corresponde a la etiqueta de `r`.

Se concluye entonces que el ancestro común más bajo entre `m` y `z` es `r`, y podemos corroborar que el resultado es correcto con la Figura 1.

2. Sea un conjunto C de n números enteros positivos distintos. ¿Cuál es la menor cantidad de números que debe eliminarse de C de tal forma que no existan $x, y \in C$ tal que $x + y$ sea un número primo?

   Diseñe un algoritmo que pueda responder esta consulta en tiempo $O(n^2\sqrt{n})$. A efectos de esta pregunta, puede suponer que consultar si un número es primo es $O(1)$.

   Pista: El teorema de ***König*** *puede ser de utilidad.*

   Veamos, para resolver este problema, comencemos construyendo un grafo

   $$G_C = (C, E)$$
   $$E = \{(x, y) \mid x, y \in C, x + y \text{ es primo}\}$$

   es decir, un grafo donde los vértices son los elementos de C y hay una arista entre cada par de vértices cuya suma sea un número primo.

   Luego, lo que buscamos es "romper" todas estas aristas, es decir, remover para cada una uno de sus extremos, de forma que no queden vértices conectados. Queremos además remover la menor cantidad de vértices posible, por lo que buscamos la cardinalidad de un cubrimiento mínimo M de G_C.

   Notemos que G_C es un grafo bipartito, ya que podemos particionar los vértices en pares e impares, porque la suma de dos números con la misma paridad es par, por lo que toda arista tendrá un extremo par y otro impar.

   Así, sabemos por el *teorema de* ***König*** que para G_C el número de vértices en un cubrimiento mínimo es igual al número de aristas en un emparejamiento máximo, por lo que podemos resolver el problema hayando la cardinalidad de un emparejamiento máximo en G_C con el algoritmo de *Hopcroft-Karp*. Un pseudocódigo para resolver el problema sería:

```
def no_suma_primo(C: Conjunto[Entero]) → Entero:
    E = {}
    P = {}
    I = {}
    r = 0

    for i in C:
        if i == 1:
            r = 1
            continue
        if es_par(i):
            P.añadir(i)
        else:
            I.añadir(i)

    for p in P:
        for i in I:
            if es_primo(p + i):
                E.añadir((p, i))

    G = {P U I, E}
    M = hopcroft_karp(G, P, I)

    return tamaño(M) + r
```

   Un caso especial a tener en cuenta es que el 2 es el único número primo par, por lo que si $1 \in C$, este se excluye del grafo G_C a construir y se suma 1 al resultado, ya que la suma de 1 consigo mismo es primo, caso que no se consideraría al hallar cubrimiento mínimo de G_C.

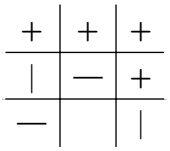
El número de aristas de G_C está acotado por n^2, y el algoritmo de *Hopcroft-Karp* tiene una complejidad de $O\left(|E|\sqrt{|V|}\right)$, por lo que el algoritmo planteado tiene una complejidad de $O\left(n^2\sqrt{n}\right)$ (suponiendo que la verificación de primalidad es $O(1)$), ya que la construcción de G_C es $O\left(n^2\right)$.

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

3. Considere un modificación del clásico juego de la vieja, en donde:

   - El primer jugador juega con — y el segundo juega con |.
   - Cada casilla puede tener alguno de estos símbolos, ninguno o ambos (en cuyo caso se forma un +).
   - En cada turno, el jugador no puede jugar en la misma casilla que el jugador anterior.
   - Gana aquel jugador que logre formar tres + en una misma fila, columna o diagonal.

   Por ejemplo, la siguiente es una configuración ganadora (donde la última jugada fue de |):

   Diga si hay una estrategia ganadora para alguno de los jugadores involucrados. Para resolver este problema, utilice el método **minimax**.

   Para encontrar si hay alguna estrategia ganadora, modelamos primero los estados del juego para poder aplicar el método **minimax**.

   De esta forma, definimos un estado como un arreglo de 9 elementos, donde cada elemento puede es un número entre 0 y 4, siendo 0 si la casilla está vacía, 1 si tiene un —, 2 si tiene un | y 3 si tiene un +.

   Codificar los estados de esta manera presenta la ventaja de que para jugar en una casilla basta sumar el valor del símbolo a la casilla correspondiente.

   Luego, para saber si un estado es ganador, basta con sumar los valores de las casillas en las filas, columnas y diagonales, y verificar si alguna suma es 9, lo que indicaría que un jugador ha ganado.

   Con esto, saber si un estado es ganador consiste simplemente en hacer ejecutar un agente minimax sobre el árbol de estados del juego, partiendo del tablero vacío, y ver si el agente puede forzar una victoria o termina perdiendo. Se representa el estado ganador con un 1, el perdedor con un -1.

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

   Con dicha implementación se pudo verificar que el valor resultante de la raíz del árbol minimax es -1, lo que indica que si ambos jugadores juegan de forma óptima, el primer jugador siempre terminará perdiendo, o lo que es lo mismo, el segundo jugador tiene una estrategia ganadora.