Trabajo Práctico 1

15 de abril de 2017

Algoritmos y Estructuras de Datos III
Primer Cuatrimestre de 2017

Autor	LU	Correo electrónico
Tarrío, Ignacio	363/15	itarrio@dc.uba.ar

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria - (Pabellón I/Planta Baja)
Intendente Güiraldes 2610 - C1428EGA
Ciudad Autónoma de Buenos Aires - Rep. Argentina
Tel/Fax: (++54 +11) 4576-3300
http://www.exactas.uba.ar

Índice

1. Ejercicio 1 (BackTracking): 2

2. Ejercicio 2 (BackTracking con Poda): 4

1. Ejercicio 1 (BackTracking):

En este ejercicio se nos pide resolver el problema utilizando un algoritmo de backtracking.

Backtraking es una técnica para buscar exhaustivamente todas las configuraciones del espacio de soluciones de un problema, para eso va generando las posibles soluciones y dejando atras los cantidatos cuando sabemos que no son validos(que no pertenecen al espacio de soluciones validas del problema).

Para este problema queremos buscar el minimo de numeros sin pintar para una secuencia de numeros de largo n. Ya que un numero puede estar pintado de rojo, azul o sin pintar, y como tenemos n numeros, podemos tener como maximo 3^n posibles combinaciones, aunque podrian ser menos ya que de esas combinaciones hay posibilidades que no son validas como soluciones ya que puede ser que la subsecuencia de rojos no sea creciente o decreciente para la de azul. Una vez generadas todas estas soluciones, deberiamos encontrar la que menos numeros tiene sin pintar y esa va a ser la solucion a nuestro problema(se nos pide la cantidad no que combinacion de rojos y azules).

Ya que estamos construyendo las distintas soluciones, empezamos decidiendo de que color pintar el numero primer numero, rojo azul o sin pintar, para cada una de estas 3 decisiones vamos a pasar al siguiente numero y repetir el proceso, asi hasta llegar al final en el que vamos a tener como pintamos los numero y nos fijamos cuantos dejamos sin pintar. Una vez que tenemos todas las soluciones buscamos la optima.

Se implemento este algoritmo de manera recursiva, dado el arreglo de numeros, el indice, y cual fue el ultimo rojo y ultimo azul pintado(si hay), devuelve la minima cantidad de numero sin pintar dentro de las posibilidades. Pedimos el ultimo rojo y el ultimo azul porque no me importa saber como los pinte si no los ultimos ya que eso me va a dar la limitante de si el proximo numero lo puedo pintar de rojo o azul(y descartamos los que no pude pintar). Entonces esta funcion recursiva guarda los resultados del indice siguiente para las 3 posibilidades(rojo, azul y sin pintar), se fija cual es la menor y devuelve esa(si es la que no se pinta tiene que sumarle uno), excepto que sea el caso baso en el que no hay indice siguiente y devolvemos 0 si podemos pintarlo(0 elementos sin pintar en un arreglo que contiene solo al ultimo elemento pintado de cualquier color) o 1 si no lo pudimos pintar, y despues se van a ir llenando las anteriores llamadas con estos casos bases.

Un poco de codigo:

```
Algoritmo 1: ej (int arreglo[], n) → res: int
  Data: arreglo = el arreglo de numeros entero
  n = el tamaño del arreglo Result: La cantidad minima de numeros sin pintar para una secuencia de numeros
        de largo n
  /* Empezamos el backtracking desde el primer indice(0) y no inicializamos todavia el ultimo
     rojo y el ultimo azul                                                                   */
  res ← BT(0, arreglo, n, NULL, NULL)
```

```
Algoritmo 2: BT (int i, arreglo[], n, ultimoRojo, ultimoAzul) → res: int
  Data: i = indice actual a pintar
  arreglo = el arreglo de numeros entero
  n = el tamaño del arreglo
  ultimoRojo = el ultimo numero que se pinto de rojo(NULL si no se pinto ninguno)
  Result: La cantidad minima de numeros sin pintar a partir de i hasta n
  numeroActual ← arreglo[i]
  if i = n − 1 then
     /* Si el indice es el ultimo numero entonces estamos en el caso base                    */
     /* Tenemos que fijarnos si podemos pintarlo                                             */
     if ultimoRojo = NULL or ultimoAzul = NULL or ultimoRojo < numeroActual or ultimoAzul >
      numeroActual then
        res ← 0
     else
        /* No lo podemos pintar porque no seria una solucion valida entonces devolvemos 1    */
        res ← 1
     end if
  else
     int minSiRojo, minSiAzul, minSinPintar
     if ultimoRojo = NULL or ultimoRojo < numeroActual then
        /* Si lo podemos pintar el numero actual de rojo osea si es mas grande que el anterior
           rojo pintado, o si es el primero rojo en pintarse, y cambiamos el ultimoRojo por
           este numero                                                                       */
        minSiRojo ← BT(i+1, arreglo, n, numeroActual, ultimoAzul)
     end if
     if ultimoAzul = NULL or numeroActual < ultimoAzul then
        /* Lo mismo con el azul                                                              */
        minSiAzul ← BT(i+1, arreglo, n, ultimoRojo, numeroActual)
     end if
     /* Calculamos tambien si no lo pintamos y al resultado le sumamos uno porque este no lo
        pintamos                                                                             */
     minSinPintar ← BT(i+1, arreglo, n, ultimoRojo, ultimoAzul) + 1
     /* retornamos el minimo de ambos(considerando que la func min es O(1) y si es nulo la
        variable no lo considera)                                                            */
     res ← min(minSiRojo, minSiAzul, minSinPintar)
  end if
```

Analisis de complejidad: Ya dijimos que la cantidad de combinaciones posibles eran 3^n, y podemos decir que este algoritmo recorre como maximo todas estas instancias porque empieza desde el indice 0 y por cada numero del arreglo prueba las 3 combinaciones(o al menos las validas), entrando recursivamente al indice siguiente(i+1) y esos lo van a repetir hasta llegar al caso base que seria llegar al ultimo indice. Ahora, cada llamada recursiva hace una cantidad de operaciones O(1) constantes (sumar, comparar, minimo) y si no es el caso base ademas hace 3 llamadas recursivas, pero reduciendo el n. Esto nos hace quedar que la complejidad de la funcion recursiva es $T(n) = 3T(n-1) + O(1)$ con $T(0) = 1$ como el caso base, que se puede demostrar facilmente (por induccion) que es $O(3^n)$. Otra forma de llegar es ver el arbol de ejecucion de la recursion, cada nodo desprende tres nodos hijos, vamos a tener un arbol de altura n, y como es un arbol ternario tenemos 3^n hojas, que son nuestras soluciones. Esta complejidad se considera exponencial y es muy mala para instancias grandes.

TODO Y PONER CASOS PEORES Y MEJORES GRAFICOS y DEMOSTRAR CORRECTITUD

2. Ejercicio 2 (BackTracking con Poda):

En este ejercicio nos piden hacer una mejora en el anterior algoritmo de backtracking y hacerle una poda. Las podas consisten en que hay instancias de soluciones parciales que ya no nos interesan y ni siquiera las calculamos, porque por ejemplo esta solucion va a hacer peor que una que ya calculamos anteriormente. Esto lo que provoca es una optimizacion porque dejamos de calcular cosas, aunque sigue siendo una busqueda exhaustiva.

Si recordamos el anterior algoritmo, para un indice calculaba el menor si lo pintabamos de rojo, despues de azul y despues sin pintar, pero que pasa si uno nos devuelve que se pueden pintar todos los proximos numeros, osea que los los otros no hace falta computarlos ya que solo nos importa la cantidad de numeros sin pintar, y no vamos a encontrar una mejor porque esta es la minima. Esta es nuestra primer poda, una vez calculado una solucion parcial para un color, nos fijamos si es optima y si lo es nos salteamos(no computamos) las otras alternativas de colores. Tambien podriamos llevar una cuenta de cual es la solucion(entera) optima y no calculamos una que pueda ser peor que esa, por ejemplo si voy por el 3 indice y encontramos una solucion que solo no puede pintar un numero, solo tenemos que buscar una solucion que pinta todos los numeros, asi que no vamos a entrar a las recursiones de no pintar.

Para la implementacion de estas podas modificamos un poco la base del codigo del anterior ejercicio asi se nos es mas facil calcular los datos. Ahora la funcion recursiva tiene 2 parametros de entrada nuevos, el minimo encontrado en las soluciones ya calculadas y la cantidad de numeros que ya no pintamos y ahora en vez de devolver un resultado parcial, devuelve la cantidad de sin pintar para todo el arreglo(ya que tenemos la cantidad que no pintamos antes).

Pseudocodigo:

Algoritmo 3: BTI (int i, arreglo[], n, ultimoRojo, ultimoAzul, optimoActual, sinPintar) → res: int

```
Data: i = indice actual a pintar
arreglo = el arreglo de numeros entero
n = el tamaño del arreglo
ultimoRojo = el ultimo numero que se pinto de rojo(NULL si no se pinto ninguno)
ultimoAzul = el ultimo azul pintado
optimoActual = la mejor solucion encontrada hasta el momento sinPintar = cantidad de numeros sin pintar
  antes de i
Result: La cantidad minima de numeros sin pintar a partir de i hasta n
numeroActual ← arreglo[i]
if i = n − 1 then
    /* Si el indice es el ultimo numero entonces estamos en el caso base                */
    /* Tenemos que fijarnos si podemos pintarlo                                          */
    if ultimoRojo = NULL or ultimoAzul = NULL or ultimoRojo < numeroActual or ultimoAzul >
     numeroActual then
        res ← sinPintar
    else
        /* No lo podemos pintar porque no seria una solucion valida entonces devolvemos la
           cantidad que pintamos antes mas 1 porque este no lo pintamos                  */
        res ← sinPintar + 1
    end if
else
    int minSiRojo, minSiAzul, minSinPintar
    if ultimoRojo = NULL or ultimoRojo < numeroActual then
        minSiRojo ← BTI(i+1, arreglo, n, numeroActual, ultimoAzul, optimoActual, sinPintar)
        /* PODA 1: si es el optimo no calculamos los demas y devolvemos este             */
        if minSiRojo = sinPintar then
            res ← minSiRojo
        end if
        if minSiRojo < sinPintar then
            optimoActual ← minSiRojo
        end if
    end if
    if ultimoAzul = NULL or numeroActual < ultimoAzul then
        /* Lo mismo con el azul                                                          */
        minSiAzul ← BTI(i+1, arreglo, n, ultimoRojo, numeroActual, optimoActual, sinPintar)
        /* PODA 1                                                                        */
        if minSiAzul = sinPintar then
            res ← minSiAzul
        end if
        if minSiAzul < sinPintar then
            optimoActual ← minSiAzul
        end if
    end if
    /* PODA 2: solo calculamos sin pintar este numero, si todavia podemos mejorar el optimo */
    if sinPintar < optimoActual −1 then
        minSinPintar ← BTI(i+1, arreglo, n, ultimoRojo, ultimoAzul, optimoActual, sinPintar + 1)
    end if
    res ← min(minSiRojo, minSiAzul, minSinPintar)
end if
```