Tarea 9 - Programación y algoritmos 2

Erick Salvador Alvarez Valencia

CIMAT A.C.,
erick.alvarez@cimat.mx

Resumen En la presente tarea se describirá el código implementado para resolver el problema llamado *Tree Isomorphism* usando el algoritmo de Ahu, el cual se explicará a detalle tanto su versión $O(n^2)$ como la $O(nlogn)$. Al final se mostrarán los resultados obtenidos.

1. Descripción del problema

El problema consiste en dados dos árboles no enraizados A y B se quiere verificar si estos son Isomorfos, la definición de Isomorfismo en árboles es la siguiente:

Sean A y B dos árboles, se dice que A es Isomorfo a B si existe una biyección ϕ tal que para cualquier par de vértices (u, v) se cumple que si estos son adyacentes en A entonces $\phi(u)$ y $\phi(v)$ son adyacentes en B.

Restricciones del problema:

1. Número máximo de casos: 400.
2. Número máximo de nodos: 100000.
3. $1 \leq u, v \leq N$, donde u y v son los valores de los nodos por cada arista.

2. Algoritmo implementado

El algoritmo de Ahu decide si dos árboles son Isomorfos de acuerdo a una representación que este genera de ambos y que, de ser igual, indica que los árboles son Isomorfos. El problema con el algoritmo es que trabaja con árboles enraizados, lo cual no es el caso de este problema. Para solucionarlo debemos buscar un nodo que sea el mismo en ambos árboles Isomorfos, y una propuesta para estos nodo sería usar los nodos centrales de los árboles, ya que se cumple que dos árboles Isomorfos comparten el mismo nodo central. Por lo cual debemos buscar primero los centros de los árboles y luego aplicar el algoritmo de Ahu. El algoritmo de búsqueda de centros consiste en generar el diámetro del árbol y buscar su mediana, la cual en este caso será el o los centros de los árboles, los cuales pueden ser entre 1 y 2.
Para generar el diámetro del árbol y obtener el centro se empieza tomando un nodo cualquiera del árbol y desde ese nodo se busca el nodo más alejado de él, posteriormente se repite este mismo proceso con el nodo obtenido. Al hacer esto

anterior podremos reconstruir el camino entre los dos nodos más alejados y este será el diámetro del árbol, posteriormente podremos obtener el o los centros. A continuación se muestra el pseudocódigo del algorito anterior mencionado:

Algorithm 1 Algoritmo para obtener el centro de un árbol.

```
1: procedure GetCenter(Tree)
2:     parent ← Vector(N + 1)        ▷ Se usará el vector de padres para reconstruir el camino
3:     fill(parent, -1)
4:     leaf1 ← BFS(Tree, 1, parent)          ▷ Hacemos un BFS desde el primer nodo
5:     fill(parent, -1)
6:     leaf ← BFS(Tree, leaf1, parent)
7:     u ← leaf2
8:     path ← {}
9:     while u ≠ parent[u] do    ▷ Reconstruimos el camino que será el diámetro del árbol
10:        path ← path ∪ parent[u]
11:        u ← parent[u]
12:    centers ← {}
13:    if path.length MOD 2 = 0 then
14:        centers ← centers ∪ path[path.length / 2 - 1]
15:        centers ← centers ∪ path[path.length / 2]
16:    else
17:        centers ← centers ∪ path[path.length / 2]
18:    return centers
```

Una vez que se tiene calculado los centros de cada árbol podemos hacer una pequeña comparación, si ambos árboles tienen diferente número de centros podemos decir con seguridad que no son Isomorfos, en caso contrario procedemos a ejecutar el algoritmo de Ahu el cual consiste en recorrer los nodos del árbol por niveles empezando por el nivel más bajo y asignarles una string la cual para los nodos hoja tiene la siguiente forma: *()* y para los nodos que no son hoja su estructura consiste en un paréntesis abierto, luego la concatenación de las strings ordenadas lexicográficamente de sus nodos hijos y finalmente un paréntesis cerrado. Haciendo esta asignación por niveles en ambos árboles podremos llegar al resultado de que si estos son Isomorfos entonces tendrán la misma string en el nodo raíz.
Lo malo de el algoritmo anterior es que si tenemos el caso degenerado de un árbol en forma de lista podemos ver que el generar una cadena para un nuevo nodo requiere concatenar el valor de la cadena del nodo hijo y en cada paso estaremos generando una cadena del doble de tamaño de la anterior por lo que al final este algoritmo es cuadrático.
Una propuesta para mejorar el tiempo del mismo es usar una representación que no genere tanto acarreo como las strings y lo que se hizo en el presente algoritmo fue asignar IDs únicos a los nodos que son diferentes y para ello se hace una DFS

en donde por cada nodo se calcula el ID de cada uno de los hijos, luego se juntan dichos IDs en un vector y eso se ordena, al final para asignar el ID del nodo actual se usa una tabla HASH donde las llaves son los vectores y esta es usada para controlar la asignación de IDs únicos, una vez que aplicada la DFS a ambos árboles se compara el ID de la raíz y si es el mismo entonces son Isomorfos. La complejidad de este algoritmo es $O(nlogn)$ por el hecho de que tiene que ordenar los vectores de IDs en cada nivel. A continuación se muestra el pseudocódigo del algoritmo:

Algorithm 2 DFS.

```
1: procedure DFS(Tree, Names, Prev, U)
2:     vAux ← Vector(N + 1)
3:     for V ← Tree[U] do                    ▷ Para todos los nodos adyacentes a U
4:         if Prev[V] = -1 then
5:             Prev[V] ← U
6:             DFS(Tree, Names, Prev, V)
7:             vAux ← vAux ∪ Names[V]    ▷ Para este punto ya se tiene calculado el
   ID de V
8:     if vAux.length = 0 then
9:         vAux ← vAux ∪ {0}             ▷ Este caso se activa cuando U es nodo hoja
10:    Sort(vAux)
11:    Names[U] ← getID(vAux)        ▷ Aquí se usa la tabla Hash para obtener el ID
```

Algorithm 3 Algoritmo de Ahu.

```
1: procedure AHU(TreeA, TreeB, RootA, RootB)
2:     namesA ← Vector(N + 1)
3:     namesB ← Vector(N + 1)
4:     prevA ← Vector(N + 1)
5:     prevB ← Vector(N + 1)
6:     prev[RootA] ← RootA
7:     prev[RootB] ← RootB
8:     DFS(TreeA, namesA, prevA, RootA)
9:     DFS(TreeB, namesB, prevB, RootB)
10:    if namesA[RootA] = namesB[RootB] then
11:        return True
12:    else
13:        return False
```

Cabe mencionar que es posible programar de otra forma este algoritmo en donde se procesen los nodos por nivel y el ID que se les asigna se hace a partir de la información de los IDs de los nodos del nivel inferior, aunque en esta versión no se requiere una tabla Hash, la ventaja de lo anterior es que se puede extender

de tal forma que usando un algoritmo como Bucket Sort logramos conseguir una implementación que corre en $O(n)$ aunque en la presente tarea sólo se llegó a la implementación de $O(nlogn)$.

3. Resultados

Se envió el código a la plataforma de SPOJ y se obtuvo el resultado de *Aceptado*.

(a) Figura 1. Resultado obtenido al enviar el código a Codechef.

4. Nota extra

Quiero agradecer mucho la dedicación que usted Dr. ha puesto en esta materia, la cual con toda sinceridad ha sido de las mejores que he llevado en la Maestría, lo admiro mucho por eso, además de que admiro el empeño que usted ha puesto a su carrera como investigador, no dudo que va a seguir logrando grandes resultados en su área, así como crecer en el SNI.

De nuevo muchas gracias, espero que el tiempo me permita ser su alumno de nuevo, aunque sea en algún pequeño curso o en una materia de oyente. Yo por mi parte seguiré esforzandome en mejorar mis habilidades como programador competitivo, aunque sea como Hobby.