

Tarea 3 - 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 el problema del ancestro común más bajo (*Longest Common Ancestor*) para el cual se implementaron 2 algoritmos, el primero de ellos es el trivial que tiene un costo computacional $O(n)$ y el otro usa una técnica para convertirlo en el problema del *Range Minimum Query* (RMQ) el cual se verá que se puede preprocesar en tiempo $O(n \log n)$ y contestar cada query en $O(1)$.

1. Descripción del problema

La búsqueda del LCA consiste en dado un árbol no necesariamente binario y un par de nodos que pertenezcan a dicho árbol, se quiere encontrar un nodo tal que sea ancestro de los dos nodos al mismo tiempo y que además, esté lo más alejado posible de la raíz, por estas restricciones antes mencionadas se puede ver que el LCA de dos nodos siempre es único. Este problema ha sido muy estudiado y tiene varias aplicaciones en diferentes áres de la computación, tales como: uso en árboles de sufijos y biología computacional.

A continuación se verán dos formas en las que se resolvió el problema, la primera es la más básica y la segunda conlleva a convertir el problema a una versión en RMQ pero que permite responder las queries muy rápido.

2. Construcción de los árboles

Previo a implementar las soluciones para el LCA se generó un algoritmo para construir árboles y en base a ello generar la entrada del programa. Para poder representar un árbol se usó una estructura que contiene lista de adyacencia, un vector que representa quienes son los padres de todos los nodos y al final quien es el nodo raíz, de esta forma se pueden almacenar árboles en donde cada nodo puede tener los nodos que quiera. Para generar los árboles se usó un algoritmo que consistía en generar una lista con los índices de los nodos, posteriormente hacer un random shuffle a dicha lista, ir tomando nodo por nodo, luego el nodo que se toma pasa a ser hijo de un nodo ya colocado previamente en el árbol, la selección de dicho nodo también es tomada aleatoriamente. De esta forma no hay manera que se generen ciclos en dicho árbol. A continuación se muestra el algoritmo usado para la generación aleatoria de árboles.

Algorithm 1 Generación aleatoria de árboles.

```
1: procedure GENERATERANDOMTREE( $N$ )
2:    $t \leftarrow \text{new Tree}()$ . ▷ Generar una estructura que almacene los árboles.
3:    $\text{nodes} \leftarrow \text{vector}(N)$ 
4:    $\text{used} \leftarrow \text{vector}(N)$ 
5:   for  $i = 0 \rightarrow N - 1$  do
6:      $\text{nodes}[i] \leftarrow i$ 
7:    $\text{randomShuffle}(\text{nodes})$ 
8:    $\text{idx} \leftarrow 0$ 
9:    $\text{used.append}(\text{nodes}[\text{idx}])$ 
10:   $\text{idx} \leftarrow \text{idx} + 1$ 
11:  while  $\text{idx} < N$  do
12:     $r \leftarrow \text{rand}() \% \text{idx}$ 
13:     $t.\text{adj}[\text{used}[r]].\text{append}(\text{nodes}[\text{idx}])$ 
14:     $\text{used.append}(\text{nodes}[\text{idx}])$ 
15:     $\text{idx} \leftarrow \text{idx} + 1$ 
16:   $t.\text{computeParents}()$ 
17:  return  $t$ 
```

Una vez que se tiene el árbol construido se usa otro algoritmo para generar el archivo de salida, en donde además de incluir el árbol se agrega un conjunto de queries que también son generadas aleatoriamente, sólo se cuida que no se elijan los mismos nodos para calcular el LCA. Al final del algoritmo 1 se manda llamar un método del árbol que se llama *computeParents* el cual sólo itera sobre la lista de adyacencia y en un arreglo va indicando para el nodo i cuál es su padre.

3. Solución trivial del LCA

El algoritmo trivial se enfoca en seleccionar uno de los dos nodos de interés, hacer un camino desde ese nodo hasta la raíz e ir marcando los nodos por los que se pasó en el camino, posteriormente se toma el otro nodo y se hace también un recorrido hasta la raíz pero en este caso se busca el primer nodo que intersekte con el camino que se formó anteriormente, dicho nodo será el LCA que se estaba buscando, nota que para este algoritmo siempre existirá un nodo en común para ambos caminos, y es fácil ver que la raíz siempre cumple esto anterior, pero en el caso de que la raíz sea el LCA significa que los nodos se encuentran en subárboles distintos. A continuación se muestra el pseudocódigo de este algoritmo.

Algorithm 2 LCA Naive.

```
1: procedure LCANAIVE( $t, u, v$ )
2:    $n \leftarrow t.n$ 
3:    $vis \leftarrow \{\}$ 
4:   for  $i = 0 \rightarrow n - 1$  do
5:      $vis[i] \leftarrow \text{False}$ 
6:   while True do
7:      $vis[u] \leftarrow \text{True}$ 
8:     if  $t.parent[u] = u$  then                                ▷ Si llegamos al nodo raíz
9:       break
10:     $u \leftarrow t.parent[u]$ 
11:  while True do
12:     $vis[u] \leftarrow \text{True}$ 
13:    if  $vis[v]$  then                                          ▷ Encontramos el LCA
14:      return  $v$ 
15:     $v \leftarrow t.parent[v]$ 
```

El algoritmo anterior siempre hace un recorrido desde los nodos de interés hasta la raíz, se puede dar el caso en donde dichos nodos sean nodos hoja, por lo que la complejidad del algoritmo sería $O(h)$ donde h representa la altura del árbol, pero también puede darse el caso en donde el árbol que nos den sea completamente una lista, y por lo tal su altura sería completamente el número de nodos, por lo que la complejidad final es de $O(n)$.

4. Solución optimizada de LCA

El enfoque de esta solución es convertir el problema de LCA a RMQ y resolverlo eficientemente, para hacer esto anterior primero hay que hacer un preproceso, en donde se hace un tour euleriano al árbol, dicho tour consiste en aplicar un DFS e ir guardando en un vector todos los nodos por los que se pasa, incluyendo cuando se visitan de vuelta. A continuación se muestra la implementación de dicho algoritmo.

Algorithm 3 Euler tour.

```
1: procedure EULERTOUR( $t, path, node$ )
2:    $n \leftarrow t.n$ 
3:    $path.append(node)$ 
4:   for  $i = 0 \rightarrow t.adj[node].size() - 1$  do
5:      $children \leftarrow t.adj[node][i]$ 
6:      $eulerTour(t, path, children)$ 
7:      $path.append(node)$ 
```

La complejidad de este algoritmo es la misma que la del DFS $O(E+v)$ ya que estamos usando una lista de adyacencia para almacenar al árbol. Nota que en la implementación anterior, sólo se está guardando en el vector los nodos por los que el camino euleriano pasó, pero para propósitos del LCA requerimos también el nivel al cual se encuentra cada nodo, ya que podemos usar esta información para convertir el problema a un RMQ. Nota que el LCA de u y de v es el nodo que está localizado en el nivel más alto posible en el camino euleriano, por lo que si guardamos la primera aparición de cada nodo en el camino euleriano podemos hacer consultas al mismo para que busque el nodo con mayor nivel (aquí por mayor nivel me refiero a un menor número) y que esté entre los nodos u y v . Con esto último podemos ver que el problema se ha reducido a hacer un RMQ en el vector que produjo el tour euleriano, y para hacer esto podemos recurrir a una Sparse table, la cual es una estructura de datos que permite hacer consultas en rangos de una manera muy rápida pero con la restricción de que el arreglo debe ser inmutable, lo cual es el caso del presente problema.

La idea principal dentro de una tabla sparse es que para una posición específica de la tabla $table[i][j]$ esta almacena el menor elemento en un rango de tamaño 2^j , es decir que $table[i][j] = \min(arr[i], arr[i+1], \dots, arr[i+2^j-1])$, por lo cual para consultar la tabla en un intervalo $[L, R]$ tenemos que ir buscando potencias de 2 tal que $L+2^k-1 \leq R$ en caso que eso se cumpla consideramos el valor de $table[L][k]$ para ir computando la respuesta, cabe decir que k siempre empieza en $\lfloor \log_2(n) \rfloor$ donde n es el tamaño del arreglo. Una vez que se elige una potencia de 2 para calcular la respuesta se debe actualizar el valor del límite inferior $L += 2^k$, esto porque para contestar una query en el rango $[L, R]$ estamos haciendo particiones con potencias de 2, es decir, primero procesamos $[L, L+2^k-1]$ y luego pasamos al rango $[L+2^k, R]$.

Con lo anterior podemos ver que el tiempo para procesar una query es $O(k) = O(\log_2(n))$, pero para el RMQ lo podemos hacer mejor, y es que una Sparse Table brilla en este tipo de situaciones, ya que la función mínimo cumple que $\min(L, R) = \min(\min(L, c), \min(b, R))$ donde $c \geq b$, es decir, no importa si un elemento se procese más de una vez porque el mínimo no cambiaría en esa situación, por lo que en lugar de procesar cada query en k intervalos podemos simplemente procesarla en 2 de tamaño 2^{k-1} . A continuación se muestra el algoritmo para procesamiento de las queries.

Algorithm 4 RMQ Sparse Table.

```

1: procedure RMQST(Path, Table, L, R)
2:    $k \leftarrow \lfloor \log_2(R - L + 1) \rfloor$ 
3:   if  $Path[Table[L][k]] \leq Path[Table[R - 2^k + 1][k]]$  then return  $Table[L][k]$ 
4:   if
5:     then return  $Table[R - 2^k + 1][k]$ 
6:   return  $0$ 

```

Con lo cual las queries las estaremos respondiendo en $O(1)$.

Ahora la construcción de la tabla podemos usar el hecho de que $2^j = 2^{j-1} + 2^{j-1}$ por lo cual un intervalo $[i, i + 2^j - 1]$ lo podemos calcular con los intervalos $[i, i + 2^{j-1} - 1]$ y $[i + 2^{j-1}, i + 2^j - 1]$ ambos de tamaño 2^{j-1} , el algoritmo es el siguiente.

Algorithm 5 Build Sparse Table.

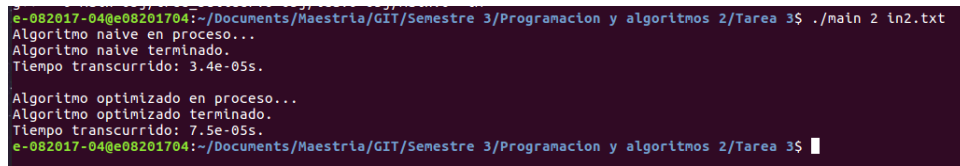
```

1: procedure BUILDST(Path, Table)
2:    $sz \leftarrow \text{Path.size}()$ 
3:    $k \leftarrow \lfloor \log_2(sz) \rfloor$ 
4:   for  $i = 0 \rightarrow sz - 1$  do
5:      $\text{Table}[i][0] \leftarrow i$ 
6:   for  $j = 1, 2^j \leq sz$  do
7:     for  $i = 0, i + 2^j - 1 < sz$  do
8:       if  $\text{Path}[\text{Table}[i][j - 1]] \neq \text{Path}[\text{Table}[i + 2^{j-1}][j - 1]]$  then
9:          $\text{Table}[i][j] \leftarrow \text{Table}[i][j - 1]$ 
10:      else
11:         $\text{Table}[i][j] \leftarrow \text{Table}[i + 2^{j-1}][j - 1]$ 
=0

```

5. Resultados

Los dos algoritmos anteriores fueron probados por diferentes instancias generadas con los algoritmos para generación de árboles aleatorios, se usaron diferentes números de nodos y números de queries. A continuación se muestran algunos de los resultados obtenidos. Cabe mencionar que los resultados de ambos algoritmos fueron comparados con una herramienta web y siempre fueron los mismos.



```

e-082017-04@e08201704:~/Documents/Maestria/GIT/Semestre 3/Programacion y algoritmos 2/Tarea 3$ ./main 2 ln2.txt
Algoritmo naive en proceso...
Algoritmo naive terminado.
Tiempo transcurrido: 3.4e-05s.

Algoritmo optimizado en proceso...
Algoritmo optimizado terminado.
Tiempo transcurrido: 7.5e-05s.
e-082017-04@e08201704:~/Documents/Maestria/GIT/Semestre 3/Programacion y algoritmos 2/Tarea 3$ █

```

(a) Figura 1. Ejecución del algoritmo con un árbol de 10 nodos con 5 queries.

Para este primer ejemplo se usó un árbol con 10 nodos y 5 queries, se puede ver que el algoritmo trivial fue un poco más rápido, esto porque el preprocesamiento que tuvo el algoritmo optimizado tuvo repercusión en el tiempo, además de que no fueron muchas queries.

```
e-082017-04@e08201704:~/Documents/Maestria/GIT/Semestre 3/Programacion y algoritmos 2/Tarea 3$ ./main 2 in1.txt
Algoritmo naíve en proceso...
Algoritmo naíve terminado.
Tiempo transcurrido: 0.039153s.

Algoritmo optimizado en proceso...
Algoritmo optimizado terminado.
Tiempo transcurrido: 0.011624s.
e-082017-04@e08201704:~/Documents/Maestria/GIT/Semestre 3/Programacion y algoritmos 2/Tarea 3$
```

(b) Figura 2. Ejecución del algoritmo con un árbol de 10000 nodos con 1000 queries.

Para el siguiente caso de prueba se usó un árbol de 10000 nodos y 1000 queries, aunque ambos algoritmos terminaron en menos de 1 segundo ya se puede notar una diferencia en los tiempos de ejecución, con el algoritmo optimizado respondiendo todas las queries en una tercera parte de lo que lo hizo el algoritmo trivial.

```
e-082017-04@e08201704:~/Documents/Maestria/GIT/Semestre 3/Programacion y algoritmos 2/Tarea 3$ ./main 2 in4.txt
Algoritmo naíve en proceso...
Algoritmo naíve terminado.
Tiempo transcurrido: 12.575s.

Algoritmo optimizado en proceso...
Algoritmo optimizado terminado.
Tiempo transcurrido: 0.154312s.
e-082017-04@e08201704:~/Documents/Maestria/GIT/Semestre 3/Programacion y algoritmos 2/Tarea 3$
```

(c) Figura 3. Ejecución del algoritmo con un árbol de 100000 nodos con 10000 queries.

En el último caso de prueba se usó un árbol con 100000 nodos y 10000 queries. Aquí es donde se nota una clara diferencia en ambos algoritmos, el primero tardó más de 12 segundos en responder todas las consultas, mientras que el último no tardó ni siquiera 1 segundo.