Universidad de los Andes

Facultad de Ingeniería
Departamento de Ingeniería de Sistemas
Diseño y Análisis de Algoritmos
Periodo 2022-10

Proyecto: Etapa II

Sergio Pardo Gutierrez
Juan Diego Calixto
Juan Diego Yepes

10 de mayo de 2022
Bogotá D.C.

Tabla de contenidos

1 Enunciado

Un grafo no dirigido de n vértices $(v_1, v_2, ..., v_n)$ es construido por un genio matemático según el valor de una permutación de los n primeros números naturales que le es concebida en sus sueños. La permutación le es suministrada como un arreglo $([p_1, p_2, ..., p_n])$ y determina la creación de las aristas. Una arista es creada para los vértices v_i ,v_j $(i < j)$ si y solo si $p_i > p_j$.

Dado un grafo permutación definido mediante una permutación p_n el problema es calcular el número de componentes conectados (a.k.a. conexos) en este grafo.

2 Especificación del problema

Para poder desarrollar las soluciones del problema es necesario especificarlo. La especificación se detalla a continuación:

E/S	Nombre	Tipo	Descripción
Entrada	$p[0, N-1]$	`array`	Arreglo de números enteros que representa la permutación
Salida	c	`int`	Número de componentes conexos del grafo permutación derivado de p

Precondición:

$$\{Q : (\forall i | 0 \leq i \leq N : \neg(\exists k | 0 \leq k \leq N-1 : p[i] = p[k]) \wedge 1 \leq p[i] \leq N)\}$$

Donde N es el tamaño de la permutación.

Para escribir la precondición nos basamos en la premisa de que para que una permutación exista no pueden haber elementos repetidos y que son números entre 1 y N. A su vez los índices del arreglo empiezan en cero.

Poscondición:

Para el desarrollo de la poscondición es necesario primero definir los siguientes predicados:

- Un grafo G es un conjunto de vértices V y ejes E donde $E \subseteq \{(u, v) | u, v \in V\}$.
- Un camino entre dos vértices es una manera de llegar de un vértice a otro. $path(x, y) \equiv (\exists \{z_0, z_1, ..., z_n\} \in V | : \{(x, z_0), (z_0, z_1), ..., (z_n, y)\} \in E)$
- Un componente conectado S dentro de un grafo puede verse como un "subgrafo" conexo dentro del grafo; de modo que $S \subseteq V$. Y que para todos los vértices dentro del mismo hay al menos un camino. Esto quiere decir que $(\forall u, v \in S | : path(u, v))$.

Luego la poscondición sería: (Teniendo un arreglo A de sets de componentes conectados S)

$$\{R : c = (+i : 0 \leq i \leq |A||(\forall Sset \in A| : S \subseteq G) : 1)\}$$

Es decir c es igual al número de componentes conectados del grafo.

3 Entorno

3.1 Entrada y salida

Como lo especifica el enunciado, la entrada y salida son archivos que deben tener la siguiente forma:

Entrada	Salida
5 1 2 3 2 1 4 3 5 6 1 4 2 5 3 1 3 2 1 6 5 4	3 3 1 1 2

En este archivo la primera línea representa el número de casos de prueba x; y las siguientes x líneas son los casos de prueba, es decir permutaciones del tamaño del número de números de cada línea. En este caso, la estructura de datos de ejemplo es guardada en una lista. Luego por cada permutación se ejecuta e imprime el resultado del algoritmo.

4 Algoritmos implementados

4.1 bfs.py

Para cada permutación en forma de lista, lo primero que se hizo fue que se convirtió la permutación en un grafo, representado por una matriz de adyacencias. Para hacer esto, se inicializa una matriz vacía a la que se le van agregando los ejes mediante el recorrido de la permutación. El recorrido es un recorrido doble, dado que hay que conocer los elementos siguientes para saber si se forma un eje.

Luego, a cada matriz resultante se le aplica el algoritmo Breadth-First Search que determina los componentes conectados.

Para iniciar el Breadth-First Search inicializa una lista de booleanos en false, que guardará la información de los nodos visitados. Además se inicializa la `queue` como estructura de

datos auxiliar para saber el siguiente nodo a recorrer. Se escoge el primer nodo a recorrer y se inicializa la variable en la que se guardaran los componentes conectados. Se inicia el recorrido, que terminará cuando todos los nodos hayan sido recorridos. Primero se revisa que el nodo no haya sido recorrido, si ya fue recorrido se avanza al siguiente, si no, se realiza la parte central del BFS. Se añade el vértice a la cola y se marca que ya fue recorrido. Mientras la cola no esté vacía, se extraerá el primer elemento de la cola y se recorrerán sus vecinos a través de la matriz de adyacencias para ver quienes no han sido recorridos. Si no han sido recorridos, se marcan y se agregan a la cola. Cuando la cola se agota, significa que se acabó un componente conectado y se le suma 1 a nuestra variable contador. Finalmente se retorna la variable contador.

4.1.1 Análisis temporal y espacial

Teniendo en cuenta que el algoritmo se ejecuta x veces donde x es el número de casos de prueba; el análisis temporal se va a realizar solo para un caso. De igual forma, el análisis temporal y espacial hecho está en notación big O; luego la cota es hacia el peor caso.

- **Para la funcion** `makeAdjMatrix()`:

Operación	Constante	Número de repeticiones
Asignación (`=`)	c_1	$4 + n + n^2$
Suma (`+`)	c_2	n^2
`.append()`	c_3	n
Comparación (`!= ==`)	c_4	$n + n^2$

  Luego la fórmula de tiempo es $T(n) = c_1 + c_2 + c_3 + c_4 = 4 + n + n^2 + n^2 + n + n + n^2 = 4 + 3n + 3n^2$, que en notación big o sería $O(n^2)$ para construir el grafo.

  De igual forma, en cuanto a complejidad espacial; estamos utilizando una matriz de adyacencias, luego $S(n) = O(n^2)$ porque por cada nodo (o elemento de la permutación) tenemos su relación (posibilidad de vértice) con todos los otros nodos. También estamos utilizando un stack para hacer el proceso pero este solo puede crecer hasta un valor menor que n luego no es representa un cambio muy grande en la complejidad.

- **Para la funcion** `bfs()`:

  En este caso ya no se va a especificar solo en términos de n sino del número de vértices V (que en este caso es equivalente a n) y el número de ejes E.

Operación	Constante	Número de repeticiones
Asignación (=)	c_1	$4 + n + n^2$
Suma (+)	c_2	$2n$
Comparación (!= ==)	c_4	$3n + 2n^2$
`len()`	c_5	$1 + 2n + n^2$
`.pop()` / `.append()`	c_6	$2n + n^2$

Según Cormen, et.al. el algoritmo de Breadth-First Search en complejidad temporal es de $O(V + E)$ y en complejidad espacial de $O(V)$. (2009). Esta complejidad se da gracias al uso de la `queue`.

Sin embargo, en nuestra implementación, al haber utilizado una matriz de adyacencias, la ecuación de complejidad temporal es del estilo: $T(n) = c_1+c_2+c_3+c_4+c_5+c_6 = 4 + n + n^2 + 2n + 3n + 2n^2 + 1 + 2n + n^2 + 2n + n^2$, luego la complejidad temporal dominante se vuelve $O(n^2)$. Esto, debido a que al ser necesario recorrer todos los ejes, también se debe recorrer la fila de tamaño n que da cuenta de sus vecinos.

Por otro lado, para la complejidad espacial solo se está utilizando un `queue` y una lista con tamaño del número de vértices (o sea n), luego tendría una complejidad de $O(n)$.

4.2 ProblemaP2.py

Este algoritmo funciona teniendo en cuenta la forma en la que se crean componentes conectados.

En principio se parte del supuesto de que hay n componentes conectados, es decir, que ningún vértice tiene un eje con otro. Después, lo que se hace es un algoritmo secuencial para encontrar el máximo (el cual se guarda en la variable `max`) y se compara con el índice del lugar en el que estoy en la lista. Si el máximo es mayor que la posición en donde estoy, debo restarle 1 al número de componentes conectados; ya que esto significa que donde estoy se generó un componente conectado.

4.2.1 Análisis temporal y espacial

Teniendo en cuenta que el algoritmo se ejecuta x veces donde x es el número de casos de prueba; el análisis temporal se va a realizar solo para un caso. De igual forma, el análisis temporal y espacial hecho está en notación big O; luego la cota es hacia el peor caso.

Operación	Constante	Número de repeticiones
Asignación (=)	c_1	$2 + n$
Suma (+)	c_2	n
Resta (-)	c_3	n
Comparación (!= ==)	c_4	n

Luego la fórmula de tiempo es $T(n) = c_1 + c_2 + c_3 + c_4 = 2n + n + n + n = 5n$, que en notación big O sería $O(n)$

De igual forma, en cuanto a complejidad espacial; estamos utilizando solo la lista de la permutación, luego $S(n) = O(n)$.

5 Respuestas a los escenarios de comprensión de problemas algorítmicos

Nota: Los escenarios son independientes entre sí.

1. Se le pide devolver como salida el número mínimo de aristas que se deben añadir para que exista un único componente conectado.

   El reto que presupone este escenario es que ahora el tamaño de la permutación puede cambiar. Los cambios que se le harían al algoritmo serían, primero saber cuántos c componentes conectados se tiene, restarlos con 1 y ese sería el número de aristas que se deben agregar.

2. Se mantiene el problema de obtener los componentes conectados, pero se cambia la instrucción de construcción de aristas así: Una arista es creada para los vértices v_i, v_j $(i < j)$ si y solo si $p_i < p_j$

   El reto que presupone este escenario es que ahora el criterio de comparación es distinto para crear un eje. Los cambios que se le aplicarían al algoritmo son que ahora se debe encontrar el mínimo y comparar con este en vez del máximo; para saber en qué momento se creó un componente conectado.

6 Referencias

Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C., (2009). Breadth-First Search. In *Introduction to algorithms* (3rd ed., pp. 594–597). essay, MIT Press.