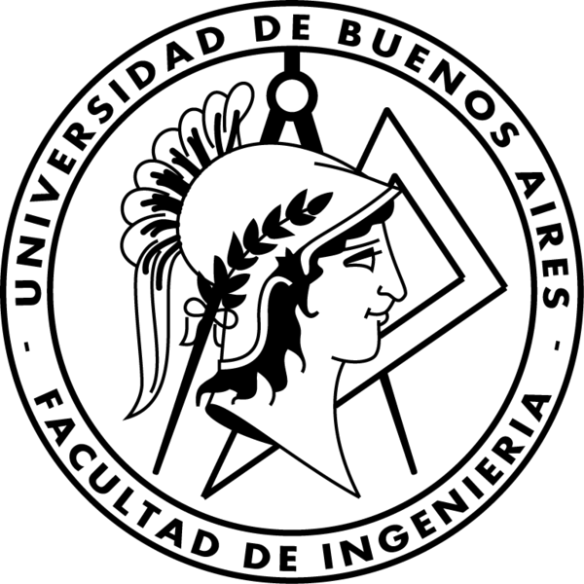
*Teoría de Algoritmos I 75.29*

**Trabajo práctico Nº1**

**

*1º Cuatrimestre 2017*

**Integrantes:**

|  |  |  |
| --- | --- | --- |
| **Apellido y nombre** | **Padrón** | **E-mail** |
| Cassinotti, Damian | 96618 | damiancassinotti.33@gmail.com |
|  |  |  |
|  |  |  |

**Comunidades en redes**

El algoritmo implementado para la resolución de este punto fue el de Kosaraju. Este algoritmo consta de dos recorridos DFS que se explican a continuación.

La función principal del primer recorrido es obtener el orden de descubrimiento de todos los vértices. Este orden será usado en el siguiente recorrido.

El segundo DFS se realiza sobre el grafo transpuesto. En nuestra implementación nos fue más eficiente releer el archivo y generar un nuevo grafo transpuesto, pues esto es O(|E|), mientras que rearmar el grafo transpuesto desde la instancia de grafo original sería O(|V|2), pues se debería recorrer toda la matriz de adyacencias.

El orden de nuestro segundo DFS está dado por el orden inverso sobre el cual se descubrieron los nodos en el paso anterior. De esta forma, cada vez que no se pueda llegar a un nodo a través de otro, se generará una nueva Componente Fuertemente Conexa.

Con respecto al rendimiento del algoritmo, este se basa en lo siguiente: dos lecturas del archivo y dos DFS. Las lecturas son O(|E|), mientras que el DFS es O(|V| + |E|) por lo visto en clase. Por ende, nuestro algoritmo es O(|V| + |E| + |V| + |E| + |E| + |E|) = O(|V| + |E|). En el siguiente gráfico se mostrará el tiempo que tomó cada instancia del problema en relación con su cantidad de vértices:

Para ejecutar este algoritmo, desde la consola de Python (versión 2.7), se importa el archivo kosaraju.py, y se crea una instancia de Kosaraju, que recibe en su constructor el path de los archivos de ejemplo.