Práctica 2

David Alvarado Torres 316167613

Roberto Ocampo Villegas 316293336

4 de noviembre de 2020

Sobre BFS:

Para este algoritmo, tuvimos que agregarle a nuestra implementación de `Nodo` los atributos `padre` e `hijos` y `distancia`. Necesitamos saber que distancia tienen los vértices hasta la raíz para comprobar que al final lo que nos queda sí es un árbol BFS. Sabemos que al incicio todos los procesos menos la raíz tienen distanica ∞. En Pyhton esto se logra simular con el comando `float('inf')`. Sobre la implementación del algoritmo en sí no hay nada que agregar que decir, ya que es un algoritmo muy sencillo.

Sobre DFS:

Para este algoritmo, tuvimos que agregarle a nuestra implementación de `Nodo` los atributos `padre` e `hijos`. Claramente los necesitamos para hacer la construcción del árbol. También hicimos que `vecinos` fuera un conjunto en vez de una lista, pues así es mucho más sencillo realizar las operaciones de diferencia con otros conjuntos, o la de verificar su un conjunto es subconjunto de algún otro.

En cuanto a los mensajes, lo que hicimos fue enviar una lista de tres elementos en el que el primero consiste la descripción del tipo de mensaje. Es decir, esta primer entrada contiene ya sea `"GO"` o `"BACK"`. Esto es importante, pues sabemos que el algoritmo debe ejecutar diferentes instrucciones dependiendo de que tipo de mensaje reciba el proceso. La segunda entrada de la lista contiene al conjunto de nodos `visitados` del remitente, que la necesitamos para saber a que procesos ya no podemos enviar mensajes GO. La última entrada contiene `id_nodo`, que es necesario enviar ya que los nodos siempre adoptan como padre al remitente de un mensaje *GO*.

Para esta práctica se especificó que, debido a la naturaleza no-determinista del algoritmo, lo que debíamos hacer cada que un vértice quisiera enviar un mensaje a uno de sus vecinos, era elegir aquel con el menor id. Es por esto que en nuestra implementación siempre que tomamos un vértice al cual enviarle un mensaje GO tenemos la línea `child = min(self.vecinos.difference(visited))` que nos regresa el elemento mínimo del conjunto $vecinos \setminus visited$.