**Instituto Tecnológico y de Estudios Superiores de Monterrey.**

**PROGRAMACIÓN DE ESTRUCTURA DE DATOS Y ALGORITMOS FUNDAMENTALES.**

****

**Actividad 4.3: Actividad Integral de grafos.**

## Profesor: *Luis Humberto González Guerra.*

**Alumna*: A00831510*** Andrea Marisol Salcedo Vázquez

*Grupo 1*

*21/11/2021*

**Reflexión personal**

-Andrea Marisol Salcedo Vázquez

Durante esta actividad realizamos la implementación de grafos para guardar diferentes puertos y las conexiones que tenían entre ellos. En este caso si un barco podía ir del puerto A al puerto B entonces también podía ir del puerto B al puerto A por lo que se hizo uso de un grafo no dirigido y no ponderado porque el viaje entre puertos valía lo mismo, es decir que si el puerto A estaba conectado al puerto B y al puerto C al barco le “costaba” lo mismo ir a cualquiera de esos dos.

Para este caso en la entrada se nos proporcionaban dos números enteros, n y m, los cuales indicaban la cantidad de puertos y el número de conexiones entre ellos, seguido de los n nombres de los puertos y las m conexiones que constaban de el nombre de dos puertos los cuales son los que estaban conectados. Primero decidimos hacer una función a la que llamamos *loadGraph* y tiene como parámetros una *listAdj* por referencia, un map con llave de *string* y contenido de *int*, *n* y *m.*

En esta función *loadGraph* lo primero que hacemos es realizar un for de n iteraciones y usar el nombre de los puertos como llave en el map para asignarles un número, el primer puerto que entre es el 0, luego el 1, y así sucesivamente. Después, realizamos otro for de m iteraciones para guardar los enlaces en la lista adjunta, primero guardamos los dos nombres en strings para después obtener el número que representa a cada uno y agregarlos a lista adjunta de ambos puertos pues es un grafo no dirigido ni ponderado.

Una vez que se cargaron los datos mediante la función *loadGraph* se recibe un número entero NQ que se refiere a la cantidad de consultas que se realizarán y luego otras NQ líneas con el nombre del puerto de inicio y el máximo número de pasos (MNP) que puede dar y se espera que se regrese la cantidad de puertos que no se pueden visitar con ese MNP. Para esto hicimos uso del método Breadth First Search (BFS) visto en clase pero le realizamos unas modificaciones para poder usarlo según el caso expuesto.

El BFS que desarrollamos recibe como parámetros un entero que representa el nodo inicial, la lista adjunta pasada por referencia y el entero que representa el máximo número de pasos. Primero creamos una queue que está formada por pares que contienen el nodo y el número de pasos a los que se encuentran del puerto inicial y un vector que nos indica el estado en el que se encuentran los nodos, es decir, si ya fueron procesados o no. Primero agregamos a la fila el puerto inicial con un 0 y cambiamos el estado de su nodo a true para marcar que ya fue procesado. Después iniciamos un while mientras la fila no esté vacía pues una vez que esté vacía quiere decir que ya terminamos de visitar los nodos, y dentro de ese while vamos obteniendo el dato y el MNP + 1 que se encuentra al frente y lo vamos eliminarlo para después checar mediante un for los puertos unidos al nodo que acabamos de checar, si estos puertos unidos aun no están procesados, es decir, si su status está en false checamos si los pasos necesarios para llegar a ellos son mayor al MNP y en dicho caso le sumamos uno a la cantidad de nodos no visitados y lo agregamos a la fila para después cambiar su estado a true, en caso de que sea menor o igual solamente lo agregamos a la fila y cambiamos su estado. Luego de esto regresamos la cantidad de nodos no visitados que se obtuvieron en el while.

Una vez que se obtuvo la cantidad de nodos no visitados mediante BFS lo imprimimos en el for y repetimos el proceso para el siguiente caso.

Como pudimos observar los grafos nos ayudan a representar nodos que están unidos entre sí y estos nodos pueden representar muchas cosas, en este caso representaron puertos pero pueden representar ciudades, áreas de un lugar, números, y muchas cosas más, y debido a su estructura para representar nodos unidos entre sí es que debemos conocer los diferentes tipos de algoritmos que son útiles para recorrerlos y para representarlos, esto debido a que no solo podemos representar sus conexiones como una lista adjunta si no que también podíamos hacerlo con una matriz de adyacencias, sin embrago elegimos hacerlo con una lista adjunta por la complejidad que esta representa ya que la mayoría de las veces la complejidad relacionada con la matriz de adyacencias es n^2, mientras que la relacionada con la lista adjunta es |V| + |E| por lo que suele tener menor complejidad esta, pero es necesario conocer ambas formas y las diferencias en cómo manejar cada una puesto que la forma de usarla no es la misma.

Además, debemos conocer los algoritmos de recorrido como DFS y BFS, esto porque algunas veces es más útil usar o modificar alguno de los dos para realizar cierta función, por ejemplo al momento de hacer el topologicalSort en la tarea 4.2 fue útil modificar el algoritmo DFS de forma que pudiera regresar un vector con el orden topológico del grafo, mientras que en esta actividad fue útil modificar el BFS para realizar el recorrido del grafo y contabilizar el número de puertos que no pueden ser visitados con un máximo número de pasos. En este caso fue útil elegir el BFS en lugar del DFS porque el BFS primero visita a todos los vecinos de un nodo y los procesa para después agregar a los vecinos del siguiente nodo y así sucesivamente, en cambio, en el DFS primero se va por todo el camino de un nodo y cuando toca fondo se regresa, también se pudo haber usado este método pero se le tendrían que haber hecho otras modificaciones y probablemente sería más complejo.

Finalmente, podemos observar que la complejidad de nuestro *loadGraph* es de O(n + m) siendo n la cantidad de nodos y m la cantidad de enlaces entre los nodos y debido a que se realizan dos for que dependen de cada uno de estos valores un for se iterará n veces y el otro m veces por lo que las iteraciones totales serían de (n + m) o (V + E). Por otro lado, nuestro BFS tiene una complejidad de O(|V| + |E|) debido al uso de la lista adjunta, en el caso de que hubiéramos usado una matriz de adyacencias esta sería de O(n^2) como se mencionó anteriormente.