Laboratorio 8 | Algoritmos de grafos

Geordie Quiroa

October 18, 2018

1.1 - Complejidad de calcular out-degrees e in-degrees para listas de adyacencia

Las listas de adyacencia relaciona las aristas entre los nodos de un grafo dirgido, incluyen una lista que contiene todos los distintos nodos del grafo y a través de una lista adyacente se presentan las relaciones que existen para cada uno de los nodos del grafo, señalando que existe una relación entre ambos nodos. Debido a esta estructura de datos, encontrar el out-degree (los nodos con los que se relaciona el nodo en la lista de nodos del grafo) presenta una complejidad de $\mathcal{O}(\mathcal{V}+\mathcal{E})$, ya que para encontrar los nodos a los que conecta cada nodo en la lista de nodos del grafo (n = V), hay que recorrer la lista de adyacencia para contar las aristas que existen para cada uno de ellos(n = E). Por lo tanto, el recorrido de cada nodo en la lista de nodos del grafo y el recorrido de la lista de adyacencia para contar las relaciones es (V + E). Por otro lado, la complejidad de calcular el in-degree para cada uno de los grafos es mayor, siendo ésta $\mathcal{O}(\mathcal{V}*\mathcal{E})$, ya que para encontrar la cantidad de veces que un nodo aparece en las relaciones de los otros nodos del grafo es necesario recorrer, para cada uno de los nodos, las listas de adyacencia de los demás nodos que forman parte del grafo.

1.2 - BFS o DFS según los siguientes escenarios

Encontrar más nodos en una red de blockchain
Para este escenario utilizaría BFS, ya que si utilizara DFS existe la posibilidad de recorrer una cadena extremadamente larga, lo que retrasaría significativamente el proceso para encontrar nodos ajenos a la cadena que se está evaluando.

Desarrollar un crawler para un motor de búsqueda
Para este escenario utilizaría BFS, con el fin de recorrer cada uno de los links que contienen las páginas por capas, es decir, no seguir profundizando en los links que contienen las otras páginas que provienen de los links descubiertos en la página anterior hasta concluir el "descubrimiento" de los links de la página anterior, y continuar de la misma forma para las siguientes "capas" de páginas.

Encontrar la salida a un laberinto
Si el laberinto tiene una sola entrada y una sola salida, utilizaría DFS, ya que habría que profundizar únicamente en una dirección, ahora si el laberinto presentara más de una salida, utilizaría BFS para encontrar ambas sin tener que recorrer el mismo camino necesariamente.

Sistema de GPS para encontrar caminos de A a B
Debido a que hay que encontrar más de un camino para el nodo destino (B), utilizaría BFS.

Detectar un ciclo detro de un grafo
Para detectar ciclos dentro de un grafo, utilizaría DFS, ya que la recursión de este algoritmo permite determinar si el nodo B, donde B es el nodo a visitar, en alguna parte de la recursión es igual al nodo inicial (A) o a algún nodo que ya fue visitado recursivamente.

2 - BFS en matrices y listas de adyacencia

Matriz de adyacencia
En el caso de utilizar BFS en una matriz de adyacencia, dicha estructura tendrá como mínimo un tamaño de V x V, siendo V los nodos que pertenecen al grafo, por lo que al recorrer dicha matriz capa por capa para cada una de las relaciones/posiciones en la matriz como lo hace BFS, hace que la complejidad de éste sea $\mathcal{O}(V^2)$.

Lista de adyacencia
Por otro lado, en la utilización de listas de adyacencia, el recorrido de BFS a lo largo de la lista de los distintos nodos (V) que pertenecen al grafo y la lista adyacente de las aristas (E) para cada uno de ellos, da lugar a que la complejidad de éste sea $\mathcal{O}(\mathcal{V}+\mathcal{E})$.

Conclusión
La estructura de datos más eficiente para implementar BFS es la estructura que utiliza listas de adyacencia.

3.1 - Eliminación de recursión en DFS utilizando stacks

Algoritmo DFS modificado

Algorithm 1 DFS with stack

```
1: procedure DFS(G)
2:    stack ← [ ]
3:    for each vertex u ∈ G.V do
4:       u.color ← white
5:     i ← rand vertex ∈ G.V
6:     i.color ← gray
7:     Push to stack ← i
8:    while stack.length != (G.V).length do
9:       for each vertex u ∈ G.V do
10:         if u.color == white then
11:            u.color ← gray
12:            Push to stack ← u
13:            for each vertex v ∈ Adj[u] do
14:               if v.color == white then
15:                  v.color ← gray
16:                  v.parent ← u
17:                  Push to stack ← v
```

3.2 - razón por la que un vértice u termina dentro del dft

La razón por la que un vertice u de un grafo dirigido puede terminar dentro del "depth-first tree" que contenga solo u, aunque u tenga aristas saliendo y entrando en el grafo G, es que este vértice u sea el último en la lista de nodos del grafo y los nodos en la lista de adyacencia para u ya hayan sido descubiertos.