
# Informe: Grafos

**Estudiante:** Jackson Fernando Merma Portocarrero

**Correo:** jmermap@unsa.edu.pe

**Materia:** Laboratorio de Estructura de Datos y Algoritmos

**Escuela:** Ingeniería de Sistemas

**Fecha:** 5 de agosto del 2021

# 1. Estructura *Grafo*

## a. Generalidades

Esta implementación se hizo pensando en el almacenamiento de datos genéticos, esto para la resolución de los problemas propuestos y mejor funcionalidad, además, se implementa dos tipos de encapsulación para la resolución de algunos problemas, esto se indicará más adelante.

## b. Clase Node

Esta clase tiene como objetivo, actuar como la última estructura que encapsula el dato a guardar, además que se le agregan atributos adicionales como un padre(***father***), siguiente(***next***) y su orden(***order***), esto con el objetivo de usarlos en BFS y DFS, donde el padre, se usará con el objetivo de almacenar un dato, mientras que el siguiente, se usará para generar una cola (segunda encapsulación). Además, es necesario recalcar que este nodo se usa para el almacenamiento directo de datos (nodos) en un arreglo estándar, siendo la naturaleza de este, ya que se necesitará extraer un índice específico.

## c. Clase Cola

Esta clase tiene el objetivo de manejar los datos en BFS y DFS de forma enlazada por la fácil implementación y manejo de estructura. Cuenta con dos atributos (***root, last***) que se encargaran de interactuar directamente en la inserción y eliminación de datos; además se implementa un método para buscar un elemento (***search***).

## d. Atributos en el Grafo

Los atributos del grafo son:

- **inf**: representa el valor máximo usando la clase *Integer* y connotado como infinito (usada para la lista de adyacencia).
- **cant y total**: Representan la cantidad de datos introducidos en el grafo(cant) y la cantidad total de nodos a insertar (total)
- **list**: Esta lista genérica de objetos, es la principal que complementará la información al buscar la posición de un dato en la **lista de Adyacencia**.
- **listAdya**: Esta lista de números, representa la lista de adyacencia de todos los grafos (con o sin pesos).
- **bfsList, dfsList y listAuxiliar**: Estas estructuras son colas enlazadas que representan el resultado final (en el caso de **bfs** y **dfs**) de la estructura, mientras que la última es una lista auxiliar usada para procesar **bfs**

## e. Métodos en el Grafo

### i. Relacionar

Este método puede ser usado de dos formas, la primera, donde simplemente hay una relación bilateral (peso 1) entre dos nodos, y la segunda, donde se puede enviar un peso en la arista; entonces, este método simplemente busca la posición correcta en **list** del dato **a** y **b** a relacionar y cambia el valor de **inf**

en la matriz de adyacencia por el peso; luego, es necesario recalcar que si se usa la primera forma de relacionar, se modifica dos posiciones de la tabla de adyacencia (posición ***i-j*** y ***j-i***), pero si se ingresa un peso a la relación, entonces solo cambia en una.

```
167     private void relation(E a, E b, int peso, boolean ambos) {
168         int posA, posB;
169         posA=capturePos(a);
170         posB=capturePos(b);
171
172         //se coloca la relacion en la lista de Adyacencia
173
174         if(posA==-1||posB==-1) {
175             System.out.println("Datos incorrectos");
176         }else {
177             this.listAdya[posA][posB]=peso;
178             if(ambos)
179                 this.listAdya[posB][posA]=peso;
180         }
181     }
```

**Figura 1: Código Java "Método privado *relation*"**

Como se observa en la figura 1, este método privado es el encargado de modificar directamente la estructura, recibiendo dos datos genéricos a relacionar, un peso y un dato booleano que representa la relación bilateral; entonces, se hace un llamado a la función ***capturePos*** (líneas 169-170) para encontrar la posición del arreglo **list** y luego modificarla en la matriz de adyacencia (línea 179).

## ii. BFS

Este método funciona de forma recursiva (método privado), donde recibe un dato genérico (**data**), para indicar desde qué nodo se quiere hacer el BFS.

```
223     public void BFS(E data) {
224         int posAbs = this.capturePos(data);
225
226         if(this.list[posAbs]!=null) {
227             this.bfsList = new Cola();
228             //primero
229             Node aux = new Node(data, null, null, 0);
230
231             this.bfsList.encolar(aux);
232             this.listAuxiliar = new Cola();
233
234             this.BFS(aux, posAbs);
235         }else {
236             System.out.println("No se puede hacer BFS desde un dato que no existe.");
237         }
238     }
```

**Figura 2: Código Java "Método público BFS"**

Como se puede observar en la figura 2, primero se busca una posición válida desde donde se quiere empezar el BFS (línea 224-226), luego en esta primera pasada, se inserta directamente un nodo a la cola principal ***bfsList*** (línea 231), se crea la cola auxiliar e inmediatamente se llama a la recursividad del método private **BFS**.

```
245     private void BFS(Node actual, int pos) {
246         for(int i=0;i<this.total;i++) {
247             if(this.listAdya[pos][i]!=inf&&pos!=i) {
248                 //existe relacion con otro nodo
249                 E dataRelac = this.list[i];//se recupera la pos, y el dato en List
250 
251                 if(listAuxiliar.search(dataRelac)==null&&bfsList.search(dataRelac)==null){
252                     Node aux = new Node(dataRelac, actual, null, actual.order+1);
253                     this.listAuxiliar.encolar(aux);
254                     this.bfsList.encolar(aux);
255                 }
256             }
257         }
258 
259         //se desencola en orden
260         while(!listAuxiliar.isEmpty()) {
261             Node aux = listAuxiliar.desencolar();
262             if(aux!=null) {
263                 int posNode = this.capturePos(aux.data);
264                 this.BFS(aux, posNode);
265             }
266         }
267     }
```

**Figura 3: Código Java "Método privado BFS"**

En la figura 3, se muestra el método directo que genera la cola BFS con los datos correspondientes, para esto se recorre todos los datos relacionados a el nodo enviado (**actual**), y luego se busca este dato en la lista auxiliar y bfs (línea 251), de no estar en ninguna se encola en ambas, esto por la forma natural de realizar un BFS (recorrido por anchura), luego de hacer esto, se desencolara los datos en la lista auxiliar y se llama a la recursividad con este nuevo dato (línea 264).

## iii. DFS

El método público DFS, funciona exactamente igual que en BFS, pero este no trabaja con una lista auxiliar por la forma inherente de procesar un DFS, es por ello que llama a la recursividad una vez hecho el primer paso (insertar al primer dato con el que se quiere empezar el DFS).

```java
private void DFS(Node actual, int pos) {
    for(int i=0;i<this.total;i++) {
        if(this.listAdya[pos][i]!=inf&&pos!=i) {
            //existe relacion con otro nodo
            E dataRelac = this.list[i];//se recupera la pos, y el dato en Lis

            if(dfsList.search(dataRelac)==null){//se verifica solo en dfsList
                Node aux = new Node(dataRelac, actual, null,actual.order+1);
                this.dfsList.encolar(aux);
                this.DFS(aux, i);
            }
        }
    }
}
```

**Figura 4: Código Java "Método privado DFS"**

Este método privado, funciona exactamente qué DFS, con la diferencia en que cuando se encola un dato que no está en la cola principal de *dfsList*, llama inmediatamente a la recursividad durante el ciclo (línea 307).

## iv. Grafo Incluido

Este método estático o de clase, recibe dos estructuras de tipo *grafo* que verifica los datos en el que se busca la inclusión y sus aristas.

```java
public static boolean grafoIncluido(Grafo a, Grafo b) {
    int [][]listAdyaA = a.listAdya, listAdyaB = b.listAdya;
    Comparable []listA = a.list, listB = b.list;

    //determina que grafo tiene menor nodos
    //se recorre todos los Datos incorrectosdatos de A
    for(int i=0;i<a.cant;i++) {
        //se verifica que el dato en (i) exista en el otro grafo

        int posDataB1 = b.capturePos(listA[i]);

        if(posDataB1==-1)//no existe
            return false;

        //se verifica las relaciones del nodo A con las del nodo B
        for(int u=0;u<a.cant;u++) {
            if(listAdyaA[i][u]!=inf) {
                int posDataB2 = b.capturePos(listA[u]);
                if(posDataB2==-1) {
                    return false;
                }else {
                    if(listAdyaB[posDataB1][posDataB2]==inf)
                        return false;
                }
            }
        }
    }
    return true;
}
```

**Figura 5: Código Java "Método *grafoIncluido*"**

Como se puede observar en la figura 5, este método, cumple un rol de buscar 2 tipos de errores, el primero donde el segundo (grafo **b**), no tenga algún nodo del grafo **a** (línea 347), y si cumple este rol de existencia, también verifica que exista las mismas relaciones de un nodo $\mathbf{a_i}$ con $\mathbf{a_j}$ en $\mathbf{b_m}$ con $\mathbf{b_n}$ (línea 357).

# 2. Ejercicios

## a. Ejercicios Propuestos

### i. Ejercicio 4: (parte a)

La esencia del problema indica que existen palabras adyacentes siempre y cuando varien en una sola letra de la misma posicion, y dada la cadena, se puede mostrar el siguiente grafo:

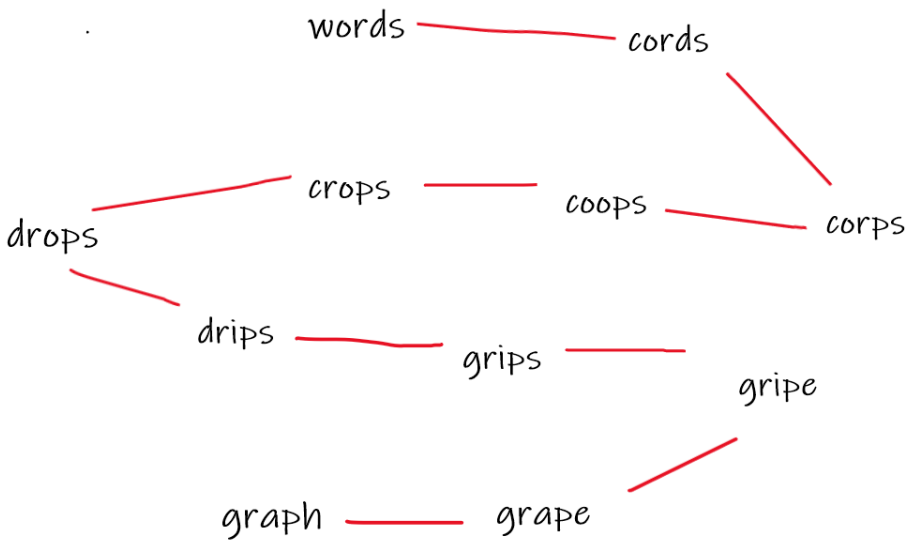


**Figura 6: Grafo 1 de palabras adyacentes**

Entonces, cómo se puede ver en la figura 6, este grafo es una estructura lineal, ya que el nodo *k* es adyacente solo al nodo *k+1*, entonces se puede deducir que con solo 2 nodos, se tiene una figura plana (1 arista para dos nodos), esto indicaría que si se hubiera propuesto al menos 3 palabras adyacentes entre sí, se tendría la siguiente figura:

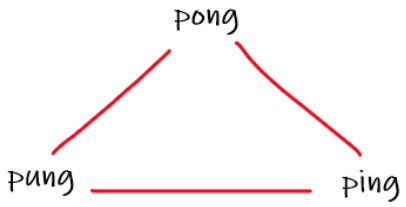


**Figura 7: Grafo 2 de palabras adyacentes**

Como se puede observar, este grafo está totalmente relacionado, y para 4 datos adyacentes se vería de la siguiente forma:


**Figura 8: Grafo 3 de palabras adyacentes**

Entonces, se concluye que este tipo de relaciones (adyacentes) entre palabras, forma un **grafo completo**.

## b. Cuestionario

### i. ¿Cuántas variantes de Dijkstra hay y cuál es la diferencia entre ellas?

Se encontró una variación del algoritmo, donde se trabaja con más 'bordes' o propiedades asociadas al peso, entonces, para solucionar esto, se hace uso de *k* colas de prioridad.

### ii. ¿Qué similitudes, diferencias y casos encuentra entre los algoritmos de caminos mínimos?

Se encontró dos tipos más de algoritmos de ruta más corta:

#### a. Algoritmo de Bellman Ford

La metodología de este algoritmo se basa en encontrar la ruta más corta por nodos que no formen un ciclo, entonces, no es necesario pasar por un mismo vértice, sin embargo se tiene que actualizar todos los nodos conectados con las nuevas distancias.

b. Algoritmo de Floy o Warhall

En este caso, la variante implica trabajar con aristas de pesos positivos y negativos, la ventaja de este algoritmo es que todas las distancias más cortas entre 2 vértices, se puede calcular en $O(V^3)$, donde **V** es el número de vértices en el gráfico.

Entonces, las principales diferencias se encuentran en la aplicación que se le da al algoritmo, ya que trabajan con diferentes metodologías, esto implica decir que Dijkstra se puede usar para casos generales, y siempre y cuando solo se tenga pesos positivos y se busque el camino más corto desde un solo punto de partida, por otro lado, algoritmos como el de Floy, trabajan con pesos negativos y otros algoritmos que sí permiten hallar la distancia más corta entre todos los nodos (árbol mínimo).

# 3. Bibliografía


- Shortest Path Algorithms Tutorials & Notes | Algorithms. (2016, April 25). HackerEarth. https://www.hackerearth.com/practice/algorithms/graphs/shortest-path-algorithms/tutorial/
- Variations of Dijkstra's Algorithm for graphs with two weight properties. (2016, February 16). Stack Overflow. https://stackoverflow.com/questions/35438749/variations-of-dijkstras-algorithm-for-graphs-with-two-weight-properties