**Teoría de Grafos**

**¿Qué es un grafo?**

Un grafo es un conjunto de objetos llamados [vértices](https://es.wikipedia.org/wiki/V%C3%A9rtice_(teor%C3%ADa_de_grafos)) o [nodos](https://es.wikipedia.org/wiki/V%C3%A9rtice_(teor%C3%ADa_de_grafos)) unidos por enlaces llamados [aristas](https://es.wikipedia.org/wiki/Arista_(teor%C3%ADa_de_grafos)) , que permiten representar [relaciones binarias](https://es.wikipedia.org/wiki/Relaci%C3%B3n_binaria) entre elementos de un [conjunto](https://es.wikipedia.org/wiki/Conjunto).

**Grafo bipartito**

Un grafo G es bipartito si puede expresarse como G = { V1 U V2, A } (es decir, sus vértices son la unión de dos grupos de vértices), bajo las siguientes condiciones:

• V1 y V2 son disjuntos y no vacíos.

• Cada arista de A une un vértice de V1 con uno de V2.

• No existen aristas uniendo dos elementos de V1; análogamente para V2.

Bajo estas condiciones, el grafo se considera bipartito, y puede describirse informalmente como el grafo que une o relaciona dos conjuntos de elementos diferentes, como aquellos resultantes de los ejercicios y puzzles en los que debe unirse un elemento de la columna A con un elemento de la columna B.

**¿Cómo chequear si un grafo es bipartito?**

Podemos chequearlo aplicando manualmente el algoritmo de coloreo de grafos. Si cuando terminamos de colorear todos los nodos tenemos en total 2 colores presentes, entonces, el grafo es bipartito.

Los pasos, que a esta altura no deberían ser necesarios enunciar, son:

1) Colorear cualquier nodo de color rojo.

2) Para todos los nodos adyacentes sin pintar, colorearlos de un color azul.

3) Para todos los nodos adyacentes sin pintar, colorearlos de rojo.

4) Repetir pasos 2 y 3.

5) Si encontramos dos nodos adyacentes del mismo color, entonces el grafo no es bipartito. De otra forma, sí lo es.

6) Si el grafo es bipartito, el algoritmo de coloreo habrá creado dos conjuntos de nodos (rojos y azules, respectivamente).

**Grafos multipartitos**

Son grafos en los cuales sus nodos pueden ser divididos en N conjuntos diferentes. Un grafo N-partito tendrá tantos colores como conjuntos de nodos. El proceso para chequear si un grafo es N-partito es el mismo que para un grafo bipartito, se aplica el algoritmo de coloreo y se observa el resultado.

**Grafo regular**

Un grafo regular es un [grafo](https://es.wikipedia.org/wiki/Grafo) donde cada vértice tiene el mismo grado. Un grafo regular con vértices de grado *k* es llamado grafo *k*-regular o grafo regular de grado k.

Para que un grafo sea de grado 1, la cantidad de nodos debe ser par.

Para que un grafo sea de grado 2, la cantidad de nodos puede ser par o impar.

En definitiva, si el grado es impar, la cantidad de nodos es par. Y si el grado es par, la cantidad de nodos puede ser par o impar.

El grado de los nodos puede ser a lo sumo cantNodos-1.

**Ciclos y recorridos de grafos**

La palabra ciclo se emplea en [teoría de grafos](https://es.wikipedia.org/wiki/Teor%C3%ADa_de_grafos) para indicar un **camino cerrado** en un grafo, es decir, en que el nodo de inicio y el nodo final son el mismo.

**Ciclo euleriano**

En la teoría de grafos, un camino euleriano es un camino que pasa por cada arista una y solo una vez. Un ciclo o circuito euleriano es un camino cerrado que recorre cada arista exactamente una vez. Si un grafo admite un ciclo euleriano, se denomina grafo euleriano.

* Sea G un grafo o multigrafo no dirigido. Entonces G tiene un ciclo de Euler si, y solo si, es conexo y todo vértice tiene grado par.
* Sea G un grafo o multigrafo no dirigido. Entonces G tiene un camino de Euler si, y solo si, es conexo y tiene solo dos vértices de grado impar.

**Camino o ciclo Hamiltoniano**

Un camino hamiltoniano es aquel que visita todos los vértices del grafo una sola vez. Si además el último vértice visitado es adyacente al primero, el camino es un ciclo hamiltoniano. [El problema de encontrar un ciclo (o camino) hamiltoniano en un grafo arbitrario](https://en.wikipedia.org/wiki/Hamiltonian_path_problem) se sabe que es [NP-completo](https://es.wikipedia.org/wiki/NP-completo).

* Un grafo hamiltoniano ha de ser conexo.
* Un grafo hamiltoniano no puede tener vértices de grado 1: en todos los vértices deben incidir al menos dos aristas, la de “entrada” y la de “salida”.
* Si *S* es un subconjunto del conjunto de vértices de un grafo *G*, escribimos *G − S* para designar el subgrafo que aparece al eliminar todos los vértices de *S* y todas las aristas adyacentes a los vértices de *S*.

**Algoritmo de Dijkstra**La finalidad del algoritmo de Dijkstra es la de darnos los caminos más cortos desde un nodo hasta el resto de los nodos (o bien, hasta un nodo en particular).

¿Qué cosas necesitamos para programar el algoritmo de Dijkstra?

* La representación de un grafo en forma de matriz o lista de adyacencia.
* Un vector de distancias (puede ser integer, double, lo que sea conveniente).
* Un vector booleano para identificar a los nodos visitados y no visitados.

**Pasos del algoritmo de Dijkstra:**

1. Marcar todos los nodos como no visitados. Es decir, inicializar el vector de nodos visitados en falso.
2. Asignar a cada nodo del vector de distancias un valor tentativo: para el nodo inicial, pondremos un 0, para todos los demás nodos, pondremos infinito.
3. Para el nodo actual en el que estamos parados, revisaremos todos los nodos adyacentes que no hayan sido visitados ya. Calcularemos la distancia tentativa a ese nodo adyacente pasando por el actual, y en caso de ser menor, actualizaremos la distancia guardada en el vector.
4. Cuando terminamos de recorrer todos los adyacentes, marcamos el nodo actual como visitado.
5. Si el nodo destino ya fue visitado, podemos parar el algoritmo.
6. Si no, seguimos y buscamos otro nodo actual tomando aquel que tenga la menor distancia en el vector de distancias.

El cálculo de la distancia se hace a través de la siguiente fórmula:

Donde i es el nodo adyacente que estamos viendo, W es el nodo en el que estamos parados, y C[w,i] es el peso de la arista que conecta W con su adyacente i.

**¿Cómo obtenemos el camino de nodos a recorrer?**

Si bien Dijkstra nos dice la distancia mínima, no nos dice cuáles son aquellos nodos que debemos recorrer para llegar al nodo X con esa distancia mínima.

Sin embargo, podemos obtener fácilmente dicho camino utilizando un vector de predecesores. El vector de predecesores será inicializado con el número del nodo inicial, y luego, cada vez que una distancia sea modificada, será actualizado de la siguiente forma:

Lo que estamos diciendo al hacer eso es: “para llegar a este nodoAdy con la distancia mínima, antes tuve que pararme en este nodoActual”.

Luego, podremos imprimir el camino mínimo desde el nodoInicial hasta un nodo determinado de la siguiente forma:

**public** void getCamino(**int** nodoHasta) {

**int** i = nodoHasta;

**while** (i != nodoInicial) {

System.***out***.println("Pasamos por el nodo: " + i);

i = predecesores[i];

}

}

**Algoritmo de Kruskal**

El algoritmo de Kruskal es un [algoritmo](https://es.wikipedia.org/wiki/Algoritmo) para encontrar un [árbol recubridor mínimo](https://es.wikipedia.org/wiki/%C3%81rbol_recubridor_m%C3%ADnimo) en un grafo conexo y ponderado. Es decir, busca un subconjunto de aristas que, formando un árbol, incluyen todos los vértices y donde el valor de la suma de todas las aristas del árbol es el mínimo.

¿Qué cosas necesitamos para programar el algoritmo de Kruskal?

* Una cola de prioridad (preferiblemente, la implementación que Java mismo ofrece).
* Una clase arista (comparable).
* Un vector de padres (indicará los padres de cada nodo).
* Un ArrayList de aristas para guardar al árbol recubridor mínimo.

Hay que tener tres consideraciones en cuenta para el algoritmo de Kruskal:

1. Las nuevas aristas añadidas no deben formar un ciclo. Un árbol, por definición, no posee ciclos.
2. El algoritmo finaliza una vez que la cantidad de nodos procesados sea igual a N-1 (siendo N la cantidad de nodos del grafo.
3. Kruskal divide a los nodos en diferentes conjuntos, separados los unos de los otros. A medida que va añadiendo las aristas de menor peso, realiza uniones entre los conjuntos. La idea principal del algoritmo es evitar añadir aristas que realicen una unión dentro de un conjunto que ya está unido. Todas las aristas que se añadan serán aristas que conecten dos conjuntos disjuntos.

**Pasos del algoritmo de Kruskal:**

1. Ordenar todas las aristas del grafo de menor a mayor. Insertar las aristas en una cola de prioridad mínima (que sea mínima quiere decir que ordena de menor a mayor) hace que las mismas sean ordenadas automáticamente al momento de insertarlas.
2. Inicializamos el vector de padres utilizando una función llamada **makeSet**. **makeSet** simplemente se encarga de dividir los nodos en conjuntos disjuntos diciendo que el padre de cada nodo es él mismo.
3. Mientras que la cantidad de nodos procesados sea menor que la cantidad de nodos menos uno…
4. Seleccionamos la arista de menor peso (es decir, la removemos de la cola de prioridad). Verificamos utilizando la función **find** si la arista forma o no un ciclo con el árbol recubridor mínimo. En caso de no formar un ciclo, añadimos nuestra arista al ArrayList, aumentamos en uno la cantidad de nodos procesados, y ejecutamos la función **union** para unir los conjuntos.
5. Repetir paso 3 y 4 hasta que se cumpla la condición.

**¿Cómo funcionan makeSet, find, y union?**

**makeSet:** es la más fácil de todas las funciones. Simplemente se encarga de correr el vector de padres e indicar que el padre de cada nodo es sí mismo. Con esto estamos creando cantNodos conjuntos disjuntos que luego usaremos para ejecutar el algoritmo.

**public** **void** makeSet(**int** padre[]) {

**for** (**int** i = 0; i < padre.length; i++) {

padre[i] = i;

}

}

**find:** dado un nodo y el vector de padres, busca recursivamente al padre del nodo. Termina su recursividad cuando se halla un nodo en el cual el padre sea sí mismo.

**public** **int** find(**int** padre[], **int** nodo) {

**if** (padre[nodo] != nodo) {

**return** find(padre, padre[nodo]);

}

**return** nodo;

}

**union:** dado el vector de padres, y dos nodos X e Y, busca a los padre de ambos nodos y, si ambos son diferentes, hace que el padre del nodo Y sea igual que el padre del nodo X. En definitiva, realiza la unión de dos conjuntos disjuntos.

**public** **void** union(**int** padre[], **int** x, **int** y) {

x = find(padre, x);

y = find(padre, y);

//Si las raíces son diferentes, hacemos que el padre de Y sea X

**if** (x != y) {

padre[y] = x;

}

}

**Algoritmo de Prim**

El algoritmo de Prim es un [algoritmo](https://es.wikipedia.org/wiki/Algoritmo) para encontrar un [árbol recubridor mínimo](https://es.wikipedia.org/wiki/%C3%81rbol_recubridor_m%C3%ADnimo) en un [grafo](https://es.wikipedia.org/wiki/Grafo) [conexo](https://es.wikipedia.org/wiki/Teor%C3%ADa_de_los_grafos#Grafos_conexos), no dirigido y cuyas [aristas](https://es.wikipedia.org/wiki/Arista_(Teor%C3%ADa_de_grafos)) están ponderadas.

¿Qué cosas necesitamos para programar el algoritmo de Prim?

* Una cola de prioridad (preferiblemente, la implementación que Java mismo ofrece).
* Una clase arista (comparable).
* Un vector booleano de nodos visitados.
* Un contador de nodos visitados.
* Un acumulador de costos.

**Pasos del algoritmo de Prim:**

1. Marcamos el nodo inicial como visitado. Aumentamos el contador de visitados en 1. Pero más importante, añadimos todas las aristas adyacentes al nodo inicial en la cola de prioridad. Haciendo esto, nos aseguramos de quitar la menor de las aristas en cada pasada.
2. Mientras que la cola de prioridad no esté vacía, y la cantidad de nodos visitados sea diferente a la cantidad de nodos totales…
3. Seleccionamos la menor de las aristas (removiéndola de la cola de prioridad), y preguntamos si ya visitamos al nodo destino de dicha arista. Si ya lo hemos visitado, significa que ya encontramos anteriormente una arista de costo menor hacia ese nodo, por lo que descartamos esta.
4. Si no visitamos al nodo destino, sumamos al acumulador de costos el costo de dicha arista, incrementamos el contador de nodos visitados, y marcamos como visitado al nodo destino. Pero más importante aún, añadimos a la cola de prioridad todas las aristas adyacentes de dicho nodo destino.
5. Retornamos el MST, o el costo. Esto depende ya de la implementación.

**Algoritmo BFS (Búsqueda por Anchura)**El algoritmo de BFS es un algoritmo de recorrido. Su función principal es la de recorrer todos los nodos de un grafo. Si bien es un algoritmo simple, en muchas ocasiones un algoritmo de recorrido puede sernos útil para resolver un problema de grafos.

¿Qué cosas necesitamos para programar el algoritmo de BFS?

* Un vector booleano de nodos visitados.
* Una cola (queue, preferiblemente la implementación de Java).
* Opcionalmente podemos tener un vector de padre o de distancias relativas. Esto varía dependiendo de lo que se quiera lograr con la implementación del algoritmo.

¿Cómo es el recorrido realizado por BFS?

El algoritmo de búsqueda por anchura realiza el siguiente recorrido: posicionándose en un nodo inicial, visita y encola los nodos adyacentes al mismo. Una vez que recorre todos los adyacentes, desencola el siguiente nodo de la cola (que será el primer adyacente visitado). Se vuelve a repetir la misma operación, esta vez visitando los nodos adyacentes del nodo actual, y acolándonos en la cola.

**Pasos del algoritmo de BFS:**

1. Marcamos el nodo inicial como visitado, y lo encolamos.
2. Desencolamos el primer nodo de la cola.
3. Recorremos todos los nodos adyacentes al nodo actual, marcándolos como visitados, y añadiéndolos a la cola. Es acá donde podemos hacer otras cosas y no solamente “recorrer” los nodos.
4. Repetir pasos 2 y 3 hasta que ya no haya más nodos por recorrer.

**Algoritmo DFS (Búsqueda por Profundidad)**No hay mucho que decir. Hace lo mismo que cualquier otro algoritmo de recorrido.

¿Qué cosas necesitamos para programar el algoritmo de DFS?

* Un vector booleano de nodos visitados.
* Una pila (stack, preferiblemente la implementación de Java).
* Opcionalmente podemos tener un vector de padre o de distancias relativas. Esto varía dependiendo de lo que se quiera lograr con la implementación del algoritmo.

¿Cómo es el recorrido realizado por DFS?

El algoritmo de búsqueda por profundidad realiza el siguiente recorrido: partiendo desde un nodo inicial, intenta llegar a lo más profundo que pueda siguiendo una sola “rama”. Una vez que llegue al final de una rama, vuelve al nodo de donde partió, y comienza nuevamente a recorrer otra rama en su profundidad.

**Pasos del algoritmo de DFS:**

1. Marcamos el nodo inicial como visitado, y lo apilamos.
2. Desapilamos el primer nodo de la cola.
3. Recorremos todos los nodos adyacentes al nodo actual, marcándolos como visitados, y añadiéndolos a la pila. Es acá donde podemos hacer otras cosas y no solamente “recorrer” los nodos.
4. Repetir pasos 2 y 3 hasta que ya no haya más nodos por recorrer.

**Algoritmo de Floyd**

El algoritmo de Floyd es similar al de Dijkstra en el sentido de que ambos hallan la distancia mínima entre los nodos. Sin embargo, Floyd se encarga de hallar la distancia mínima de todos los nodos hacia todos los nodos. Es decir, no se centra en la distancia mínima de un único nodo hacia todos los demás.

El algoritmo de Floyd “crea” una matriz nueva una cantidad K de veces, siendo K el número de nodos. Además, por cada matriz, recorre todos los nodos, y todos los adyacentes de los nodos. Esto da una complejidad cúbica.

¿Qué cosas necesitamos para programar el algoritmo de Floyd?

* Una matriz copia de la matriz de adyacencia original, que utilizaremos como matriz de distancias.

**Pasos del algoritmo de Floyd:**

1. Clonamos o copiamos la matriz de adyacencia original.
2. Inicializamos la diagonal principal de esta matriz en 0 (no es necesario si ya lo hicimos de antemano al cargar la matriz de adyacencia original).
3. Comenzamos contando desde un K = 0.
4. Tomamos el primer de los nodos, y recorremos el resto de los nodos del grafo.
5. En cada nodo que recorremos, preguntamos si la distancia pasando por el nodo K es menor que la distancia desde el nodo actual hasta dicho nodo. En caso de ser así, actualizamos la distancia en la matriz.
6. Repetimos el paso 3 y 4 por todos los nodos que haya.
7. Incrementamos K y repetimos el paso 2 hasta haber recorrido K veces, siendo K la cantidad de nodos.

La pregunta que se hace el algoritmo de Floyd es la siguiente:

¿Es la distancia pasando por el nodo K, menor que la distancia entre i y j?

**Algoritmo de Warshall**

El algoritmo de Warshall se encarga de crear una matriz de clausura transitiva. Lo que nos dice esta matriz son todas las posibles uniones que existan entre los nodos. En términos simples, nos indica qué nodos están conectados entre sí, a través de caminos directos o indirectos.

¿Qué cosas necesitamos para programar el algoritmo de Warshall?

* Una matriz booleana.

**Pasos del algoritmo de Warshall:**

1. Inicializamos la matriz de booleana recorriendo la matriz de adyacencia original. Lo que hacemos es recorrer nodo por nodo, y asignar un valor **true** a cada uno de sus adyacentes. De esta forma marcamos que hay una conexión entre ambos.
2. Comenzamos contando desde un K = 0.
3. Tomamos el primer de los nodos, y recorremos el resto de los nodos del grafo.
4. En cada nodo que recorremos, preguntamos si existe una conexión directa entre el nodo actual y el nodo que estamos viendo, o si existe una conexión indirecta pasando a través del nodo K.
5. Repetimos el paso 3 y 4 por todos los nodos que haya.
6. Incrementamos K y repetimos el paso 2 hasta haber recorrido K veces, siendo K la cantidad de nodos.

La pregunta que se hace el algoritmo de Warshall es la siguiente:

¿Existe una conexión directa entre los nodos i y j, o una conexión indirecta entre i y j, pasando por k?

**Algoritmo de Coloreo de Grafos**

El algoritmo de coloreo de grafos intenta averiguar el número cromático de un grafo, que es el menor número de colores necesarios para colorear un grafo.

Un grafo que puede ser asignada una k-coloración (propia) es k-coloreable y es k-cromático si su número cromático es exactamente k. Una k-coloración es lo mismo que una partición del conjunto de vértices en k conjuntos independientes, y los términos k-partito y k-coloreable tienen el mismo significado.

De todas formas, no existe un algoritmo de coloreo de grafos que nos dé el número cromático con total certeza (salvo en casos simples). Existen diferentes variantes y formas de obtener una aproximación al número cromático real.

**Algoritmo de coloración secuencial básico**

El orden en que se colorean los vértices se decide antes de que se empiece a colorearlos. Dada una ordenación de los vértices del grafo, los algoritmos secuenciales asignan el mínimo color posible al siguiente vértice. Es decir, si queremos colorear v, teniendo ordenados numéricamente los colores, asignamos a v el color más pequeño que no aparece entre los asignados a los vecinos de v ya coloreados.

**Algoritmo de coloración de Welsh y Powell**

Es una variante del algoritmo de coloración secuencial básico, también conocida como "Primero el de mayor grado". Es debida a Welsh y Powell, y en este algoritmo, los vértices se ordenan inicialmente de acuerdo a sus grados. Es decir, ordenamos de forma que d(v1) >= d(v2) >= ... >= d(vn).

**Algoritmo de coloración de Matula, Marble, Isaacson**

Es una variante del algoritmo de coloración secuencial básico, también conocida como "El de menor grado el último". Se debe a Marble, Matula e Isaacson, y en este algoritmo, los vértices se ordenan en orden inverso. Primero se elige vn como el vértice de menor grado, luego se elige vn-1 como el vértice de menor grado en G-{vn}, y así se continúa recursivamente, examinando los vértices de menor grado y eliminándolos del grafo.

**Algoritmo de coloración al azar**

No se sigue un orden en específico, los nodos son tomados al azar hasta que el grafo queda completamente coloreado.

En la cátedra nos recomiendan usar el algoritmo de Welsh-Powell.

¿Qué cosas necesitamos para programar el algoritmo de coloración de Welsh-Powell?

* Un ArrayList de clase Nodo, que guardará los nodos de la matriz con sus respectivos grados.
* Un contador de colores. Opcionalmente, también se podría tener un ArrayList de clase Color, en caso de que queramos saber los colores que se usaron.
* Un booleano que utilizaremos como flag.
* Si estamos interesados en saber la cantidad máxima de colores (o la cantidad mínima), deberemos tener en alguna parte una variable que, por cada pasada que le hagamos al algoritmo, guarde el (máximo o mínimo) valor obtenido.

**Pasos del algoritmo de coloración de Welsh-Powell:**

1. Necesitamos hallar los grados de cada uno de los nodos. Para esto, recorremos nodo por nodo en la matriz de adyacencia, y contamos la cantidad de nodos adyacentes que poseen. Una vez terminado de recorrer sus adyacentes, creamos un nuevo nodo y lo añadimos a la lista.
2. Ordenamos la lista de mayor a menor.
3. Tomamos el primer nodo de la lista.

Es un bardo esto. Que lo explique otro. O vean el código. O el libro ese de estructura de datos.

**Complejidad Computacional de los algoritmos**

*Teniendo en cuenta que se usa una matriz de adyacencia.*

|  |  |
| --- | --- |
| **Algoritmo** | **Complejidad** |
| Dijkstra | N2 |
| Prim | N2 |
| Kruskal | N2 |
| BFS | N2 |
| DFS | N2 |
| Floyd | N3 |
| Warshall | N3 |
| Coloreo | N2/ N3 |

El algoritmo de coloreo, dependiendo de cómo lo encares, tiene más o menos complejidad. Yo apunto a que es N2.