Universidad Central de Venezuela
Facultad de Ciencias
Escuela de Computación

Lecturas en Ciencias de la Computación
ISSN 1316-6239

Algoritmos Básicos de Grafos

Ernesto Coto

ND 2003-02

Laboratorio de Computación Gráfica

Febrero, 2003

Algoritmos Básicos de Grafos

Ernesto Coto
ecoto@strix.ciens.ucv.ve
ecoto@opalo.ciens.ucv.ve

Universidad Central de Venezuela. Facultad de Ciencias. Escuela de Computación.
Laboratorio de Computación Gráfica (LCG)
Venezuela. Caracas
Apdo. 47002, 1041-A.

ND 2003-02

Resumen

Los grafos son solo abstracciones matemáticas, pero son útiles en la práctica porque nos ayudan a resolver numerosos problemas importantes. Estas notas describen algunos de los algoritmos y métodos más conocidos para resolver problemas de procesamiento de grafos, así como proponen alternativas para la representación de los mismos en el computador. Cada algoritmo o método está acompañado de figuras que ilustran su funcionamiento y de una breve descripción teórica que incluye el estudio de la complejidad en tiempo del mismo. En adición se incluye la descripción de una taxonomía de clasificación de problemas de procesamiento de grafos, basada en el grado de dificultad de lo mismos.

Estas notas abarcan desde problemas simples como el de recorrido de grafos, hasta problemas más complejos como los de flujo en redes. Es recomendable que el lector tenga conocimientos básicos de estructuras de datos dinámicas y de teoría de grafos.

Palabras Claves: Grafos. Algoritmia. Algoritmos de Grafos. Procesamiento de Grafos. Estructuras de Datos para Grafos.

Febrero, 2003

CONTENIDO

1. CONCEPTOS BASICOS

Hablando intuitivamente, un *grafo* es un conjunto de nodos unidos por un conjunto de líneas o flechas. Por lo general, los nodos son entes de procesamiento o estructuras que contienen algún tipo de información y las líneas o flechas son conexiones o relaciones entre estos entes. Si se utilizan flechas para conectar los nodos decimos que el grafo es *dirigido* (también llamado *digrafo*) porque las relaciones entre los nodos tienen una dirección. En caso contrario el grafo es *no dirigido*. En cualquiera de los dos casos, bien sea que se utilicen líneas o flechas, a estas relaciones se les puede llamar simplemente *aristas*. Frecuentemente las aristas también tienen algún tipo de información asociada (distancia, costo, confiabilidad, etc.), en cuyo caso estamos en presencia de un *grafo pesado*.

Las secuencias de aristas forman *caminos* o *ciclos*. Un ciclo es un camino que termina en el mismo nodo donde comenzó. Si el camino recorre todos los nodos del grafo es llamado *tour*. El número de aristas en un camino es la *longitud del camino*.

Se dice que un grafo es *conexo* si se puede llegar desde cualquier nodo hasta cualquier otro mediante un camino. De lo contrario no es conexo, pero puede dividirse en *componentes conexas*, que son subconjuntos de nodos y aristas del grafo original que si son conexos. Un grafo conexo sin ciclos es llamado un *árbol*.

Estos son apenas unos cuantos conceptos de lo que se conoce como la *Teoría de Grafos*. El objetivo de estas notas no es cubrir por completo dicha teoría sino enfocarnos en la implementación de este tipo de estructuras y las operaciones y algoritmos más comunes e importantes que se aplican sobre las mismas.

2. REPRESENTACION EN EL COMPUTADOR

Hay por lo menos dos maneras evidentes de representar un grafo en un computador, utilizando la notación pseudoformal propuesta en **[2]**. La primera utiliza lo que se conoce como una matriz de adyacencia, a saber:

```
CLASE Grafo
      Privado:
            Arreglo de <Tipo> Nodos de [1..N];
            Arreglo de logico MatrizDeAdyacencia de [1..N][1..N];
      Publico:
            # Todas las operaciones aquí
FCLASE
```

Un valor verdadero en la posición *(i,j)* de la matriz indica que hay una arista que conecta al nodo *i* con el nodo *j*. Para representar un grafo pesado se puede cambiar la matriz de adyacencia de lógicos a una matriz de registros, siendo el peso un campo del registro.

Esta representación es muy útil cuando se cumplen dos condiciones principales. Primero, si se conoce el número exacto de nodos en el grafo no hace falta utilizar estructuras dinámicas porque las estáticas se pueden acceder con mayor facilidad. Segundo, la matriz de adyacencia no contiene un gran número de elementos en `FALSO`.

Si no se cumple la primera de estas condiciones y el número de nodos en el grafo puede variar drásticamente entonces se justifica el uso de estructuras dinámicas. En este caso, lo ideal es utilizar una lista lineal de nodos conteniendo la información en <Tipo>. Por supuesto, si el número de nodos va a cambiar entonces tampoco se justificas tener un matriz estáticas, por lo que debería utilizarse una matriz esparcida.

```
CLASE Nodo              #Nodo de la lista de <Tipo>
      Publico:
         entero Id;
         <Tipo> Info;
      ↑Nodo proximo;
      #Operaciones aquí
FCLASE

#La declaración de los nodos para las columnas (NodoC), filas (NodoF) y
#nodos internos de la matriz van aquí. Son las declaraciones de una
#matriz esparcida implementada con listas con posición lógica fija o
#relativa.

CLASE Grafo
      Privado:
         ↑Nodo    primerNodo;       # 1er. nodo de la lista
         ↑NodoF   primeraFila;      # 1era. fila de la matriz esparcida
         ↑NodoC   primeraColumna;   # 1era. columna de la matriz esparcida
      Publico:
         # Todas las operaciones aquí
FCLASE
```

Sin embargo, puede que esta solución no sea la más conveniente por el hecho de que la matriz esparcida ocupará más memoria que su contraparte estática a medida que el número de conexiones entre los nodos sea mayor, ya que habrá pocos elementos no nulos dentro de la matriz. En ese caso, se debe utilizar otra implementación.

```
CLASE Nodo              #Nodo de la lista de <Tipo>
      Publico:
            entero Id;
            <Tipo> Info;
            ↑Nodo proximo;
            ↑Ady ListaDeAdyacentes;
            #Operaciones aquí
FCLASE

CLASE Ady               #Nodo de la lista de nodos adyacentes
      Publico:
            ↑Nodo AdyAeste;
            ↑Ady proximo;
            #Operaciones aquí
FCLASE

CLASE Grafo
      Privado:
            ↑Nodo primerNodo;       # 1er. nodo de la lista
      Publico:
```

```
            # Todas las operaciones aquí
FCLASE
```

Esta representación ocupa menos memoria que la anterior sin importar si la matriz esparcida esta muy llena o no, pero puede incrementar la complejidad de algunas operaciones, como por ejemplo saber cuales nodos son adyacentes a un nodo en particular (en caso de un grafo dirigido).

De nuevo, si las aristas también tienen algún tipo de información asociada bastaría una leve modificación en la clase `Ady` estructura para agregarla.

Por supuesto, se deben implementar las operaciones más comunes como agregar un nodo al grafo, eliminar un nodo del grafo y conectar un nodo con otro. En adición, se puede implementar todo tipo de operaciones basadas en la teoría de grafos, para resolver los problemas que se describen a continuación.

3. PROBLEMAS DE PROCESAMIENTO DE GRAFOS

Los algoritmos que se tratan en este texto son fundamentales y son muy útiles en muchas aplicaciones, pero solamente son una introducción al tema de algoritmos de grafos. Existe un gran variedad de problemas relacionados con grafos y una gran variedad de algoritmos para procesamiento de grafos, pero claramente no todo problema de grafos es sencillo de resolver y en muchas ocasiones tampoco es sencillo determinar que tan difícil puede ser resolverlo.

En **[5]** podemos encontrar un categorización de problemas de grafos de acuerdo al grado de dificultad para resolverlos, a saber:

- *Fáciles*: Un problema fácil de procesamiento de grafos es aquel que se puede resolver utilizando un programa eficiente y elegante. Frecuentemente su tiempo de ejecución es lineal en el peor caso, o limitado por un polinomio de bajo grado en el número de nodos o el número de aristas.

  Generalmente, también podemos decir que el problema es fácil si podemos desarrollar un algoritmos de fuerza bruta que aunque sea lento para grandes grafos, es útil para grafos pequeños e inclusive de tamaño medio. Entonces, una vez que sabemos que el problema es fácil, buscamos soluciones eficientes y escogemos la mejor de ellas.

- *Tratable*: Un problema tratable de procesamiento de grafos es aquel para el que se conoce un algoritmo que garantiza que sus requerimientos en tiempo y espacio están limitados por una función polinomial en el tamaño del grafo (número de nodos + número de aristas).

  Todo problema fácil es tratable, pero se hace la distinción debido a que el desarrollo de una solución eficiente para resolverlo es extremadamente difícil o imposible. Las soluciones a algunos problemas intratables nunca han sido

escritas en programas, o tiempo tiempos de ejecución tan altos que no puede contemplarse su utilización en la práctica.

- *Intratable*: Un problema intratable de procesamiento de grafos es aquel para el que no se conoce algoritmo que garantice obtener un solución del problema en una cantidad razonable de tiempo. Muchos de estos problemas tienen la característica de que podemos utilizar un método de fuerza bruta para probar todas las posibilidades de calcular la solución, y se consideran intratables porque existen demasiadas posibilidades a considerar.

  Esta clase de problemas es extensa y muchos expertos piensan que no existen algoritmos eficientes para solucionar estos problemas. El término `NP-hard` describe los problemas de esta clase, el cual representa un altísimo nivel de dificultad. En **[5]** se puede estudiar el significado de este término.

- *Desconocida*: Existen problemas de procesamiento de grafos cuya dificultad es desconocida. No hay un algoritmo eficiente conocido para resolverlos, ni son conocidos como `NP-hard`. El problema de isomorfismo de grafos pertenece a esta clase.

Algunos de los problemas más conocidos de procesamiento de grafos son:

- *Conectividad Simple:* Consiste en estudiar si el grafo es conexo, es decir, si existe al menos un camino entre cada par de vértices.

- *Detección de Ciclos:* Consiste en estudiar la existencia de al menos un ciclo en el grafo

- *Camino Simple:* Consiste en estudiar la existencia de un camino entre dos vértices cualquiera.

- *Camino de Euler:* Consiste en estudiar la existencia de un camino que conecte dos vértices dados usando cada arista del grafo exactamente una sola vez. Si el camino tiene como inicio y final el mismo vértice, entonces se desea encontrar un *tour de Euler*.

- *Camino de Hamilton:* Consiste en estudiar la existencia de un camino que conecte dos vértices dados que visite cada nodo del grafo exactamente una vez. Si el camino tiene como inicio y final el mismo vértice, entonces se desea encontrar un *tour de Hamilton*.

- *Conectividad Fuerte en Dígrafos:* Consiste en estudiar si hay un camino dirigido conectando cada par de vértices del dígrafo. Inclusive se puede estudiar si existe un camino dirigido entre cada par de vértices, en ambas direcciones.

- *Clausura Transitiva:* Consiste en tratar de encontrar un conjunto de vértices que pueda ser alcanzado siguiendo aristas dirigidas desde cada vértice del dígrafo.

- *Árbol de Expansión Mínima:* Consiste en encontrar, en un grafo pesado, el conjunto de aristas de peso mínimo que conecta a todos los vértices.

- *Caminos cortos a partir de un mismo origen:* Consiste en encontrar cuales son los caminos más cortos conectando a un vértice *v* cualquier con cada uno de los otros vértices de un dígrafo pesado. Este es un problema que por lo general se presenta en redes de computadores, representadas como grafos.

- *Planaridad:* Consiste en estudiar si un grafo puede ser dibujado sin que ninguna de las líneas que representan las aristas se intercepten.

- *Pareamiento (Matching):* Dado un grafo, consiste en encontrar cual es el subconjunto más largo de sus aristas con las propiedad de que no haya dos conectados al mismo vértice. Se sabe que este problema clásico es resoluble en tiempo proporcional a una función polinomial en el número de vértices y de aristas, pero aun no existe un algoritmo rápido que se ajuste a grandes grafos.

- *Ciclos Pares en Dígrafos:* Consiste en encontrar en un dígrafo un camino de longitud par. Este problema puede lucir simple ya que la solución para grafos no dirigidos es sencilla. Sin embargo, aun no se conoce si existe un algoritmo eficiente para resolverlo.

- *Asignación:* Este problema se conoce también como pareamiento bipartito pesado (*bipartite weigthed matching*). Consiste en encontrar un pareamiento perfecto de peso mínimo en un grafo bipartito. Un *grafo bipartito* es aquel cuyos vértices se pueden separar en dos conjuntos, de tal manera que todas las aristas conecten a un vértice en un conjunto con otro vértice en el otro conjunto.

- *Conectividad General:* Consiste en encontrar el número mínimo de aristas que al ser removidas separarán el grafo en dos partes disjuntas (conectividad de aristas). También se puede encontrar el número mínimo de nodos que al ser removidos separarán el grafo en dos partes disjuntas (conectividad de nodos).

- *El camino más largo:* Consiste en encontrar cual es el camino más largo que conecte a dos nodos dados en el grafo. Aunque parece sencillo, este problema es una versión del problema del tour de Hamilton y es `NP-hard`.

- *Colorabilidad:* Consiste en estudiar si existe alguna manera de asignar *k* colores a cada uno de los vértices de un grafo, de tal forma de que ninguna arista conecte dos vértices del mismo color. Este problema clásico es fácil para *k*=2 pero es `NP-hard` para *k*=3.

- *Conjunto Independiente:* Consiste en encontrar el tamaño del mayor subconjunto de nodos de un grafo con la propiedad de que no haya ningún par conectado por una arista. Este problema es `NP-hard`.

- *Clique:* Consiste en encontrar el tamaño del clique (subgrafo completo) más grande en un grafo dado.

- *Isomorfismo de grafos:* Consiste en estudiar la posibilidad de hacer dos grafos idénticos con solo renombrar sus nodos. Se conocen algoritmos eficientes para solucionar este problema, para varios clases particulares de grafos, pero no se tiene solución para el problema general. Este problema es `NP-hard`.

A continuación estudiaremos algunos algoritmos clásicos de procesamiento de grafos que resuelven algunos de los problemas anteriores.

4. RECORRIDO DE GRAFOS

Cualquier algoritmo de recorrido de grafos consiste básicamente en visitar un nodo del grafo y luego ir visitando los nodos conectados a este. Este principio se aplica recursivamente comenzando desde un nodo inicial cualquiera del grafo.

Lo que diferencia un algoritmo de recorrido de otro es, una vez ubicado en un nodo en particular, la forma en que se visitan los nodos conectados a este. Por supuesto, estos algoritmos pueden ser aplicados en grafos dirigidos o no dirigidos.

Los dos algoritmos "clásicos" de recorrido de grafos son el recorrido en profundidad y en anchura. Precisamente por ser "clásicos" han sido estudiados con anterioridad y se les conoce su orden de complejidad en tiempo y todos los beneficios de aplicarlos.

- Recorrido en profundidad

Para efectuar un recorrido en profundidad de un grafo, se selecciona cualquier nodo como punto de partida (por lo general el primer nodo del grafo) y se marcan todos los nodos del grafo como "no visitados". El nodo inicial se marca como "visitado" y si hay un nodo adyacente a este que no haya sido "visitado", se toma este nodo como nuevo punto de partida del recorrido. El recorrido culmina cuando todos los nodos hayan sido visitados.

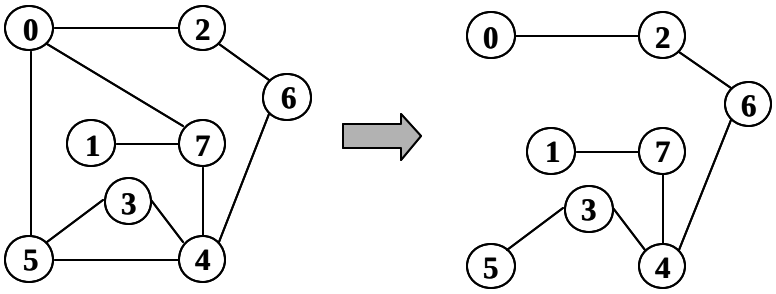
Se dice que el recorrido es en profundidad, porque para visitar otro nodo adyacente del nodo inicial, primero se deben visitar TODOS los nodos adyacentes al que se eligió antes. Es así, como el número de ambientes recursivos varía dependiendo de la profundidad que alcance el algoritmo.

Agregando el campo `logico visitado;` en la última implementación de grafos tratada, podríamos implementar el recorrido en profundidad de la siguiente manera:

```
Privado:

ACCION DFS_R(↑Nodo Actual)

      #se marca el nodo actual como visitado
      Actual↑.visitado ← verdad;

      ↑Ady aux;
      aux ←Actual.↑ListaDeAdyacentes;

      #se recorre la lista de adyacentes al nodo Actual para
      #visitarlos

      Mientras ( aux ≠ NULL ) hacer

            si ( (aux↑.AdyAeste)↑.visitado ≠ verdad) entonces

                  #se hace la llamada recursiva a los nodos
                  #adyacentes para visitarlos

                  DFS_R(aux↑. AdyAeste);
            fsi
            aux ← aux↑.proximo;
      fmientras

FACCION

Publico:

ACCION DFS()            #Depth-First Search

      ↑Nodo aux;
      aux ← primerNodo;

      #aquí se recorre la lista de nodos iterativamente
      #haciendo las llamadas  al DFS_R (DFS recursivo) cada vez que
      #se consiga a alguien no visitado. Esto se hace para evitar
      #que el algoritmo no funcione si el grafo tiene varias com-
      #ponentes conexas

      mientras (aux ≠ NULL) hacer
            si (aux↑.visitado ≠ verdad) entonces
                  DFS_R(aux);
            fsi
            aux ← aux↑.proximo;
      fmientras

FACCION
```

Este algoritmo recorre todos los nodos del grafo pero fácilmente puede modificarse para que sea una función que encuentre un nodo en particular dentro de grafo. Este algoritmo se conoce como el algoritmo DFS (Depth-First Search).

Una bondad de este algoritmo es que los nodos solo se vistan una vez. Esto implica que si se salvan en alguna estructura las aristas que se van recorriendo se obtiene un conjunto de aristas de cubrimiento mínimo del grafo, lo cual se utiliza frecuentemente se utiliza para reducir la complejidad del grafo cuando la perdida de información de algunas aristas no es importante. Este resultado se conoce como árbol DFS (*DFS Tree*).



El algoritmo de recorrido en profundidad tiene orden O(máx(A,N)) donde N es el numero de nodos y A es el numero de aristas. Esto es porque un grafo de N nodos puede tener más de N aristas, en cuyo caso se ejecutan más de N ciclos en `DFS_R`, pero si por el contrario hay menos aristas que nodos, de cualquier manera se visitaran todos los nodos en `DFS`.

El DFS puede modificarse fácilmente y utilizarse para resolver problemas sencillos como los de conectividad simple, detección de ciclos y camino simple. Por ejemplo, el número de veces que se invoca a la acción `DFS_R` desde la acción `DFS` en el algoritmo anterior es exactamente el número de componentes conexas del grafo, lo cual representa la solución al problema de conectividad simple.

- Recorrido en anchura

En este algoritmo también se utiliza la estrategia de marcas los nodos como "visitados" para detectar la culminación del recorrido, pero los nodos se recorren de una manera ligeramente distinta.

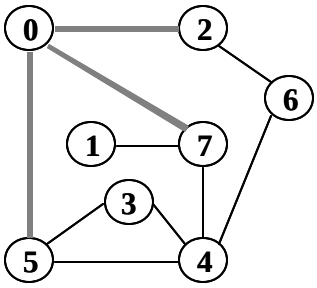
De nuevo, se selecciona cualquier nodo como punto de partida (por lo general el primer nodo del grafo) y se marcan todos los nodos del grafo como "no visitados". El nodo inicial se marca como "visitado" y luego se visitan TODOS los nodos adyacentes a este, al finalizar este proceso se busca visitar nodos más lejanos visitando los nodos adyacentes a los nodos adyacentes del nodo inicial.

Este algoritmo puede crear menos ambientes recursivos que el anterior porque visita mas nodos en un mismo ambiente, pero esto depende de cómo este construido el grafo. El algoritmo se conoce como el algoritmo de BFS (Breadth-First Search).

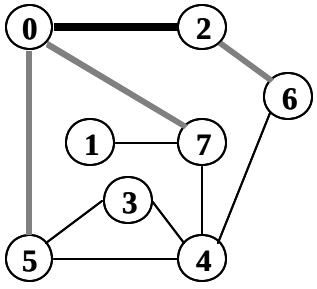
Este algoritmo tiene exactamente el mismo orden en tiempo de ejecución del algoritmo de recorrido en profundidad y también se puede obtener el conjunto de aristas de cubrimiento mínimo del grafo.

Una diferencia notable entre el DFS y el BFS es que este ultimo necesita de una estructura auxiliar, que por lo general es una cola, para el almacenamiento de las aristas que se van a visitar durante el recorrido.

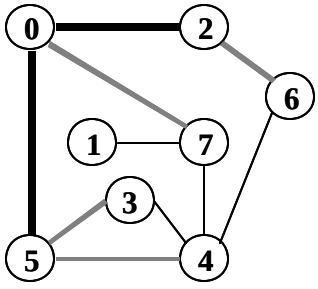
El siguiente ejemplo ilustra el funcionamiento del algoritmo BFS sobre un grafo de ejemplo. La secuencia de ilustraciones va de izquierda a derecha y de arriba hacia abajo.



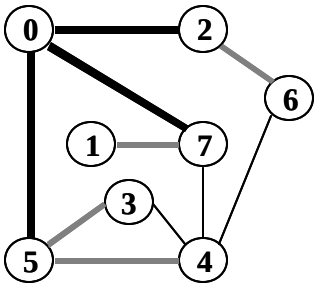
C: 0-2 0-5 0-7



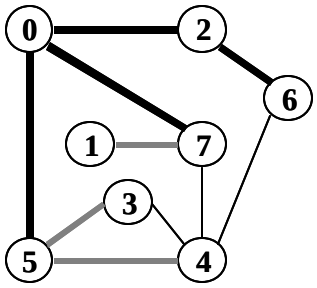
C: 0-5 0-7 2-6



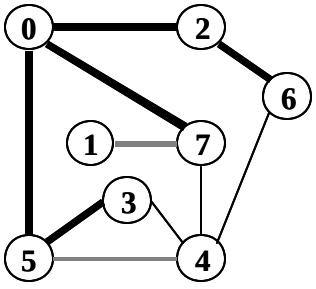
C: 0-7 2-6 5-3 5-4



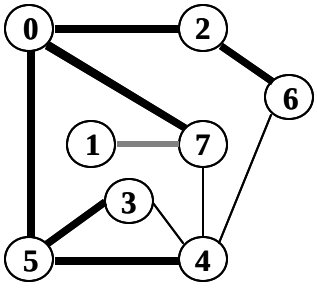
C: 2-6 5-3 5-4 7-1 7-4



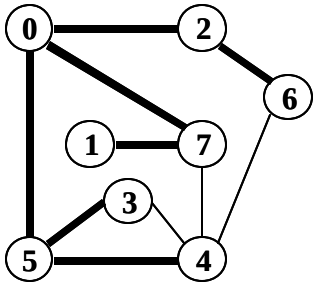
C: 5-3 5-4 7-1 7-4 6-4



C: 5-4 7-1 7-4 6-4 3-4



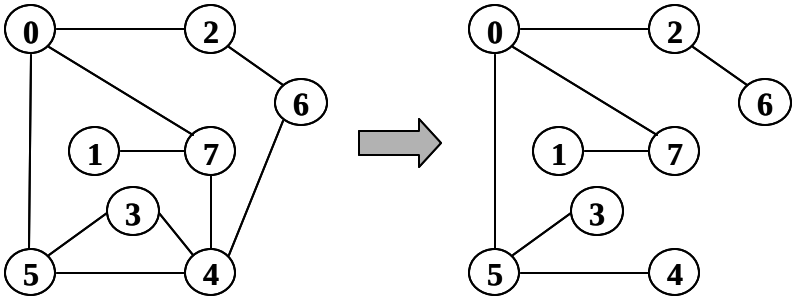
C: 7-1 7-4 6-4 3-4



Comenzamos introduciendo a la cola C todos las aristas adyacentes al nodo inicial (el nodo 0). Luego, extraemos la arista 0-2 de la cola y procesamos las aristas adyacentes a 2, la 0-2 y la 2-6. No colocamos la arista 0-2 en la cola porque el vértice 0 ya fue visitado. Luego, extraemos la arista 0-5 de la cola y procesamos las aristas adyacentes a 5. De manera similar a la anterior, no se toma en cuenta la arista 0-5, pero si se encolan las aristas 5-3 y 5-4. Seguidamente, extraemos la arista 0-7 y encolamos la arista 7-1. La arista 7-4 esta impresa en color gris porque si bien es un enlace adyacente a 7, podríamos evitar encolarla debido a que ya existe una arista en la cola que nos lleva hasta el nodo 4.

Para completar el recorrido, tomamos las aristas que quedan en la cola, ignorando aquellas que están impresas en color gris cuando queden de primeras en la cola. Las aristas entran y salen de la cola en el orden de su distancia del vértice 0.

Al igual que en el DFS, usando el BFS se puede obtener un conjunto de aristas de cubrimiento mínimo del grafo, conocido como el árbol (*BFS Tree*).



También puede modificarse fácilmente y utilizarse para resolver problemas sencillos como los de conectividad simple, detección de ciclos y camino simple.

El BFS es el algoritmo clásico para encontrar el camino más corto entre dos nodos específicos en un grafo, mientras que DFS nos ofrece muy poca ayuda para esta tarea debido a que el orden en el que se visitan los nodos no tiene absolutamente ninguna relación con la longitud de los caminos.

5. CALCULO DEL CAMINO MAS CORTO

Todo camino en un dígrafo pesado tiene un *peso* asociado, el cual es la suma de los pesos de las aristas del camino. Esta medida esencial nos permite formular problemas como el de encontrar el camino con el menor peso entre dos vértices. El tópico de esta sección es el cálculo de este tipo de camino, donde la longitud del camino no se mide en base al número de aristas del mismo, sino en base al peso del camino.

Con esto último en mente, definiremos al *camino más corto* entre dos nodos de un dígrafo pesado, como el camino dirigido que tenga la propiedad de tener el peso mínimo entre todos los caminos que existan entre dicho par de nodos.

Se pueden plantear tres tipos de problemas:

- *Camino más corto origen-destino:* Dados dos nodos *v* y *w* de un grafo, encontrar el camino más corto que comience en *v* y culmine *w*.

- *Camino más corto a partir de un origen:* Dado un nodo *v* de un grafo, encontrar el camino más corto desde *v* hasta cada uno de los demás nodos.

- *Camino más corto entre cada par de nodos*

El primero de estos problemas no es más que un caso particular del segundo. En lo que respecta al segundo, por lo general lo que se busca es tratar de obtener un árbol de caminos más cortos o el SPT (*shortest-path tree*), que consiste en un árbol dirigido que tiene como raíz al nodo origen y todo camino en el árbol en un camino corto en el grafo. El tercero de los problemas se puede resolver obteniendo los SPT correspondientes a cada uno de los nodos del grafo.

- Algoritmo de Dijkstra

El algoritmo de Dijkstra resuelve el problema de encontrar los caminos más cortos a partir de un origen, en grafos pesados que no tengan pesos negativos.

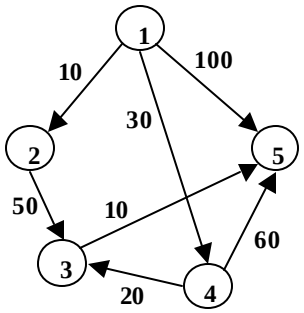
El algoritmo de Dijkstra es un algoritmo voraz que opera a partir de un conjunto S de nodos cuya distancia más corta desde el origen ya es conocida. En principio, S contiene sólo el nodo origen. En cada paso, se agrega algún nodo *v* a S, cuya distancia desde el origen es la más corta posible. Bajo la hipótesis de que los pesos son no negativos, siempre es posible encontrar un camino más corto entre el origen y *v* que pasa sólo a través de los nodos de S, al que llamaremos "especial". En cada paso del algoritmo, se utiliza un arreglo D para registrar la longitud del camino "especial" más corto a cada nodo. Una vez que S incluye todos los nodos, todos los caminos son "especiales", así que D contendrá la distancia más corta del origen a cada vértice. Se puede utilizar un arreglo P, para ir almacenando los caminos más cortos.

A continuación se muestra un esbozo del algoritmo de Dijkstra en lenguaje pseudoformal :

```
#Supóngase la existencia del siguiente grafo

Conjunto V;                        # Conj. de nodos del grafo-N nodos
Arreglo C de real [1..N][1..N]; # Matriz de costos de las aristas

#y las siguientes variables auxiliares

entero i,w,v;
Arreglo D de real [1..N];     # arreglo auxiliar de costos
Arreglo P de entero [1..N];   # arreglo que guarda los caminos más
                              # cortos
Conjunto S;                   # Conjunto de nodos evaluados
                              # Inicializado con el conjunto vacío

#el esbozo del algoritmo de Dijkstra

S.Agregar(1);                 # tomando el nodo 1 como nodo origen
Para i ←2 hasta N hacer
      D[i] ← C[1][i];         # asignando valores iniciales a D
Fpara
Para i ← 1 hasta N-1 hacer
      w ← nodo en V-S tal que D[w] sea un mínimo
      S.Agregar(w);
      Para cada nodo v ∈ V-S hacer
```

```
                D[v] ← min (D[v], D[w]+C[w,v]);
        Fpara
        P[v] ← w;
Fpara
```

Al final de la ejecución, cada posición del arreglo P contiene el nodo inmediato anterior a *v* en el camino más corto a partir del nodo inicial. Dado el grafo:



al final de la ejecución el arreglo P debe tener los valores P[2]=1, P[3]=4 , P[4]=1 y P[5]=3. Para encontrar el camino más corto del nodo 1 al nodo 5, por ejemplo, se siguen los predecesores en orden inverso comenzando en 5. Así, es sencillo encontrar que el camino más corto del nodo 1 al 5, es el 1, 4, 3, 5.

Usando el arreglo P, o bien mediante alguna otra heurística, se puede construir fácilmente un SPT, tomando el nodo inicial del recorrido como raíz del árbol y luego añadiendo una arista y un nodo a la vez, siempre tomando la próxima arista que pertenezca al camino más corto a un nodo que no este en el SPT.

El orden en tiempo de ejecución del algoritmo de Dijkstra depende muchísimo de las características del grafo y de cómo este almacenado. Si se emplea una matriz de adyacencia como en el código anterior, entonces el ciclo más interno es de O(N) y como se ejecuta N-1 veces, tendríamos un algoritmo de $O(N^2)$. Si por el contrario se utiliza una representación dinámica, justificada en el hecho de que el número de aristas es mucho menor a N^2, se podría emplear un árbol parcialmente ordenado para organizar los nodos V-S, y por lo tanto se reduciría la complejidad del ciclo más interno un orden logarítmico.

Aunque se puede ejecutar Dijkstra N veces para resolver el problemas de caminos cortos entre cada par de nodos, el *Algoritmo de Floyd* **[5]** también resuelve este problema, con una complejidad en tiempo similar.

6. ORDENAMIENTO TOPOLOGICO

El objetivo del ordenamiento topológico (*topological sorting*) es el de ser capaz de procesar los nodos de un dígrafo acíclico (DAG - *directed acyclic graph*) de tal forma de que cada nodo sea procesado antes que todos los nodos a los que apunta.

Hay dos maneras naturales de definir esta operación básica, aunque son esencialmente la misma, a saber:

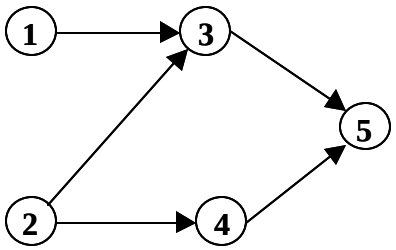
- Reetiquetado: Dado un DAG, reetiquetar sus nodos de tal forma que cada arista dirigida vaya de un nodo con un identificador bajo en numero a un nodo con un identificador cuyo identificador sea mayor.

- Reposicionado: Dado un DAG, reposicionar sus nodos en una línea horizontal de tal forma de que todas sus aristas dirigidas apunte de izquierda a derecha.

Por lo general, se usa el término de ordenamiento topológico para referirse a la versión de reposicionado.

El ordenamiento topológico puede efectuarse con facilidad agregándole una instrucción al procedimiento `DFS_R` de la sección de algoritmos de recorrido de grafos. Solo se debe agregar la instrucción `ESCRIBIR(Actual↑.Id);` en la línea antes del `FACCION`. Cuando el `DFS_R` termine de buscar en todos los nodos adyacentes a un nodo dado *x*, imprime *x*. El efecto de llamar a `DFS(v)` es imprimir en orden topológico inverso todos los vértices de un DAG accesible desde *v* por medio de un camino en el DAG. Esta técnica funciona porque no existen arcos de retroceso en un DAG.

Los DAG pueden utilizarse para modelar la estructura del pensum de una carrera universitaria. Suponiendo que se tiene el siguiente grafo con la estructura de prerrequisitos de 5 cursos universitarios:



Ejecutando un DFS modificado que comience en el nodo 1, obtenemos la secuencia 5, 3, 1, 4, 2, lo cual implica que 2, 4, 1, 3, 5 en un ordenamiento topológico de este GDA. Al tomar los cursos en esta secuencia se puede satisfacer la estructura de prerrequisitos dada.

Otra forma de encontrar un ordenamiento topológico es examinando todos los nodos del GDA y aquellos nodos que no tengan aristas incidentes sobre ellos son eliminados del grafo e introducidos en un cola. Este proceso se repite hasta que no queden nodos en el GDA. Los identificadores en la cola resultante representan un ordenamiento topológico del GDA. Tomando como ejemplo el grafo de la figura anterior, es sencillo observar que en una primera iteración se eliminarían los nodos 1 y 2. Luego los nodos 3 y 4 y finalmente el nodo 5. Esto resulta en la siguiente cola de identificadores:1, 2, 3, 4, 5, que es un ordenamiento topológico del dicho GDA.

7. CAMINOS DE EULER

La dificultad de problemas relacionados con caminos de Euler depende en gran medida de que es lo que se quiere estudiar. Esto es, que la complejidad de un algoritmo que estudie la existencia de un camino de Euler en un grafo no es la misma que la de uno que de hecho encuentre cuál es el camino.

Euler demostró que es sencillo determinar si existe un camino de Euler en el grafo, ya que todo lo que hay que hacer es verificar el grado de los nodos. Existen un teorema de la teoría de grafos que establece que un grafo tiene un camino de Euler si y solo sí es este es conexo y exactamente dos de sus nodos tienen grado impar. De manera similar, un grafo tiene un tour de Euler si y solo sí es conexo y todos sus nodos tienen grado par.

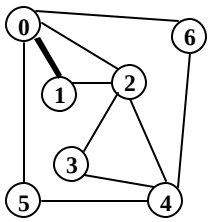
Basado en estos teoremas se puede verificar la existencia de un camino de Euler antes de intentar encontrar el camino, lo cual podría ahorrar una gran cantidad de procesamiento en la búsqueda de un camino que nunca logrará encontrarse. Además, la comprobación de grado de los nodos de un grafo es una tarea que fácilmente podría ser de complejidad lineal en el número de nodos del mismo.

Ahora bien, si lo que se desea es encontrar el conjunto de enlaces que conforman el camino, se podría utilizar un solución recursiva directa o un *backtracking*, pero esto no es recomendable. En el caso del tour tendríamos que verificar todos los caminos posibles del grafo y esto es una tarea de complejidad factorial en el número de aristas del grafo. En el caso de un camino con un origen y un destino fijos, el número de posibilidades se reduce, pero aun así tomaría mucho tiempo estudiar todas los posibles caminos con esta restricción.

Una técnica más adecuada es la de encontrar ciclos en el grafo, borrando de las listas de adyacencia las aristas que se encuentren y almacenando en un pila los nodos que se vayan encontrando, de tal manera de registrar los nodos del camino e imprimir sus enlaces, así como poder verificar caminos alternativos en cada nodo. Esta técnica puede encontrar un tour de Euler, si existe en el grafo, en tiempo lineal.

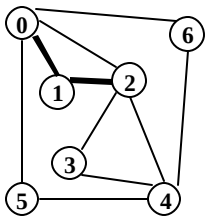
El siguiente ejemplo ilustra el funcionamiento de esta estrategia sobre un grafo de ejemplo. La secuencia de ilustraciones va de izquierda a derecha y de arriba hacia abajo. En cada una de las figuras podemos ver el grafo, las lista de adyacencia de cada nodo y la pila P que lleva registro del tour. El tour comienza en el nodo 0 y la primera arista a visitar es la 0-1. Nótese que en la primera figura P dicha arista no aparece en las listas y que en P ya se encuentran los nodos 0 y 1.

0: 2 5 6
1: 2
2: 0 1 3 4
3: 2 4
4: 2 3 5 6
5: 0 4
6: 0 4



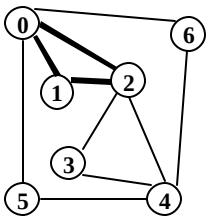
P: 0 1

0: 2 5 6
1:
2: 0 3 4
3: 2 4
4: 2 3 5 6
5: 0 4
6: 0 4



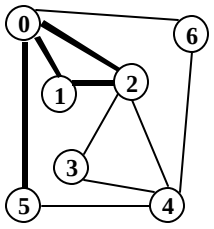
P: 0 1 2

0: 5 6
1:
2: 3 4
3: 2 4
4: 2 3 5 6
5: 0 4
6: 0 4



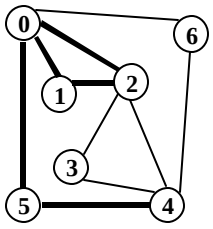
P: 0 1 2 0

0: 6
1:
2: 3 4
3: 2 4
4: 2 3 5 6
5: 4
6: 0 4



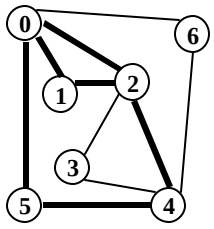
P: 0 1 2 0 5

0: 6
1:
2: 3 4
3: 2 4
4: 2 3 6
5:
6: 0 4



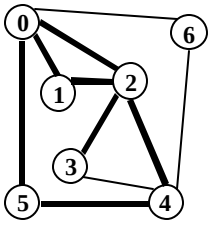
P: 0 1 2 0 5 4

0: 6
1:
2: 3
3: 2 4
4: 3 6
5:
6: 0 4



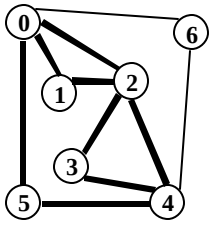
P: 0 1 2 0 5 4 2

0: 6
1:
2:
3: 4
4: 3 6
5:
6: 0 4



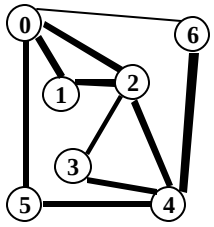
P: 0 1 2 0 5 4 2 3

0: 6
1:
2:
3:
4: 6
5:
6: 0 4



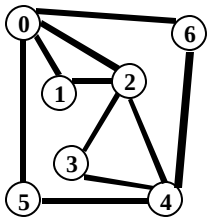
P: 1 2 0 5 4 2 3 4

0: 6
1:
2:
3:
4:
5:
6: 0



P: 0 1 2 0 5 4 2 3 4 6

0:
1:
2:
3:
4:
5:
6:



P: 0 1 2 0 5 4 2 3 4 6 0

Una vez que las listas de adyacencias estén vacías el algoritmo ha culminado su ejecución. La secuencia de nodos que se desapilan de P nos indican un tour de Euler posible para el grafo anterior: 0-6-4-3-2-4-5-0-2-1-0.

Ahora bien, supóngase que en la sexta imagen no se hubiese viajado al nodo 2 sino al nodo 6, entonces en la séptima figura se hubiese viajado al nodo 0 y desde este último no podemos elegir más enlaces, lo mismo ocurre con el nodo 6. En estos casos el algoritmo desapila de P los nodos aislados y continua con el nodo 4, en cuyo caso el algoritmo culminaría cuando P contenga los identificadores: 0 1 2 0 5 4 3 2 4. Es evidente que desde 4 se puede alcanzar a 6 y a 0 porque estaban apilados luego de él en la pila, así que basta con apilarlos de nuevo en el orden en que apilados originalmente: 6 0. En esta variante del problema se obtendría un tour de Euler diferente al anterior: 0-6-4-2-3-4-5-0-2-1-0.

8. ARBOL DE EXPANSION MINIMA

Anteriormente, tratamos el problema de encontrar la ruta más corta entre dos nodos en un digrafo pesado, pero si el grafo no es dirigido y lo que se desea es encontrar la manera menos costosa de conectar todos los puntos, entonces nos enfrentamos al problema de encontrar un *árbol de expansión mínima* (MST – *Minimum Spanning Tree*).

Un árbol de expansión de un grafo conexo es un subgrafo que contiene todos los nodos del grafo y no tiene ciclos. El árbol de expansión mínima de un grafo pesado no dirigido es el árbol de expansión cuyo peso (la suma de los pesos de todas sus aristas) no es mayor al de ningún otro árbol de expansión.

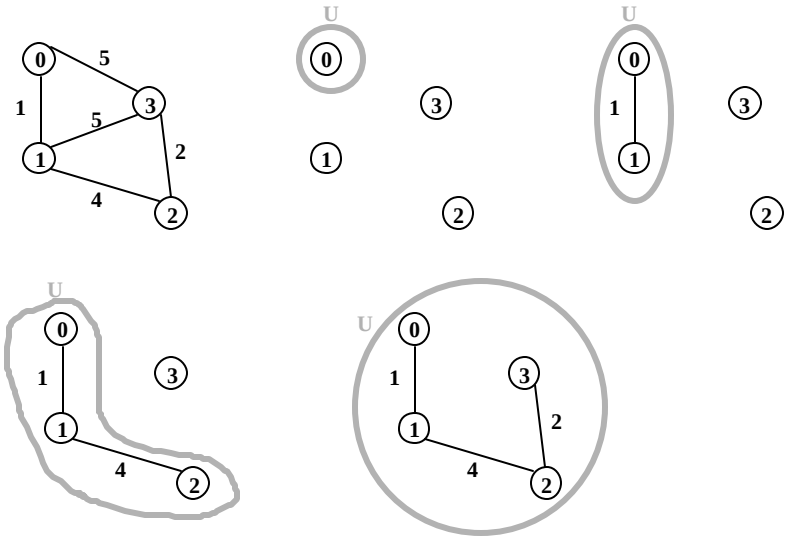
El problema de encontrar el MST de un grafo pesado no dirigido arbitrario tiene una gran cantidad de aplicaciones importantes y se conocen algoritmos para encontrarlo al menos desde 1920, pero la eficiencia de las implementaciones varía ampliamente y los investigadores aún buscan mejores métodos. En esta sección se examinan dos algoritmos clásicos para resolver este problema.

- Algoritmo de Prim

El algoritmo de Prim es tal vez el algoritmo de MST más sencillo de implementar y el mejor método para grafos densos. Este algoritmo puede encontrar el MST de cualquier grafo conexo pesado.

Sea *V* el conjunto de nodos de un grafo pesado no dirigido. El algoritmo de Prim comienza cuando se asigna a un conjunto *U* de nodos un nodo inicial perteneciente a *V*, en el cual "crece" un árbol de expansión, arista por arista. En cada paso se localiza la arista más corta *(u,v)* que conecta a *U* con *V-U*, y después se agrega *v*, el vértice en *V-U*, a *U*. Este paso se repite hasta que *V=U*. El algoritmo de Prim es de $O(N^2)$, donde | V | = N.

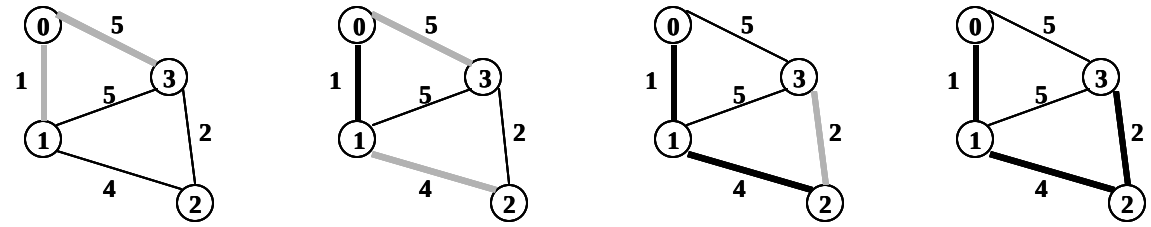
El siguiente ejemplo ilustra el funcionamiento del algoritmo. La secuencia de ilustraciones va de izquierda a derecha y de arriba hacia abajo. La primera imagen muestra el grafo pesado y las siguientes muestran el funcionamiento del algoritmo de Prim y como va cambiando el conjunto *U* durante la ejecución.



Agregar un nodo al MST es un cambio incremental, para implementar el algoritmo de Prim debemos enfocarnos en la naturaleza de ese cambio incremental. La clave está en notar que nuestro interés esta en la distancia más corta de cada vértice de *U* a *V-U*. Al

agregar un nodo *v* al árbol, el único cambio posible para cada vértice *w* fuera del árbol es que agregar *v* coloca a *w* más cerca del árbol. Esto es, no es necesario verificar la distancia de *w* a todos los demás nodos del árbol, solo se necesita verificar si la adición de *v* al árbol necesita actualizar dicho mínimo. Esto se puede lograr agregando una estructura de datos simple para evitar repetir cálculos excesivos, y hacer que el algoritmo sea más rápido y más simple.

El siguiente ejemplo ilustra la metodología anterior, utilizando una pila de aristas P, en donde las aristas se van apilando en menor a mayor según el peso de la misma. La secuencia de ilustraciones va de izquierda a derecha.



Se comienza en el nodo 0 y se apilan las aristas adyacentes a este nodo, en orden decreciente. La arista mínima es la que esta en el tope de la pila, así que se desapila y se agrega al MST. Seguidamente se procede con el nodo 1 y se apilan sus aristas adyacentes, con la diferencia de que hay que verificar si dichas aristas representan un nuevo camino mínimo con respecto a las aristas que ya están introducidas en la pila. En este caso no se apila la arista 1-3 porque ya hay una arista que lleve a 3 con el mismo costo en la pila. Se toma 1-2 porque esta en el tope y se procede con el nodo 2. Cuando se va a apilar la arista 2-3, se encuentra que ya hay un camino que lleve a 3 pero de mayor costo, así que se desapila 0-3 y luego se apila 2-3. Se agrega 2-3 al MST y termina el proceso, porque el conjunto de nodos en el MST es igual al conjunto de vértices del grafo original.

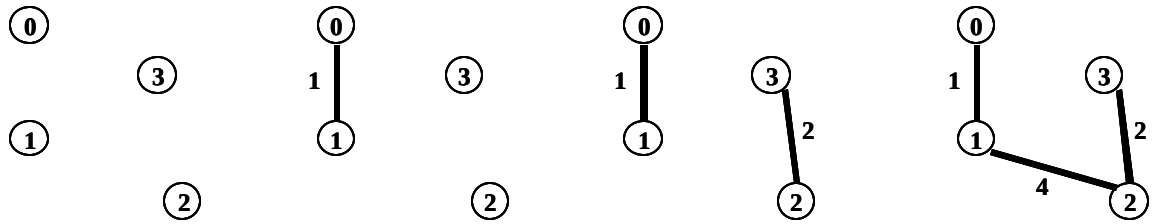
- Algoritmo de Kruskal

El algoritmo de Prim construye un MST una arista a la vez, encontrando una nueva arista que agregar a un MST que va creciendo en cada paso. El algoritmo de Kruskal también construye el MST una arista a la vez, con la diferencia que este encuentra una arista que conecte dos MST que van creciendo dentro de un bosque de MST crecientes, formado de los nodos del grafo original.

El algoritmo comienza a partir de un conjunto de árboles degenerados formados por un solo nodo, que son los nodos del grafo, y se comienzan a combinar los árboles de dos en dos usando la arista menos costosa posible, hasta que solo quede un solo árbol: El MST.

Dada una lista de las aristas del grafo, el primer paso del algoritmo de Kruskal es ordenarlas por peso (usando un *quicksort* por ejemplo). Luego se van procesando las aristas en el orden de su peso, agregando aristas que no produzcan ciclos en el MST. El algoritmo de Kruskal es de $O(A*\log_2 A)$, donde A es el número de aristas del grafo.

El siguiente ejemplo ilustra el funcionamiento del algoritmo. La secuencia de ilustraciones va de izquierda a derecha y de arriba hacia abajo.

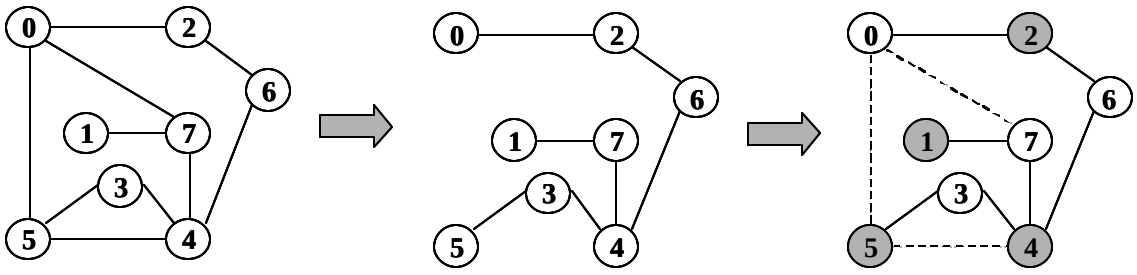


9. GRAFOS BIPARTITOS, CICLOS IMPARES Y COLORACIÓN

El problema de verificar si un grafo es bipartito es equivalente al problema de colorabilidad de grafos para dos colores, simplemente son el mismo problema pero con diferente nomenclatura. Con respecto al problema de detección de la existencia de ciclos impares, cualquier grafo con un ciclo de longitud impar es claramente no colorable para dos colores. En vista de que los tres problemas son equivalentes es posible elaborar un solo algoritmo que resuelva los tres problemas.

Para colorear un grafo con dos colores, dado un color asignado a un vértice *v*, debe colorearse el resto del grafo asignando el otro color a cada vértice adyacente a *v*. Este proceso es equivalente a alternar los dos colores para los nodos en cada nivel de un árbol DFS, y durante el retorno de las llamadas recursivas verificar si no hay arista que conecte dos nodos del mismo color, ya que tal arista es evidencia de un ciclo de longitud impar.

A continuación ilustraremos esta idea, usando el árbol DFS de la sección de recorrido de grafos. La primera figura muestra el grafo original, la segunda el árbol DFS y la tercera es el grafo coloreado a blanco y gris, usando la heurística descrita anteriormente.



Una vez construido el árbol DFS y coloreado los nodos, es sencillo comprobar que las aristas 5-4 y 0-7 conectan dos nodos del mismo color, lo cual es evidencia de la existencia de un ciclo de longitud impar. De hecho, el ciclo 0-5-3-4-7-0 es un ciclo de longitud impar. Si eliminamos 5-4 y 0-7 el grafo sería colorable y no habría ciclos de longitud impar.

Como vimos anteriormente el recorrido DFS tiene complejidad lineal, por lo tanto estos tres problemas también. Esto ilustra las maneras en las que DFS pueden darnos una

idea de la estructura del grafo al mismo tiempo que demuestra que es posible resolver varios problemas de procesamiento de grafos en tiempo lineal.

10. COMPONENTES FUERTEMENTE CONEXAS

Una componente fuertemente conexa de un dígrafo es un conjunto máximo de nodos en el cual existe un camino que va desde cualquier nodo del conjunto hasta cualquier otro nodo del conjunto. Si el dígrafo conforma una sola componente fuertemente conexa, se dice que el dígrafo es fuertemente conexo.

Es sencillo desarrollar una solución de fuerza bruta para este problema. Solo se debe comprobar para cada par de nodos *u* y *v*, si *u* es alcanzable desde *v* y viceversa. Sin embargo, se puede utilizar el DFS para determinar con eficiencia las componentes fuertemente conexas de una dígrafo.

- Método de Kosaraju

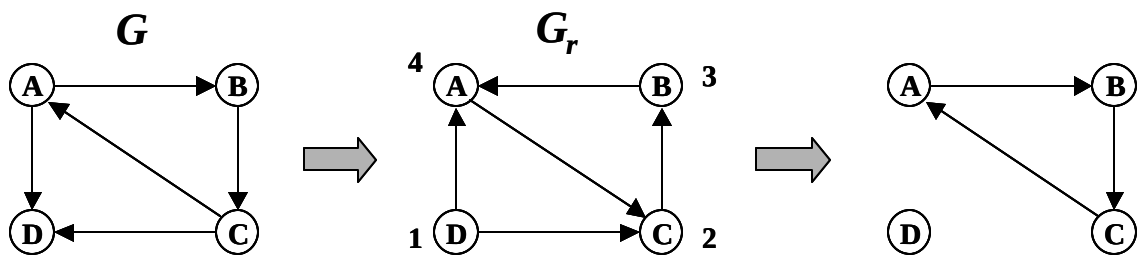
El método de Kosaraju es simple de explicar y de implementar. Sea *G* un dígrafo, a continuación se presenta una descripción del algoritmo de Kosaraju, a saber:

- Se ejecuta un DFS y se construye un dígrafo nuevo G_r, invirtiendo las direcciones de todos los arcos de *G*. Esto es, se permutan los nodos del grafo en el orden definido por el recorrido en postorden del mismo (para un dígrafo acíclico este proceso es equivalente a un ordenamiento topológico). Se deben enumerar los nodos en el orden de terminación de las llamadas recursivas del DFS.

- Luego se ejecuta un DFS en G_r, partiendo del nodo con numeración más alta de acuerdo con la numeración asignada en el paso anterior. Si el DFS no llega a todos los nodos, se inicia la siguiente búsqueda a partir del nodo restante con numeración más alta.

- Cada árbol del conjunto de árboles DFS resultante es una componente fuertemente conexa de *G*.

A continuación ilustraremos este método. La primera figura muestra el grafo *G*, la segunda muestra el grafo G_r obtenido luego del primer paso del método y la última figura muestra las dos componente fuertemente conexas del grafo original *G*.



Se puede observar que en G_r las aristas estas invertidas. Los números que pueden verse a los lados de los nodos son los asignados durante la terminación de las llamadas del DFS, donde el nodo A fue el nodo inicial. La modificación del algoritmo original de DFS para lograr esta numeración es sumamente sencilla.

Las componentes resultantes son el producto de un DFS en G_r comenzando con el nodo A y luego otro DFS comenzando con el nodo D, siguiendo la heurística del segundo paso del método descrito.

El método de Kosaraju encuentra las componentes fuertemente conexas de cualquier dígrafo en tiempo y espacio lineal, mostrando una tremenda superioridad con respecto al algoritmo de fuerza bruta descrito al comienzo de esta sección.

11. TOUR HAMILTONIANO Y TSP

Encontrar un tour hamiltoniano para un grafo cualquiera es un problema intratable, ya que la única solución conocida para el caso general del problema es la de buscar la solución entre todas las posibilidades de ciclos en el grafo. Como la gran mayoría de los problemas intratables, el problema de encontrar un tour hamiltoniano es un problema `NP-hard`.

Otros problemas que son variaciones del tour hamiltoniano heredan el mismo nivel de dificultad. Por ejemplo, uno de los problemas más famosos de procesamiento de grafos que esta íntimamente relacionado con el tour hamiltoniano es el Problema del Agente Viajero (TSP – *Traveling Salesperson Problem*).

El problema del TSP consiste en encontrar el tour de costo mínimo que pase por todos los nodos del grafo, donde los nodos por lo general representan ciudades y los pesos de las aristas costos de viaje de una ciudad a otra. Es evidente su relación con el problema del tour hamiltoniano, y si no es posible encontrar una solución eficiente para éste, mucho menos para el TSP.

Se ha intentado atacar el problema de TSP utilizando otras técnicas. En **[1]** se plantea un posible solución utilizando algoritmos de búsqueda local, pero solo funciona para acelerar el algoritmo en casos particulares del problema. También se pueden utilizar algoritmos genéticos para obtener una aproximación al costo mínimo del TSP.

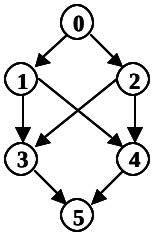
12. FLUJO EN REDES

Los grafos y dígrafos son solo abstracciones matemáticas, pero son útiles en la práctica porque nos ayudan a resolver numerosos problemas importantes. En esta sección pensaremos en los dígrafos pesados como *redes* y trataremos de resolver problemas referentes a situaciones dinámicas donde un cierto material *fluye* a través de la red, con diferentes costos asociados a diferentes rutas. Estos son llamados *problemas de flujo en redes*.

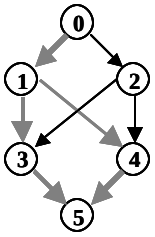
Un *red de flujos* es una red de aristas con pesos positivos, denominados *capacidades*. Un *flujo* en una red de flujos es un conjunto de pesos de aristas – que se denominan *flujos de aristas* – que satisfacen las condiciones de que ningún flujo de arista es mayor que su capacidad y que el flujo total que entra a cada nodo interno al flujo, es igual al flujo total que sale de dicho nodo. El valor del flujo es la suma de los flujos de las aristas que lo conforman.

La propiedad característica de un flujo es la condición de equilibrio local de que para cada nodo interno, el flujo que entra a dicho nodo es igual al flujo que sale del mismo. El flujo que entra a un nodo es la suma de los pesos de todas las aristas cuyo destino es dicho nodo y el flujo que sale es la suma de los pesos de todas las aristas cuyo origen es dicho nodo.

La figura siguiente muestra un red de flujos y un flujo posible de 0 a 5. Es sencillo comprobar que el flujo está en equilibrio.



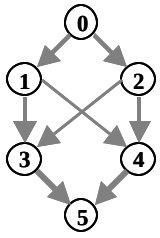
	cap.
0-1	2
0-2	3
1-3	3
1-4	1
2-3	1
2-4	1
3-5	2
4-5	3



	cap.	flujo
0-1	2	2
0-2	3	0
1-3	3	1
1-4	1	1
2-3	1	0
2-4	1	0
3-5	2	1
4-5	3	1

Uno de los problemas más comunes de flujo en redes es el de *Flujo Máximo*. Esto es, dada un red de flujos, encontrar un flujo de un nodo *v* a uno *u* tal que ningún otro flujo tenga mayor valor. El flujo que sale de *v* debe ser igual al flujo que entra a *u*. En algunas aplicaciones basta con saber cuál es el valor del flujo, pero generalmente se desea saber cuál es el flujo que lo origina (los flujos de las aristas).

La figura siguiente muestra un ejemplo de un Flujo Máximo para el grafo anterior.

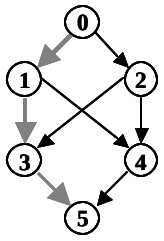


	cap.	flujo
0-1	2	2
0-2	3	2
1-3	3	1
1-4	1	1
2-3	1	1
2-4	1	1
3-5	2	2
4-5	3	2

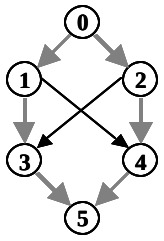
En 1962, L. R. Ford y D. R. Fulkerson **[4]** desarrollaron una técnica efectiva para resolver problemas de flujo máximo. Es un método genérico para aumentar la capacidad del los flujos incrementalmente a lo largo de los caminos que van del origen al destino, que sirve como la base para un familia de algoritmos. Esta técnica es conocida como el método Ford-Fulkerson en la literatura clásica, aunque también puede encontrarse como el *aumenting-path method*.

Considérese cualquier camino dirigido del origen al destino en la red de flujos. Sea *x* la mínima de las capacidades de las aristas no usadas en el camino. Es posible incrementar el flujo de la red al menos en *x*, incrementando el flujo de las aristas del camino en dicho monto. De esta forma se obtiene el primer intento de flujo en la red. Luego debemos encontrar otro camino, incrementar el flujo en el camino, y continuar hasta que todos los caminos del origen al destino tengan al menos una *arista llena* (el flujo usa toda la capacidad de la arista).

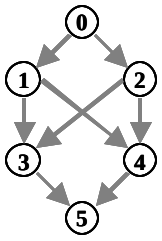
Las siguientes figuras ilustran el funcionamiento de esta estrategia sobre el grafo de ejemplo. La secuencia de ilustraciones va de izquierda a derecha



	cap.	flujo
0-1	2	2
0-2	3	0
1-3	3	2
1-4	1	0
2-3	1	0
2-4	1	0
3-5	2	2
4-5	3	0



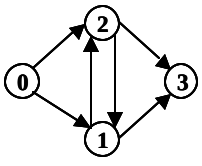
	cap.	flujo
0-1	2	2
0-2	3	1
1-3	3	2
1-4	1	0
2-3	1	0
2-4	1	1
3-5	2	2
4-5	3	1



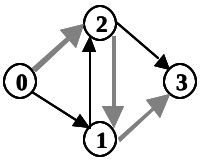
	cap.	flujo
0-1	2	2
0-2	3	2
1-3	3	1
1-4	1	1
2-3	1	1
2-4	1	1
3-5	2	2
4-5	3	2

Aunque esta estrategia calcula el flujo máximo en varios casos, también falla en muchos otros. Para mejorar el algoritmo de tal manera de que siempre encuentre el flujo máximo, se debe considerar una manera más general de incrementar el flujo de origen a destino a través del grafo no dirigido subyacente. Las aristas de cualquier camino del origen al destino *avanzan* o *retroceden*. Las aristas que avanzan van con el flujo y las que retroceden van en sentido contrario al flujo. Ahora bien, para cada camino que no tenga aristas llenas que avancen ni *aristas vacías* (flujo cero) que retrocedan, podemos incrementar la cantidad de flujo en la red incrementando el flujo en las aristas que avanzan y decrementándolo en las aristas que retroceden. La cantidad en la que el flujo puede ser incrementado esta limitado por la mínima capacidad que no haya sido usada en las aristas que avanzan y los flujos de las aristas que retroceden.

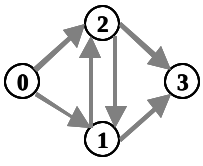
Las siguientes figuras ilustran el funcionamiento de esta ultima estrategia en un nuevo grafo de ejemplo. La secuencia de ilustraciones va de izquierda a derecha y de arriba hacia abajo.



	cap.	flujo
0-1	3	0
0-2	1	0
1-3	1	0
1-2	1	0
2-1	1	0
2-3	3	0

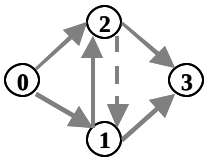


	cap.	flujo
0-1	3	0
0-2	1	1
1-3	1	1
1-2	1	0
2-1	1	1
2-3	3	0



	cap.	flujo
0-1	3	1
0-2	1	1
1-3	1	1
1-2	1	1
2-1	1	1
2-3	3	1

Hasta este punto no se han tomado en cuenta las aristas en retroceso y se tiene un flujo cuyo valor es 6. Ahora tratemos el camino 0-1-2-3, pero usando la arista 2-1 que va en dirección contraria al flujo.



	cap.	flujo
0-1	3	2
0-2	1	1
1-3	1	1
1-2	1	1
2-1	1	0
2-3	3	2

Obsérvese que se le resto valor al flujo que retrocede para agregárselo a los que avanzan. Aquí se obtiene un flujo de 7, que es el flujo máximo de esta red. En este momento ya no hay más caminos que no tengan aristas llenas que avancen o aristas vacías que retrocedan y el algoritmo culmina su ejecución.

Este proceso describe la base para el algoritmo clásico de Ford-Fulkerson (*aumenting-path method*) para problemas de flujo máximo en redes. En resumen:

- Se comienza con flujo nulo en todas partes. Luego se incrementa el flujo a lo largo de cualquier camino de origen a destino que no tenga aristas llenas que avancen o aristas vacías que retrocedan. Se continúa hasta que no hayan más caminos como estos en la red.

Este método siempre encuentra el flujo máximo, sin importar como se escojan los caminos.

Cuando las capacidades son valores enteros, el flujo se incrementa en al menos una unidad en cada iteración, así que la terminación es finita. Mas aún, para un grafo con V nodos y A aristas con capacidades enteras positivas, el algoritmo ejecuta un máximo de V*A iteraciones para encontrar el flujo máximo. Chvátal **[3]** describe el caso en el que las capacidades son irracionales.

13. BIBLIOGRAFIA

[1] Aho, Alfred V.; Hopcroft, Jhon E. ; Ullman, Jeffrey D. *Data Estructures and Algorithms.* Addison-Wesley. Massachusetts, EUA. 1983.

[2] Coto, Ernesto. *Lenguaje Pseudoformal para la Construcción de Algoritmos*. ND 2002-08. Laboratorio de Computación Gráfica. Universidad Central de Venezuela. Caracas, Venezuela. Octubre, 2002.

[3] Chvátal, V. *Linear Programming*. W.H. Freeman and Co., New York, NY, 1983.

[4] Ford Jr., L. R. ; Fulkerson, D. R. *Flows in Networks.* Princeton University Press, Princeton, NJ, 1962.

[5] Sedgewick, Robert. *Algorithms in C++, Third Edition, Part 5: Graph Algorithms.* Addison-Wesley. 2002.