**El grafo**

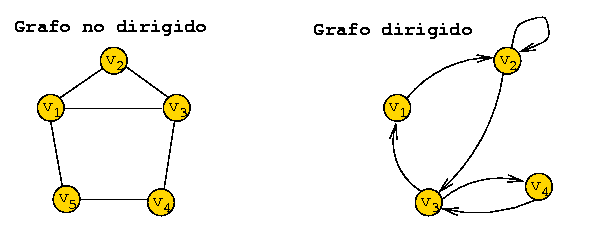
**El grafo consiste de un conjunto V de vértices (o nodos) y un conjunto E de arcos que conectan a esos vértices.**

En primera instancia debemos entender que es un grafo. Un grafo es una composición de un conjunto de objetos conocidos como nodos que se relacionan con otros nodos a través de un conjunto de conexiones conocidas como aristas.

Los grafos  permiten estudiar las relaciones que existen entre unidades que interactúan con otras.

Podemos representar diversas situaciones o elementos con grafos. Estos son extraordinariamente útiles en situaciones complejas, es por esto, que es común conseguir la implementación de análisis de grafos en estudios de ciencias exactas, ciencias sociales y en aplicaciones informáticas.

**Ejemplos:**



Además de esto, los grafos pueden ser extendidos mediante la adición de rótulos (labels) a los arcos. Estos rótulos pueden representar costos, longitudes, distancias, pesos, etc.

**DFS - Recorrido en profundidad**

Un Recorrido en profundidad (en inglés DFS o Depth First Search) es un algoritmo que permite recorrer todos los nodos de un grafo. Es una generalización del recorrido preorden de un árbol.

Este método es una forma básica de recorrer un grafo implementando recursividad. En este recorrido se basan los recorridos preorden y postorden para árboles binarios. Supóngase que una persona se encuentra en algún sistema de cuevas interconectadas, o en un laberinto, en una cierta intersección (o nodo), y que se le pide a esta persona que busque la salida, que se encuentra en un determinado nodo. En esta búsqueda se podrían emplear varias opciones.

Una posibilidad que probablemente no se utilizase sería la búsqueda en amplitud. Intuitivamente, una estrategia que comienza en algún nodo de la cueva y que después visita todos y cada uno de los nodos adyacentes siguientes de la cueva, no parece demasiado prometedora.

**La estrategia de recorrido en profundidad es la siguiente:**

1. Se toma un nodo “s” como comienzo, y se marca.

2. A continuación se toma y se marca un nodo no marcado adyacente a “s”, y ese nodo pasa a ser el nuevo nodo de partida, dejando posiblemente por el momento al nodo inicial original con nodos no explorados.

1. La búsqueda continúa por el grafo hasta que el camino en curso finalice con un grafo de salida igual a cero, o bien en un nodo en que todos los nodos adyacentes estén marcados.

4. A continuación la búsqueda vuelve al último nodo que todavía tenga nodos adyacentes sin marcar, y continúa marcando todos los nodos de forma recursiva hasta que ya no queden nodos sin marcar.

A medida que recorremos el grafo, iremos numerando correlativamente los nodos encontrados Suponemos que todos estos números son cero inicialmente, y utilizamos un contador global n, también inicializado en cero. El siguiente algoritmo en seudo-código muestra cómo se puede hacer este tipo de recorrido recursivamente:

DFS(v) // recorre en profundidad a partir del vértice v

{

++n;

DFN[v]=n;

for(todo w tal que {v,w} está en E y DFN[w]==0)

DFS(w);

}

Para hacer un recorrido en profundidad a partir del nodo v, utilizamos el siguiente programa principal:

n=0;

for(todo w)

DFN[w]=0;

DFS(v);

Si hubiera más de una componente conexa, esto no llegaría a todos los nodos. Para esto podemos hacer:

n=0;

ncc=0; // número de componentes conexas

for(todo w)

DFN[w]=0;

while(existe v en V con DFN[v]==0)

{

++ncc;

DFS(v);

}

**BFS - Recorrido en amplitud**

El recorrido de búsqueda en anchura, en amplitud o expansión, es una estrategia aplicable indistintamente al caso de grafos dirigidos y no dirigidos. El recorrido en anchura es una generalización del recorrido por niveles de un árbol. Se trata de visitar un nodo inicial y luego a todos los nodos que están a un arco de distancia de éste, luego a todos los nodos que están a dos arcos de distancia de éste y así sucesivamente, hasta alcanzar a todos los nodos a los que se pueda llegar desde el nodo inicial. Una aplicación típica de este recorrido es la resolución de problemas de planificación. La búsqueda en amplitud se puede utilizar para hallar la distancia más corta entre algún nodo inicial y los nodos restantes del grafo. Esta distancia más corta es el mínimo número de aristas que hay que recorrer para pasar desde el nodo inicial hasta el nodo concreto que se esté examinando.

Comenzando en el nodo “s”, esta distancia se calcula examinando todas las aristas incidentes en el nodo “s”, y pasando después a un nodo adyacente “w”, repitiéndose entonces todo el proceso. El recorrido continúa hasta que se hayan examinado todos los nodos del grafo. En una búsqueda en amplitud, cada nodo se visita o es procesado en algún sentido, dependiendo de la aplicación concreta. La búsqueda comienza en un nodo concreto del grafo. A continuación la búsqueda se extiende a los nodos del grafo que estén más próximos al nodo inicial antes de visitar ningún otro.

**El recorrido en anchura trabaja de la siguiente manera:**

1. Se visita el nodo inicial.

2. Después de visitar el nodo inicial se visitan todos los sucesores.

3. Después los sucesores de los sucesores.

4. Se repiten los pasos 1 y 2 hasta encontrar un nodo sin arcos salientes o ya visitados.

5. Si después de visitar todos los descendientes del primer nodo, todavía quedan más nodos por visitar, se repite el proceso reiniciando. Este tipo de recorrido es muy usado para problemas de planificación (problema de laberinto).

**Algoritmo de Dijkstra.**

También llamado algoritmo de caminos mínimos, es un algoritmo para la determinación del camino más corto dado un vértice origen al resto de vértices en un grafo con pesos en cada arista. Su nombre se refiere a Edsger Dijkstra, quien lo describió por primera vez en 1959.

**Aplicaciones**

En múltiples aplicaciones donde se aplican los grafos, es necesario conocer el camino de menor costo entre dos vértices dados:

* Distribución de productos a una red de establecimientos comerciales.
* Distribución de correos postales.
* Sea G = (V, A) un grafo dirigido ponderado.

El problema del camino más corto de un vértice a otro consiste en determinar el camino de menor costo, desde un vértice u a otro vértice v. El costo de un camino es la suma de los costos (pesos) de los arcos que lo conforman.

**Características del algoritmo**

* Es un algoritmo greddy.
* Trabaja por etapas, y toma en cada etapa la mejor solución sin considerar consecuencias futuras.
* El óptimo encontrado en una etapa puede modificarse posteriormente si surge una solución mejor.

Pasos del algoritmo

Algoritmo de Dijkstra. Inicialización.

* Sea V un conjunto de vértices de un grafo.
* Sea C una matriz de costos de las aristas del grafo, donde en C[u,v] se almacena el costo de la arista entre u y v.
* Sea S un conjunto que contendrá los vértices para los cuales ya se tiene determinado el camino mínimo.
* Sea D un arreglo unidimensional tal que D[v] es el costo del camino mínimo del vértice origen al vértice v.
* Sea P un arreglo unidimensional tal que P[v] es el vértice predecesor de v en el camino mínimo que se tiene construido.
* Sea vinicial el vértice origen. Recordar que el Algoritmo Dijkstra determina los caminos mínimos que existen partiendo de un vértice origen al resto de los vértices.

La idea del algoritmo es mantener un conjunto A de nodos "alcanzables" desde el nodo origen e ir extendiendo este conjunto en cada iteración.

Los nodos alcanzables son aquellos para los cuales ya se ha encontrado su camino óptimo desde el nodo origen. Para esos nodos su distancia óptima al origen es conocida. Inicialmente A={s}.

Para los nodos que no están en A se puede conocer el camino óptimo desde s que pasa sólo por nodos de A. Esto es, caminos en que todos los nodos intermedios son nodos de A. Llamemos a esto su camino óptimo tentativo.

En cada iteración, el algoritmo encuentra el nodo que no está en A y cuyo camino óptimo tentativo tiene largo mínimo. Este nodo se agrega a A y su camino óptimo tentativo se convierte en su camino óptimo. Luego se actualizan los caminos óptimos tentativos para los demás nodos.

El algoritmo es el siguiente:

A={s};

D[s]=0;

D[v]=cost(s,v) para todo v en V-A; // infinito si el arco no existe

while(A!=V)

{

Encontrar v en V-A tal que D[v] es mínimo;

Agregar v a A;

for(todo w tal que (v,w) está en E)

D[w]=min(D[w],D[v]+cost(v,w));

}

*Implementaciones*:

* Usando una cola de prioridad para la tabla D el tiempo es O(m log n).
* Usando un arreglo con búsqueda secuencial del mínimo el tiempo es O(n2).

*Ejemplo*:

