E.A.P. DE INGENIERIA DE SISTEMAS E INFORMATICA

Módulo de Tópicos I

I Unidad

Lic. Luis Ramirez Milla

Abril - 2014

Chimbote – Perú

I N D I C E

Página

Presentación

UNIDAD I: ARBOLES, MONTICULOS Y GRAFOS

Capítulo I: Introducción.

1.1 Análisis de algoritmos 3
1.2 Estructura de datos 6
1.3 Arboles binarios 8
1.4 Árboles de búsqueda 10

Capítulo II: Arboles 2-3-4. Arboles Rojo/Negro.

2.1 Introducción. 12
2.2 Arboles descendentes 2-3-4 13
2.3 Arboles Roji-Negros 17

Capítulo III: Colas de prioridades. Montículos.

3.1 Cola de prioridades 23
3.2 Montículos binarios 24
3.3 Montículos – d 26
3.4 Montículos binarios a la izquierda 26

Capítulo IV: Grafos I. Algoritmos fundamentales. Conectividad. Grafos ponderados.

4.1 Algoritmos sobre grafos elementales 28
4.2 Conectividad 34
4.3 Grafos ponderados 35

Capítulo V: Grafos II. Grafos dirigidos. Flujo de red.

5.1 Grafos dirigidos 39
5.2 Flujo de red 42

REFERENCIAS BIBLIOGRAFICAS. 45

PRESENTACION

El propósito del presente módulo es servir como guía a los estudiantes que cursan la asignatura Tópicos I. El material presentado es apropiado para el desarrollo de la primera unidad, describiendo en forma detallada términos que forman parte de la formación integral del futuro ingeniero de sistemas.

El presente modulo esta estructurado en cinco capítulos, cada uno de los cuales se detallan a continuación:

Capitulo I, describe el marco introductorio a las estructuras de datos avanzados en los relacionado a los arboles de búsqueda..

Capitulo II, describe el reconocimiento de los arboles multirrama, y en particular los 2-3-4 y los rojinegros como variante de implementación de los 2-3-4.

Capitulo III, describe la implementación de algoritmos que permitan la asignación de prioridades adecuadas y que sirvan como base para la construcción de algoritmos más avanzados.

Capitulo IV, describe la implementación de algoritmos fundamentales para la solución de problemas basados en la teoría de grafos en lo relacionado a conectividad y ponderación.

Capitulo V, describe la implementación de algoritmos fundamentales para la solución de problemas basados en la teoría de grafos en lo relacionado a direccionalidad y flujos de red.

Finalmente se incluyen las referencias bibliográficas.

UNIDAD I

ARBOLES, MONTICULOS Y GRAFOS

CAPITULO I

Introducción

1.1 Análisis de algoritmos

Para la mayoría de los problemas existen varios algoritmos diferentes. **¿Cómo elegir uno que conduzca a la mejor implementación?**

Normalmente los problemas a resolver tienen un “tamaño” natural (en general, la cantidad de datos a procesar), al que se denominara **N** y en función del cual se tratara de describir los recursos utilizados (con frecuencia, la cantidad de tiempo empleado).

El punto de interés es el **estudio del caso medio**, es decir, el tiempo de ejecución de un conjunto “tipo” de datos de entrada, y **el del peor caso**, el tiempo de ejecución para la configuración de datos de entrada mas desfavorable.

Pasos a considerar en el análisis de algoritmos:

El primer paso del análisis de un algoritmo es establecer las características de los datos de entrada que utilizara y decidir cuál es el tipo de análisis más apropiado.

Idealmente, sería deseable poder obtener, para cualquier distribución de probabilidad de las posibles entradas, la correspondiente distribución de los tiempos empleados en la ejecución del algoritmo. Pero no es posible alcanzar este ideal para un algoritmo que no sea trivial, de manera que, por lo regular, se limita el desarrollo estadístico intentando probar que el tiempo de ejecución es siempre menor que algún “límite superior” sea cual sea la entrada, e intentando obtener el tiempo de ejecución medio para su entrada “aleatoria”.

El segundo paso del análisis de un algoritmo es identificar las operaciones abstractas en las que se basa, con el fin de separar el análisis de la implementación.

Así, por ejemplo, se separa el **estudio del número de comparaciones** que realiza un algoritmo de ordenación del **estudio para determinar cuántos microsegundos** tarda una computadora concreta en ejecutar un código de maquina cualquiera producido por un compilador determinado para el fragmento de código if (a[i] < V)… El primero dependerá de las propiedades el algoritmo, mientras que el segundo dependerá de las propiedades de la computadora.

El tercer paso del análisis de un algoritmo es analizarlo matemáticamente, con el fin de encontrar los valores del caso medio y del peor caso para cada una de las cantidades fundamentales.

No es difícil encontrar un límite superior del tiempo de ejecución de un programa – el reto es encontrar el mejor límite superior, aquel que se encontraría si se diera la peor entrada posible –. Esto produce el peor caso: el caso medio normalmente requiere un análisis matemático más sofisticado.

Clasificación de los algoritmos

La mayoría de los algoritmos tienen un parámetro primario **N**, normalmente el número de elementos de datos a procesar, que afecta muy significativamente al tiempo de ejecución. El parámetro **N** podría ser el grado de un polinomio, el tamaño de un archivo a ordenar o en el que se va a realizar una búsqueda, el número de nodos de un grafo, etc.

Proporcionalidad del tiempo de ejecución de un algoritmo

Proporcionalidad	Características
1	La mayor parte de las instrucciones de la mayoría de los programas se ejecutan una vez o muy pocas veces (constante)
Log N	Cuando el tiempo de ejecución de un programa es logarítmico, este será ligeramente ms lento a medida que crezca N.
N	Cuando el tiempo de ejecución de un programa es lineal, eso significa generalmente que para cada elemento de entrada se realiza una pequeña cantidad de procesos.
N log N	Este tiempo de ejecución es el de los algoritmos que resuelven un problema dividiéndolo en pequeños sub-problemas, resolviéndolos independientemente, y combinando después las soluciones.
N^2	Cuando el tiempo de ejecución de un algoritmo es cuadrático, solo es práctico para problemas relativamente pequeños.
N^3	Un algoritmo que procesa tríos de elementos de datos (por ejemplo un bucle anidado triple) tiene un tiempo de ejecución cubico y no es útil más que en problemas pequeños.
2^N	Pocos algoritmos con un tiempo de ejecución exponencial son susceptibles de poder ser útiles en la práctica, aunque aparecen de forma natural al aplicar el

	método de la “fuerza bruta” en la resolución de problemas.

1.2 Estructuras de datos

Estructuras lineales

Son flexibles pero son secuenciales, un elemento detrás de otro. Vectores, listas.

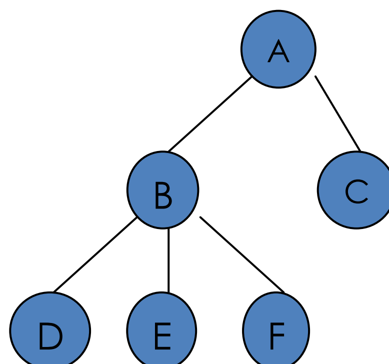
Estructuras no lineales

- Junto con los árboles, los grafos son estructuras de datos no lineales
- Superan las desventajas de las listas
- Sus elementos se pueden recorrer de distintas formas, no necesariamente uno detrás de otro.

Son muy útiles para la búsqueda y recuperación de información

ARBOLES

Estructura que organiza sus elementos formando jerarquías: PADRES e HIJOS



Los elementos de un árbol se llaman **nodos**

Si un nodo **p** tiene un enlace con un nodo **m**,

p es el padre y **m** es el hijo

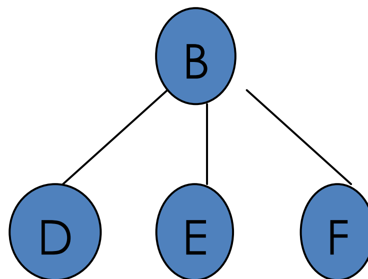
Los hijos de un mismo padre se llaman: **hermanos**

Todos los nodos tienen al menos un padre, menos la raíz: A

Si no tienen hijos se llaman **hoja**: D, E, F y C

Un subárbol de un árbol

Es cualquier nodo del árbol junto con todos sus descendientes



TERMINOLOGIA

Camino: Secuencia de nodos conectados dentro de un árbol

Longitud del camino: Es el número de nodos menos 1 en un camino

Nivel de un árbol: Es el número de nodos entre la raíz y el nodo más profundo del árbol

Altura del árbol: Es el nivel más alto del árbol

Un árbol con un solo nodo tiene altura 1

Nivel (profundidad) de un nodo: Es el número de nodos entre el nodo y la raíz.

Grado (aridad) de un nodo: Es el número de hijos del nodo

Grado (aridad) de un árbol: Máxima aridad de sus nodos

Tipo Abstracto Dato - ARBOL: Definición Informal

Valores:

Un nodo puede almacenar contenido y estar enlazado con sus árboles hijos (puede ser dos o más)

Operaciones: Dependen del tipo de árbol, pero en general tenemos:

- Vaciar o inicializar el Árbol
- Eliminar un árbol
- Saber si un árbol está vacío
- Recorrer un árbol

Tipo Abstracto Dato - ARBOL: Definición formal

<arbol> ::= <<**NULL**>> | <nodo>
<nodo> ::= <contenido>{<arbol>}
<contenido> ::= <<**dato**>>{<<**dato**>>}

1.3 Arboles Binarios

Tipo especial de árbol

Cada nodo no puede tener más de dos hijos

Un árbol puede ser un conjunto:

- **vacío**, no tiene ningún nodo
- o constar de tres partes:
  Un nodo raíz y
  Dos subárboles binarios: izquierdo y derecho

La definición de un árbol binario es recursiva

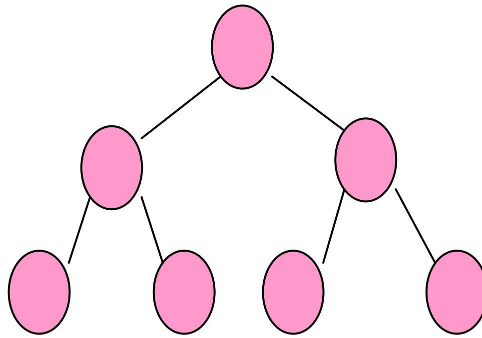
La definición global depende de si misma

Arboles binarios llenos

Un árbol de altura h, está lleno si:

Todas sus hojas están en el nivel h

Los nodos de altura menor a h tienen siempre 2 hijos



Sea T un árbol

Si T está vacío,

Entonces T es un árbol binario lleno de altura 0

Si T no está vacío, y tiene h>0

Esta lleno si los subárboles de la raíz, son ambos árboles binarios llenos de altura h-1

Arboles binarios completos

Un árbol de altura h está completo si:

Todos los nodos hasta el nivel h-2 tienen dos hijos cada uno y

En el nivel h-1, si un nodo tiene un hijo derecho, todas las hojas de su sub-árbol izquierdo están a nivel h

Si un árbol está lleno, también está completo

Un árbol equilibrado es cuando

La diferencia de altura entre los subárboles de cualquier nodo es máximo 1

Un árbol binario equilibrado totalmente

Los subárboles izquierdo y derecho de cada nodo tienen la misma altura: es un árbol lleno

Recorridos

Recorrer es:

- Visitar todos los elementos de una estructura

Como recorrer un árbol

- Hay tantos caminos, ¿cuál escoger?
    - Existe tres recorridos típicos, nombrados de acuerdo a la posición de la raíz
        - Preorden: **raíz** - subarbol izq. - subarbol der.
        - Enorden : subarbol izq. - **raíz** - subarbol der.
        - Postorden : subarbol izq. - subarbol der. - **raíz**

1.4 Árbol de búsqueda

Árbol binario de búsqueda (abb)

- Los elementos en un árbol
    - Hasta ahora no han guardado un orden
    - No sirven para buscar elementos

- Los arboles de búsqueda
- Permiten ejecutar en ellos búsqueda binaria
- **Dado un nodo:**
    - Todos los nodos del sub. Izq. Tienen una clave menor que la clave de la raíz
    - Todos los nodos del sub. Der. Tienen una clave mayor que la clave de la raíz

Creación de un abb

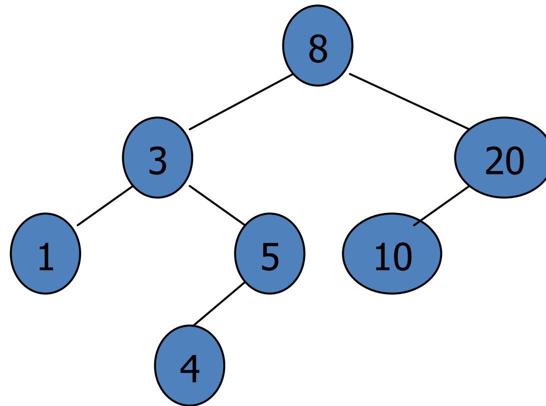
Un árbol de búsqueda debe mantener

- A la derecha valores mayor a la raíz
- A la izquierda valores menor a la raíz

Ejemplo:

- Construya árbol con los siguientes elementos:

- 8, 3, 1, 20, 10, 5, 4



CAPITULO II

Arboles 2-3-4. Arboles Rojo/Negro

2.1 Introducción

Arboles equilibrados AVL

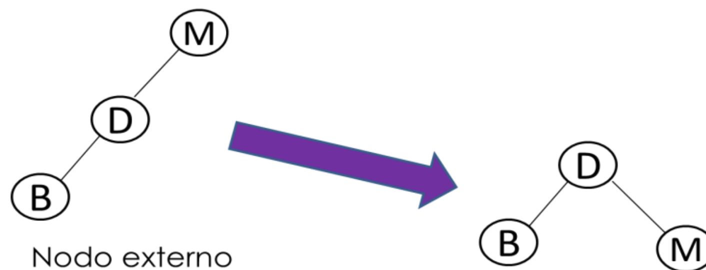
Los algoritmos de árboles binarios fundamentales (búsqueda secuencial, binaria, indirecta, etc.) son muy útiles en un gran número de aplicaciones, pero tienen un problema de dar un mal rendimiento en el peor caso. En la búsqueda en arboles binarios es posible hacerlo mucho mejor, pues hay una técnica que permite garantizar que el peor caso no ocurrirá. A esta técnica se le llama **equilibrar**.

Propiedades:

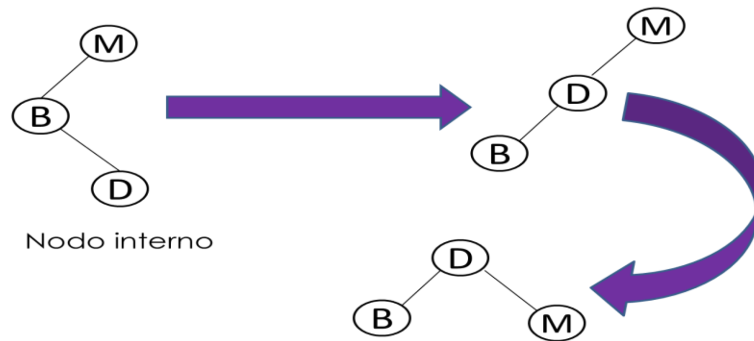
- La diferencia entre las alturas de los hijos, nunca es mayor a 1
- Sus dos hijos son AVL

Operaciones:

- Rotaciones simples



- Rotaciones dobles



Los árboles AVL tienen un buen manejo del balanceo. Sin embargo, en las inserciones es necesario un recorrido descendente para establecer el lugar de la inserción y otro recorrido ascendente para actualizar las alturas de los nodos y, posiblemente, ajustar su equilibrio. Para entender con más facilidad la filosofía de funcionamiento de estos árboles, recurriremos a los árboles 2–3–4. No se suelen utilizar en la vida real pero son una buena herramienta didáctica para entender los árboles rojinegros.

2.2 Arboles descendentes 2-3-4

Definición

Son árboles de 4 ramas de búsqueda, son equilibrados mediante la técnica de partición y recombinación para mantener el equilibrio.

Las reglas de los árboles 2-3-4.

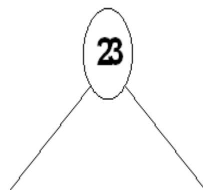
1. Cada nodo puede tener 1, 2 o 3 valores ordenados. Por lo tanto pueden tener 2, 3 o 4 hijos que deben cumplir las siguientes reglas:

a) Si el nodo solo tiene una clave se comporta como un árbol binario de búsqueda.
b) Si el nodo tiene dos valores de claves, debe tener tres hijos, el subárbol izquierdo tiene valores menores que la primer clave, el árbol central tiene valores entre la clave uno y la dos, el árbol derecho tiene valores mayores que la segunda clave.
c) Si el nodo tiene tres valores de clave, denominados c1,c2 y c3. El subárbol izquierdo tiene valores menores que c1, el subárbol central izquierdo tiene los valores mayores que c1 y menores que c2, el subárbol central derecho tiene los valores mayores que c2 y menores que c3, el subárbol derecho tiene valores mayores que c3.

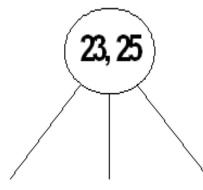
2. Todos los caminos desde la raíz a los nodos externos tienen la misma altura.

3. Los nodos internos no tienen enlaces nulos.

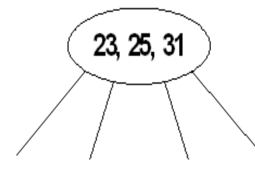
Ejemplos de los tres tipos de nodos que puede tener un árbol 2–3–4



Tipo 2

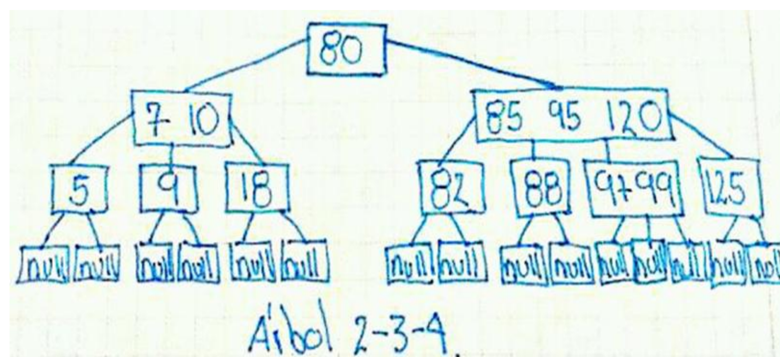


Tipo 3



Tipo 4

Ejemplo de Árbol 2-3-4

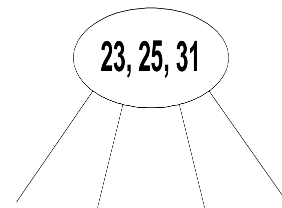


La inserción

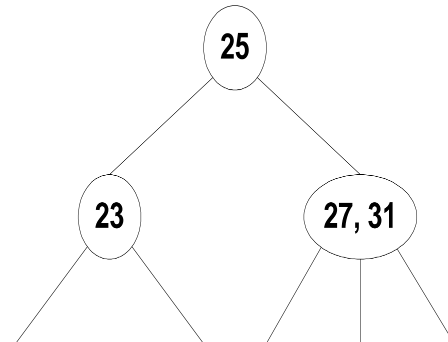
Siempre se inserta un elemento como hoja. La altura no pude ser diferente en ninguna hoja, esto obliga a realizar operaciones de balanceo. Una vez que se llega a identificar la ubicación del nuevo nodo, si es un 2-nodo o 3-nodo se inserta directamente. En el caso de 4-nodo se realiza una partición dividiendo dicho nodo en dos nodos 2-nodo y ubica el valor clave central en el nodo padre.

Como se indicó anteriormente, el inconveniente se presenta cuando el algoritmo indica que la nueva clave se debe insertar en un nodo **Tipo 4*.*** La solución podría ser la siguiente:

Si $k_i = 27$ y se tiene el siguiente nodo **Tipo 4:**

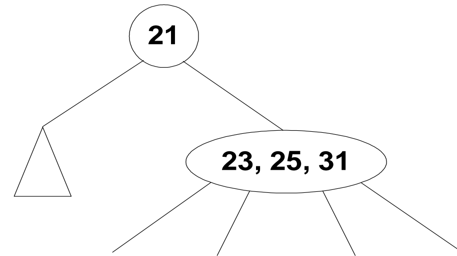


Entonces, el nuevo sub–árbol sería:

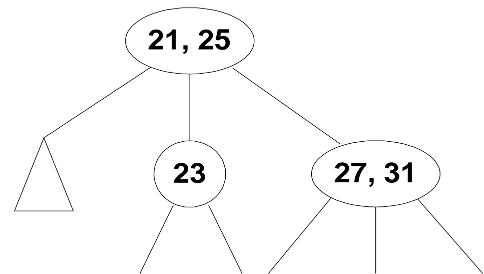


Sin embargo, es necesario considerar qué sucede cuando el nodo Tipo–4 anterior es hijo de un nodo Tipo–2 o Tipo–3. Con la solución propuesta anteriormente, se estaría desperdiciando la posibilidad de incluir la clave 25 dentro del padre y así evitar problemas de balanceo y una creación innecesaria de un nodo. Esta podría ser una solución:

Si $k_i = 27$ y se tiene el siguiente el siguiente



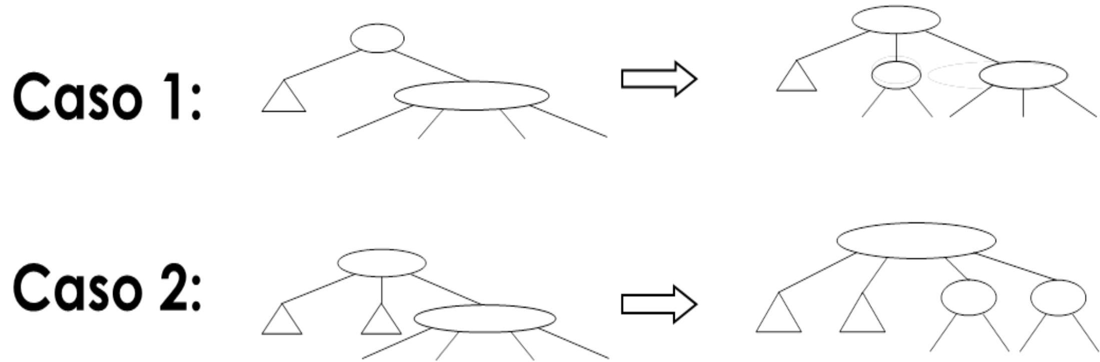
Entonces, el nuevo sub–árbol sería:



Ahora sólo queda evaluar qué hacer cuando el padre de un nodo Tipo–4 también es un Tipo–4. Esto es necesario ya que el nodo padre no tiene capacidad de alojar más claves. Una solución podría ser la de dividir ambos pero esta decisión lleva a la ineficiencia de que, en el peor de los casos, sea necesario subir hasta la raíz.

Un procedimiento posible es el de asegurar que el camino de búsqueda nunca terminará en un nodo Tipo–4 y dividir todos los nodos Tipo–4 que se vayan encontrando en el recorrido. Esto se ve claramente en las soluciones propuestas anteriormente para la inserción de la clave 27: nunca se optó por descender por el puntero delimitado por las claves 25 y 31 y crear un nodo Tipo–2 que contenga la clave 27.

Las divisiones de los nodos Tipo–4 en otros tipos de nodo nunca generan problemas de balanceo. Esto se ve claro en los siguientes posibles casos:



No es preocupante que en el Caso 2 se "traslade" el nodo Tipo–4 a un nivel superior. En el siguiente recorrido descendente que se tenga que realizar, se dividirá dicho nodo. En cualquier caso, la resultante de los dos casos muestra que nunca se insertará una clave en un nodo Tipo–4.

El único caso que generará un aumento de nivel se presenta cuando la raíz es un nodo Tipo–4. Se la dividirá, entonces, en un padre Tipo–2 con dos hijos del mismo tipo. Sin embargo, este aumento de nivel tampoco genera problemas de balanceo.

Esta política de inserción permite que el árbol siempre se mantenga muy bien balanceado. En los árboles AVL, por ejemplo, se tiene que recurrir a un número de balanceo por cada nodo y, cuando se genera una diferencia de pesos, de deben realizar rotaciones. "En un árbol 2–3–4, nunca se realizan rotaciones".

2.3 Arboles Roji-Negros

Definición

Un árbol rojinegro es un árbol balanceado, de búsqueda, cuyas hojas pueden ser "rojas" o "negras" y que sigue las siguientes PROPIEDADES:

1. Cada nodo está coloreado con los colores rojo o negro.
2. La raíz siempre es negra.
3. Si un nodo es rojo, sus hijos deben ser negros.

4. Cualquier camino desde la raíz a una hoja o a un hijo nulo, debe tener la misma cantidad de nodos negros.
5. Cada apuntador de Hoja es de color negro

¿Cómo se cumplen las propiedades?

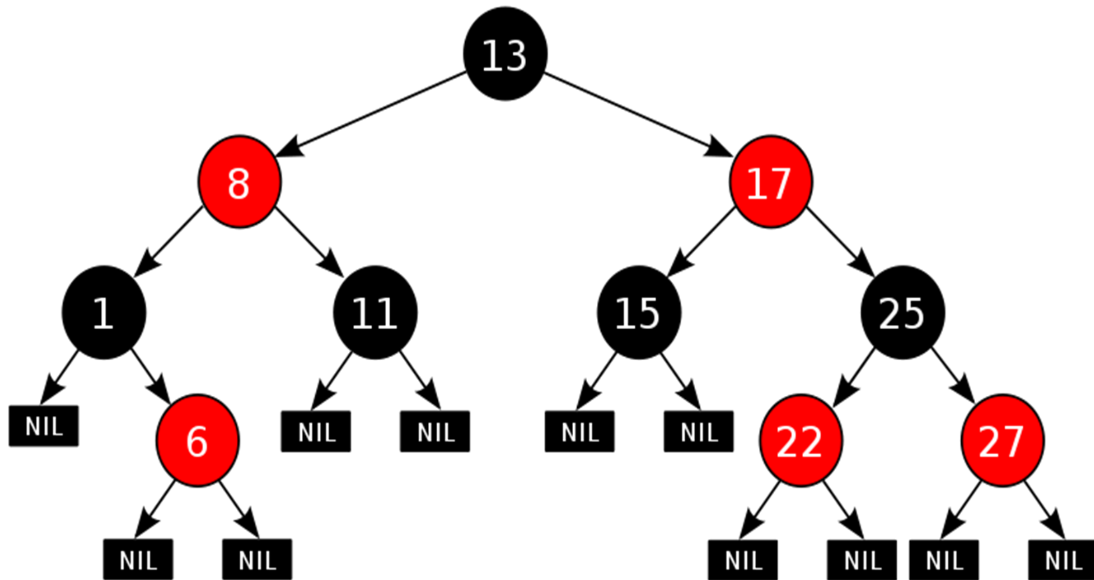
- Se puede cambiar el color de los nodos
- Se pueden realizar rotaciones
- No se permiten llaves duplicadas

Tomando como base de trabajo los árboles 2–3–4, el nodo “**rojo**” está indicando que su clave “comparte el nodo” con la clave de su padre en un árbol 2–3–4.
Siguiendo esta indicación, un subárbol que tenga un padre **negro** y sus dos hijos **rojos** está representando un nodo **Tipo–4** en un árbol 2–3–4. En esto, se entiende el porqué de la propiedad n° 3.

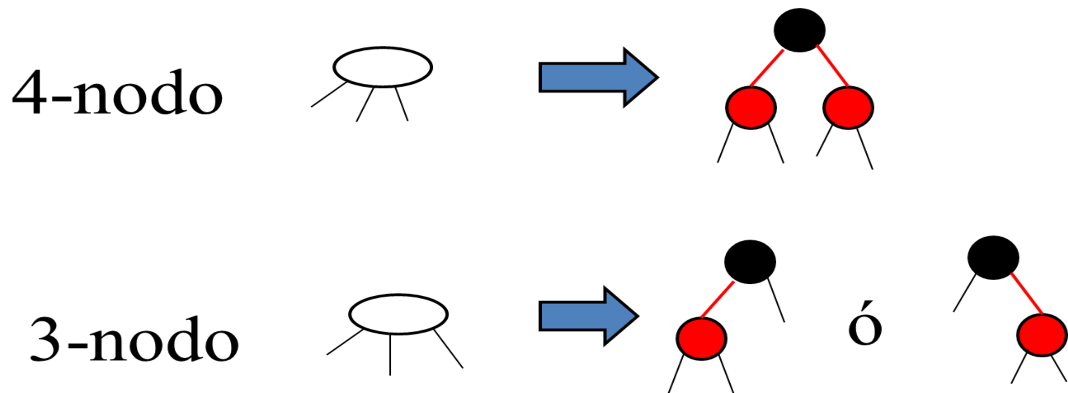
Características

- Fueron inventados por Rudolf Bayer (1972) el cuál los llamo árboles simétricos binarios.
- El Nodo contiene un campo extra (bit) el cuál se utiliza para almacenar el color del enlace que apunta a dicho nodo. Este campo será 1 si el enlace que apunta al nodo es rojo y 0 si es negro.
- El árbol puede recorrerse por cualquier color, ya sea rojo o negro.
- Los recorridos más largos varían a lo más el doble del más corto, esto quiere decir que el árbol está prácticamente balanceado.

Ejemplo de árbol Roji-Negro



Representación rojinegra



Inserción

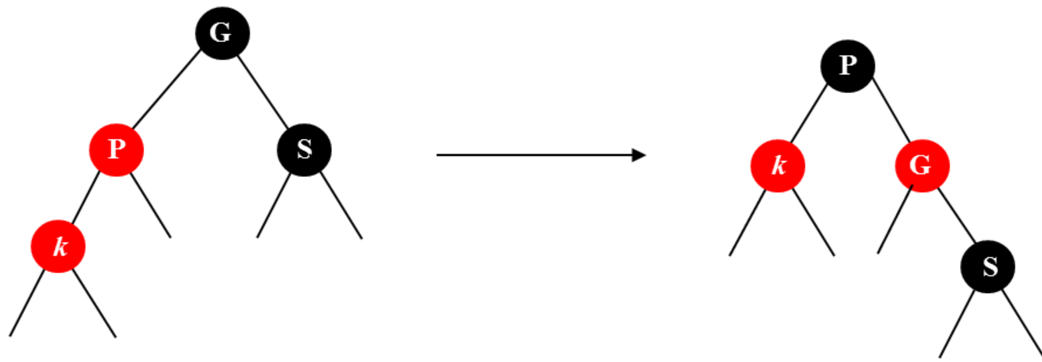
Un árbol Rojo-Negro es un árbol binario, por lo tanto una inserción en este se hará de la misma forma que en un binario, pero el nodo a insertar será siempre rojo. Posteriormente se reajustan las propiedades del mismo.

Inserción ascendente

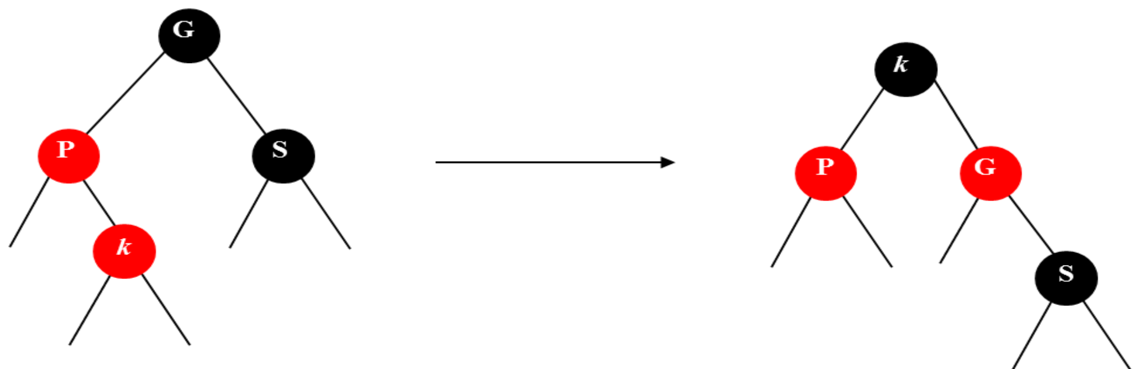
El proceso de inserción ascendente de una clave "k" comienza con la premisa de que todos los nodos a insertar deben ser rojos.

Caso 1: Si su padre es negro, el algoritmo finaliza con la inserción.

Caso 2: Si su padre es rojo, k es un nodo exterior respecto de su abuelo (G) y el hermano de su padre (S) es negro (se supone que las referencias null son negras), entonces se debe hacer una rotación simple y un cambio de color según se indica en el gráfico:



Caso 3: Si su padre es rojo, k es un nodo interior respecto de su abuelo (G) y el hermano de su padre (S) es negro (se supone que las referencias null son negras), entonces se debe hacer una rotación doble y un cambio de color según se indica en el gráfico:



Caso 4: Sólo queda ver el caso en que el hermano del padre (S) también sea rojo. En este caso, tanto el padre (P) como el hermano del padre (S) se colorean a negro y el abuelo pasa a rojo.

Inserción descendente

Los árboles rojinegros presentan la gran ventaja de que sus rutinas de inserción y eliminación se pueden efectuar con un único recorrido descendente. Este manejo, además se puede realizar con una implementación no recursiva. Como resultado, se obtendrán algoritmos más simples y eficientes.

El proceso de inserción descendente de una clave “k” evita tener que subir por el árbol para realizar las rotaciones o los cambios de color. Básicamente se trata de que, a la hora de insertar el nodo, S no sea rojo.

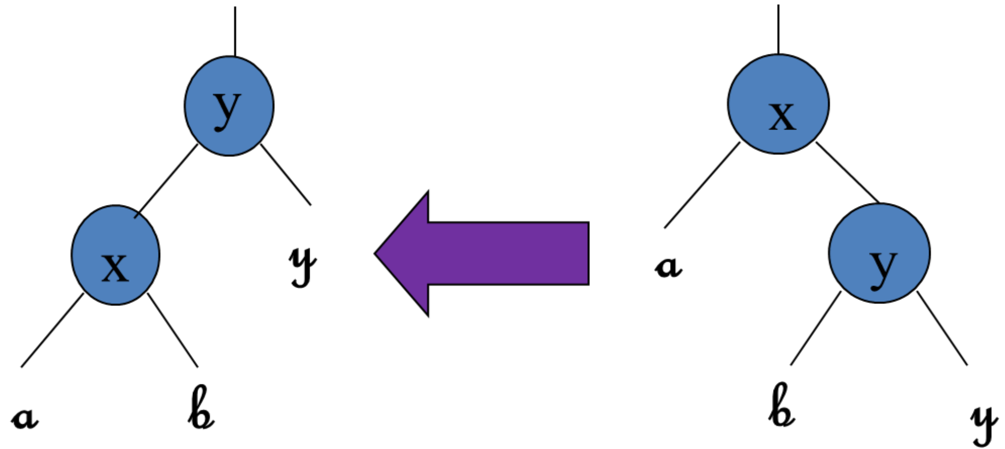
En el recorrido hacia abajo, cuando se encuentra un nodo X con dos hijos rojos, convertimos a X en rojo y a sus hijos en negro.
Si introducimos dos nodos rojos consecutivos, se pueden aplicar los Casos 2 o 3.

Rotación

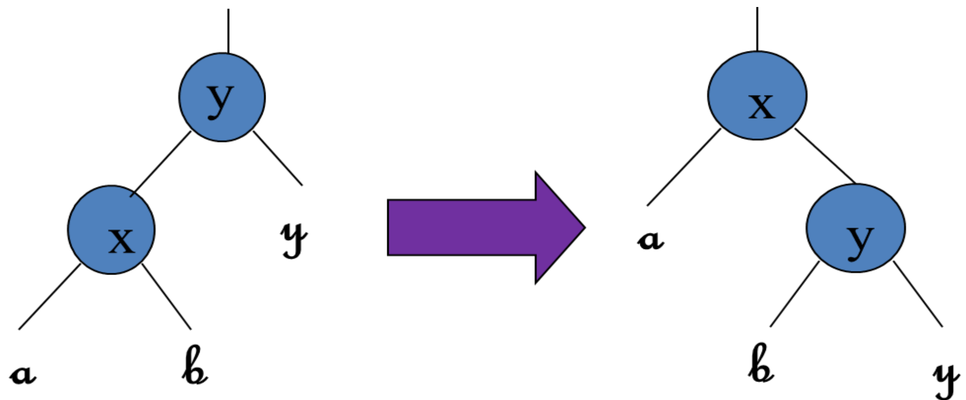
Las operaciones INSERTA y ELIMINA de un árbol Rojo-Negro modifican la estructura de este y pueden violar las propiedades de los mismos antes mencionadas, por lo cual es necesario reestablecerlas lo que implicaría cambiar el color de algunos nodos.

Tipos de rotación

- Rotación Izquierda:
  Cuando realizamos una Rotación Izquierda a un nodo X, asumimos que su hijo derecho no es nulo.



- Rotación Derecha:
  Cuando realizamos una Rotación Derecha a un nodo Y, asumimos que su hijo izquierdo no es nulo.



3.1 Cola de prioridades

Introducción

En muchas aplicaciones **los registros con clave se deben procesar en orden**, pero no necesariamente en orden completo, ni todos a la vez. A veces se forma un conjunto de registros y se procesa el mayor; a continuación posiblemente se incluyan otros elementos y luego se procesa el nuevo registro máximo y así sucesivamente.
Una estructura de datos apropiada para un entorno como este es aquella que permita insertar un nuevo elemento y eliminar el mayor. Esta estructura que se puede contrastar con las colas (donde se elimina el más antiguo) o con las pilas (donde se elimina el más reciente), se denomina **cola de prioridad.**

Las colas de prioridad son estructuras de datos que resultan ser útiles en muchas aplicaciones informáticas. Es útil pensar en los valores de las claves asociados con los elementos como prioridades. Así, las claves obedecen a una relación de orden total.

Las aplicaciones de las colas de prioridades incluyen por ejemplo:

- La gestión de un planificador de tareas en un Sistema MultiUsuario.
  - los trabajos que consumen menos recursos
  - los trabajos del administrador del sistema
- La gestión de los trabajos enviados a impresión
  - los trabajos más importantes primero
  - los trabajos más cortos primero

Por razones de utilidad se debe precisar algo más sobre la forma de tratar las colas de prioridad, puesto que existen varias operaciones que pueden ser necesarios llevar a cabo sobre ellas, para preservarlas y poderlas utilizar con eficacia en las aplicaciones.

Lo que se desea es construir y mantener una estructura de datos que contenga registros con claves numéricas (prioridades) y que cuente con algunas de las operaciones siguientes:

- Construir una cola de prioridad a partir de N elementos
- Insertar un nuevo elemento
- Suprimir el elemento más grande
- Cambiar la prioridad de un elemento
- Unir dos colas de prioridad en una más grande

Las operaciones más importantes en una cola de prioridades se refieren aquellas que permiten repetidamente seleccionar el elemento de la cola de prioridad que tiene como clave el valor mínimo (máximo). Esto conlleva a que una cola de prioridad **P** debe soportar las siguientes operaciones:

ColaPrioridad(T)
Insertar(P,x): añade el elemento **x** a la cola de prioridad **EncontrarMin(P):** Devuelve el elemento de **P** con la prioridad con menor valor. **EliminarMin(P):** Quita y devuelve el elemento con la prioridad con menor valor.

3.2 Montículos binarios

Propiedades estructurales de los montículos

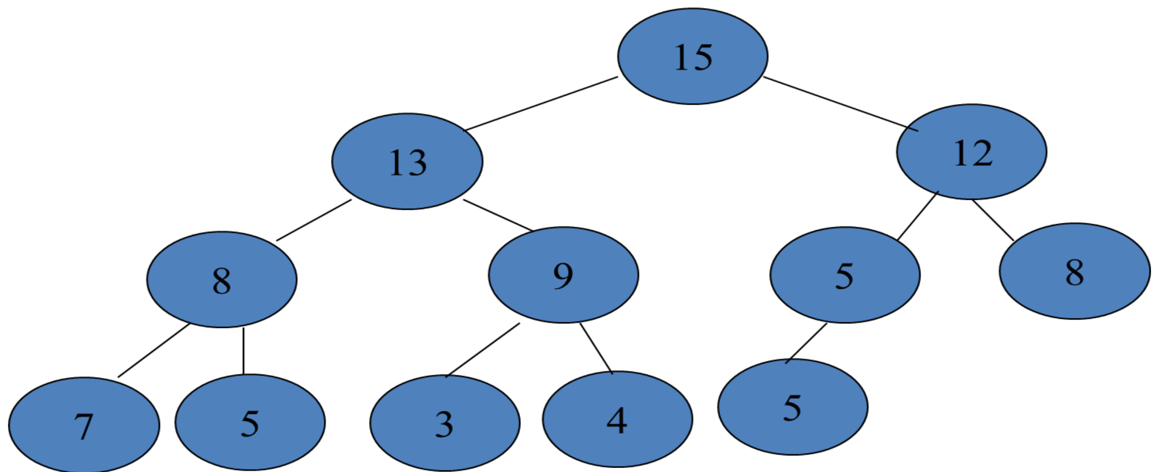
Un montículo binario (o simplemente montículo o heap) es un árbol binario completo: todos los niveles están llenos con la posible excepción del nivel más bajo, que se llena de izquierda a derecha.

Un árbol binario completo de altura **h**, tiene entre $\mathbf{2^h}$ y $\mathbf{2^{h+1}-1}$ nodos, esta regularidad facilita su representación mediante un vector. Para cualquier elemento en la posición **i** del vector, el hijo izquierdo está en la posición **2i**, el hijo derecho en **2i+1** y el padre en **i/2**

Propiedades de orden de los montículos

1. El mínimo (máximo) está en la raíz
2. Y como todo subárbol también es un montículo, todo nodo debe ser menor (mayor) o igual que todos sus descendientes

Ejemplo de montículos de máximos



15	13	12	8	9	5	8	7	5	3	4	5
1	2	3	4	5	6	7	8	9	10	11	12

Al hacer uso de un vector, se observa que:

- No hay necesidad de almacenar punteros como en los ABB.
- Los cálculos de índices tardan menos tiempo que los de referencia de punteros asociados a una representación enlazada.

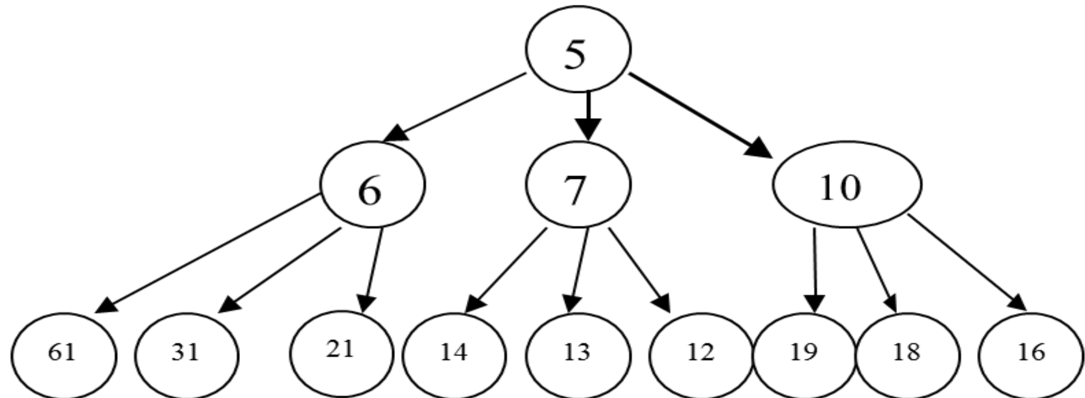
Mantenimiento de montículos

La operación EncontrarMin() o EncontrarMax(), se realiza en orden constante ya que solo será necesario acceder al valor de la raíz.

Las operaciones Insertar(x), EliminarMin() o EliminarMax() no tienen implantaciones triviales en un montículo binario. Es necesario asegurar que ambas operaciones no destruyan las propiedades del montículo.

3.3 Montículos – d

Montículos parecidos a los binarios, excepto que todos los nodos tienen **d** hijos.



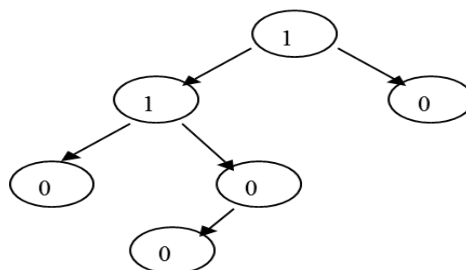
Características

- Más bajos que los montículos binarios altura $\log_d n$.
- Mejora la operación de insertar
- EliminarMin es más costosa $O(d \log_d n)$
- Se pueden implantar en arreglos
- Bueno cuando hay muchas inserciones y pocas eliminaciones
- Bueno cuando el tamaño del montículo es muy grande.

3.4 Montículos binarios a la izquierda

Longitud del camino nulo (lcn)

- Es la longitud del camino más corto entre **x** y un nodo hoja.
- Longitud del camino nulo de un nodo hoja con un hijo es 0.
- lcn(nulo) = -1 (longitud del camino nulo es -1)



Propiedad del montículo a la izquierda

1. La **lcn** del hijo izquierdo es al menos tan grande como la del hijo derecho
2. El árbol se desvía con mayor profundidad al lado izquierdo.

Un montículo a la izquierda posee dos propiedades:

- Propiedad estructural basada en la longitud del camino nulo
- Propiedad de orden (como el montículo binario)

Montículo a la izquierda

No

No cumple propiedad 1

lcn de cualquier nodo es 1 más que la mínima longitud del camino nulo de sus hijos

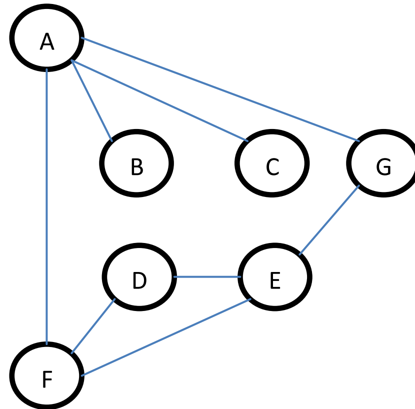
CAPITULO IV

Grafos I. Algoritmos fundamentales. Conectividad. Grafos ponderados

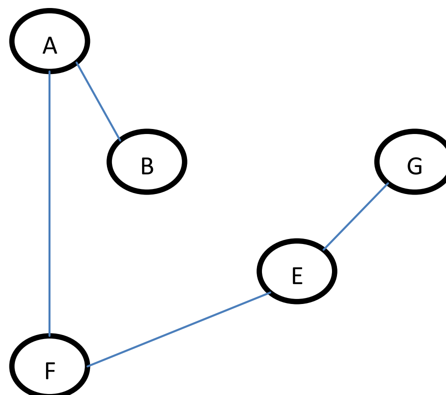
4.1 Algoritmos sobre grafos elementales

4.1.1 Glosario

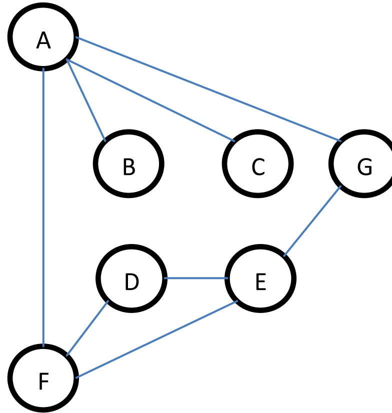
Grafo: Un grafo es una colección de vértices y aristas. Los vértices son objetos simples que pueden tener un nombre y otras propiedades; una arista es una conexión entre dos vértices.



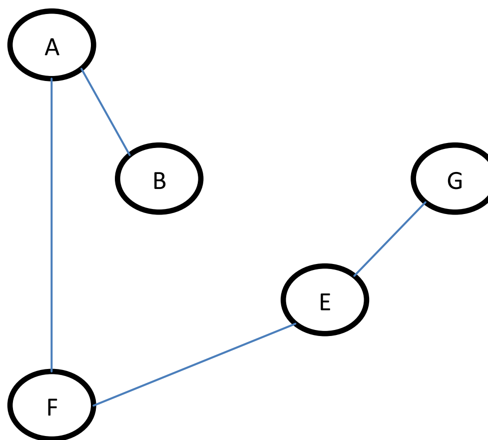
Camino: Un camino entre los vértices x e y de un grafo es una lista de vértices en la que dos elementos sucesivos están conectados por aristas del grafo.



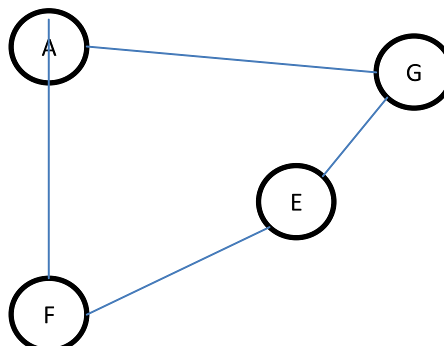
Grafo conexo: Un grafo es conexo si hay un camino desde cada nodo hacia otro nodo del grafo.



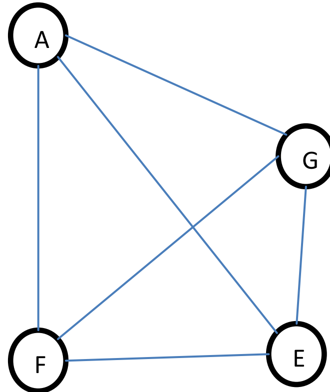
Camino simple: Es un camino en el que no se repite ningún vértice.



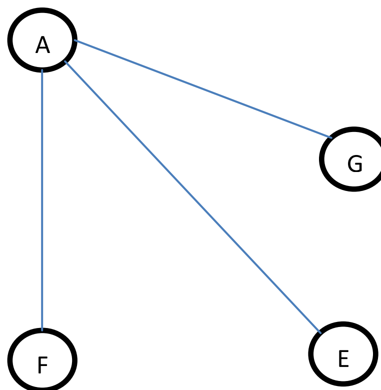
Ciclo: es un camino simple con la característica de que el primero y el último vértice son el mismo.



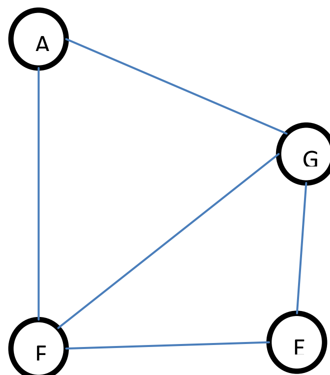
Grafo completo: Son los grafos con todas las aristas posibles.



Grafo disperso: Tienen relativamente pocas aristas.



Grafo denso: Les falta muy pocas aristas de todas las posibles.

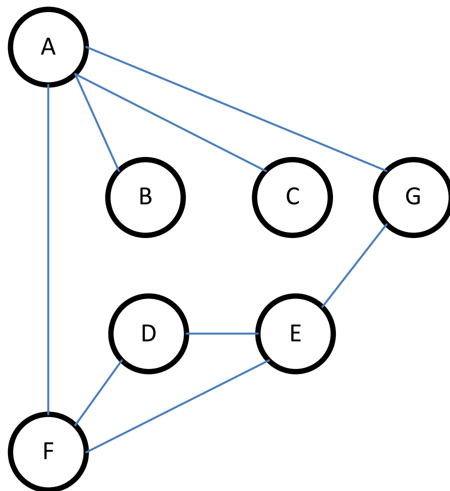


4.1.2 Representación

Matriz de adyacencia

Se construye un array de V*V valores booleanos en el que a[x][y] es igual a 1 si existe una arista desde el vértice x al y, y a 0 en el caso contrario.

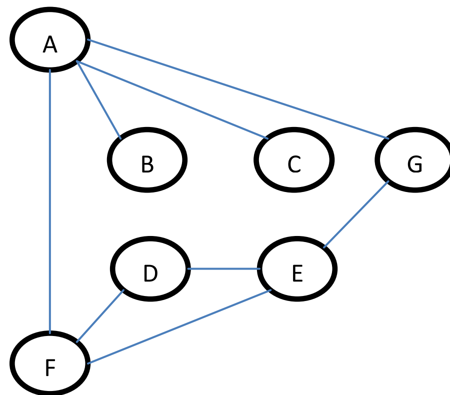
Es de destacar que en realidad cada arista se representa con dos bits: una arista que enlace x e y se representa con valores verdaderos tanto en a[x][y] como en a[y][x].

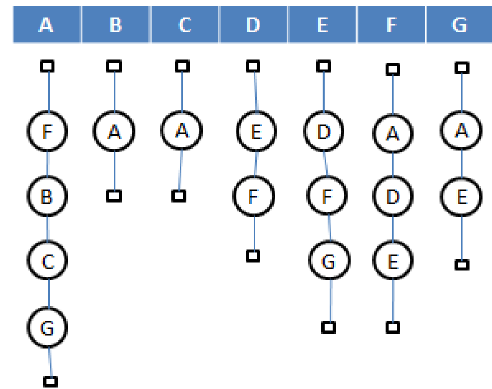


	A	B	C	D	E	F	G
A	1	1	1	0	0	1	1
B	1	1	0	0	0	0	0
C	1	0	1	0	0	0	0
D	0	0	0	1	1	1	0
E	0	0	0	1	1	1	1
F	1	0	0	1	1	1	0
G	1	0	0	0	1	0	1

Lista de adyacencia

La lista de adyacencia se adapta mejor a lo casos en los que los grafos a procesar no son densos.



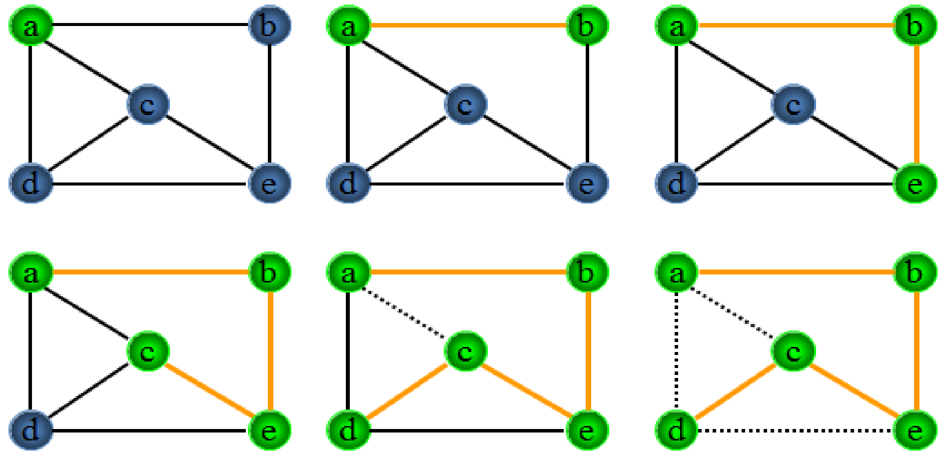


4.1.3 Recorridos en un grafo

En una gran cantidad de problemas con grafos, es necesario visitar sistemáticamente los vértices y aristas del grafo. La búsqueda en **PROFUNDIDAD** y en **AMPLITUD**, son dos técnicas importantes de recorrido del grafo.

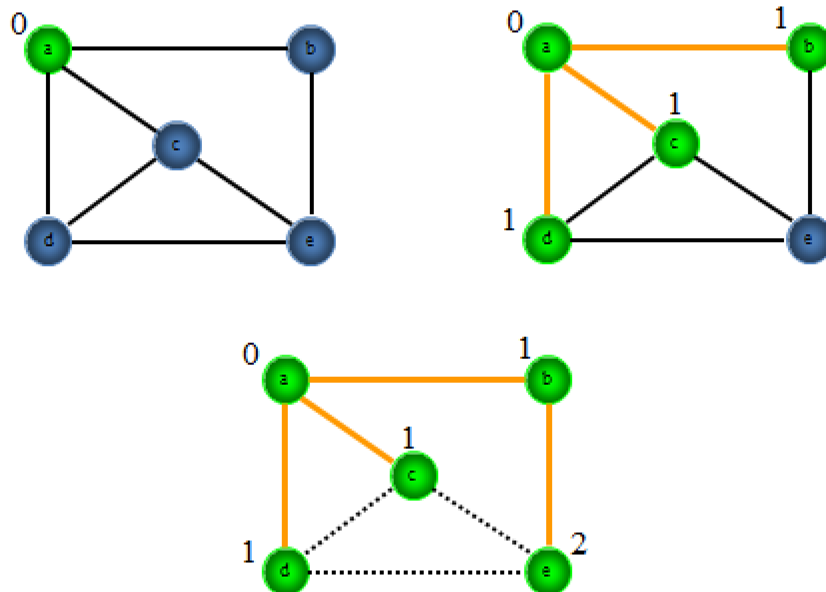
Búsqueda en Profundidad (BEP)

Se comienza en el vértice inicial (vértice con índice 1) que se marca como vértice activo. Hasta que todos los vértices hayan sido visitados, en cada paso se avanza al vecino con el menor índice siempre que se pueda, pasando este a ser el vértice activo. Cuando todos los vecinos al vértice activo hayan sido visitados, se retrocede al vértice **X** desde el que se alcanzó el vértice activo y se prosigue siendo ahora **X** el vértice activo.



Búsqueda en Amplitud (BEA)

Se comienza en el vértice inicial (vértice con índice 1) y se marca como vértice activo, a diferencia con la BEP ahora se visitan en orden creciente de índice todos los vecinos del vértice activo antes de pasar al siguiente. Hasta que todos los vértices hayan sido visitados, en cada paso se van visitando en orden creciente de índice todos los vecinos del vértice activo. Cuando se han visitado todos los vecinos del vértice activo, se toma como nuevo vértice activo el primer vértice **X** visitado después del actual vértice activo.

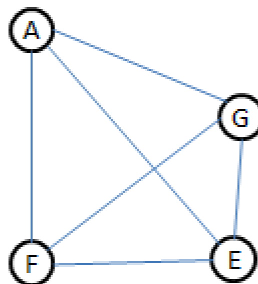


4.2 Conectividad

4.2.1 Problemática

Se examinara una generalización de la conectividad denominada biconectividad, cuyo interés reside en conocer si **hay más de un medio de pasar de un vértice de un grafo a otro**.

Grafo biconexo: Un grafo es biconexo, si solo si, existen al menos dos caminos diferentes que conecten cada par de vértices. De esta forma si se suprime un vértice y todas las aristas que inciden en el, el grafo permanece conexo.



El problema se denomina a veces como “unión-pertenencia”.

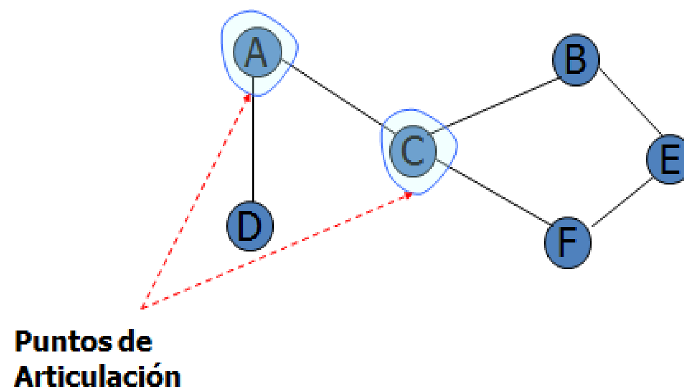
4.2.2 Puntos de articulación

Un punto de articulación en un grafo conexo es un vértice que si se suprime romperá el grafo en dos o más piezas.

En un grafo no dirigido conexo**:**

- Existen vértices que si se eliminan

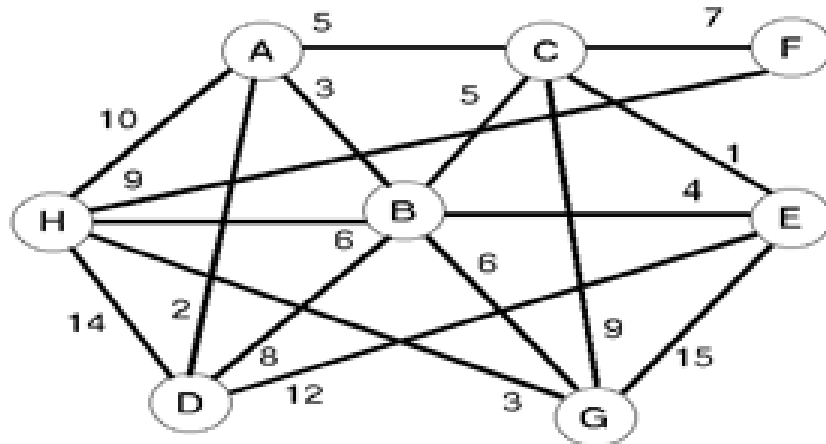
  - "Desconectan" al Grafo
  - Lo dividen en componentes conexas
- Estos vértices "clave" son puntos de articulación
- Si un grafo no tiene Punto de Articulación
  - Es biconexo
- La conectividad de un grafo
  - Es el número de nodos que se necesitan eliminar para dejar a un grafo "desconectado"



4.3 Grafos ponderados

4.3.1 Problemática

Con frecuencia se desea modelar problemas prácticos utilizando grafos en los que se asocia a las aristas unos pesos o costes. En un mapa de líneas aéreas, en el que las aristas representan rutas de vuelo, los pesos pueden representar distancias o tarifas. Estas situaciones hacen aparecer de forma natural cuestiones como el **minimizar costes**.



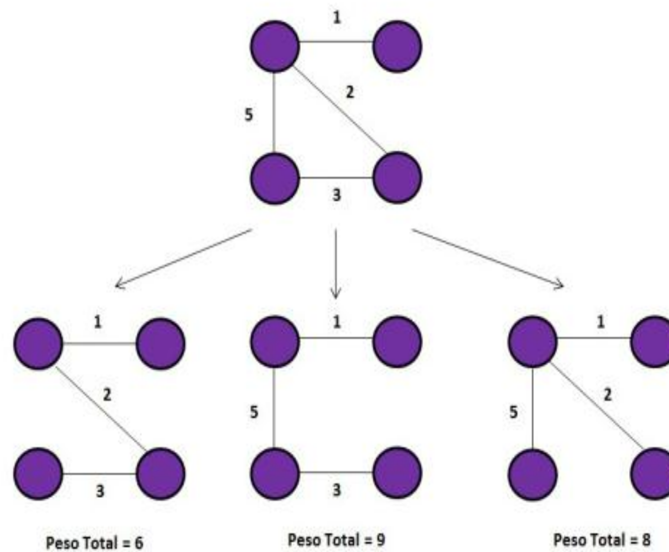
Se examinara los algoritmos de dos de estos problemas:

1. Encontrar la forma de conectar todos los puntos al menor coste (**problema del árbol de expansión mínima**).
2. Encontrar el camino de menor coste entre dos puntos dados (**problema del camino más corto**).

La forma de representar a los grafos ponderados es obvia. En la representación por matriz de adyacencia, la matriz puede contener pesos de aristas en lugar de valores booleanos y en la representación por listas de adyacencia se puede añadir un campo a cada elemento de la lista, a manera de peso.

Árbol de expansión mínimo

Un árbol de expansión mínimo de un grafo ponderado es una colección de aristas que conectan todos los vértices y en el que la suma de los pesos de las aristas es al menos inferior a la suma de los pesos de cualquier otra colección de aristas que conecten todos los vértices.



Algoritmo genérico

Se puede construir el árbol de expansión mínimo comenzando en cualquier vértice y tomando siempre el vértice más próximo de todos los que se hayan elegido. En otras palabras, se busca la arista de menor peso entre todas las que conectan vértices que ya están en el árbol con vértices que no lo están todavía, y después se añade al árbol la arista y el vértice a los que conduce la anterior.

Algoritmo de Kruskal

Se puede construir el árbol de expansión mínimo comenzando en cualquier vértice y tomando siempre el vértice más próximo de todos los que se hayan elegido. En otras palabras, se busca la arista de menor peso entre todas las que conectan vértices que ya están en el árbol con vértices que no lo están todavía, y después se añade al árbol la arista y el vértice a los que conduce la anterior.

Camino más corto

El problema del camino más corto consiste en encontrar entre todos los caminos que conectan a dos vértices x e y dados de un grafo ponderado el que cumple con la propiedad de que la **suma de las ponderaciones de todas las aristas sea mínima**. Si las ponderaciones son todas iguales a 1, entonces el problema sigue siendo interesante: ahora consiste en encontrar el camino que contenga al mínimo de aristas que conecten a x e y.

Algoritmo de Dijkstra

El algoritmo de Dijkstra determina la ruta más corta desde un nodo origen hacia los demás nodos para ello es requerido como entrada un grafo cuyas aristas posean pesos.

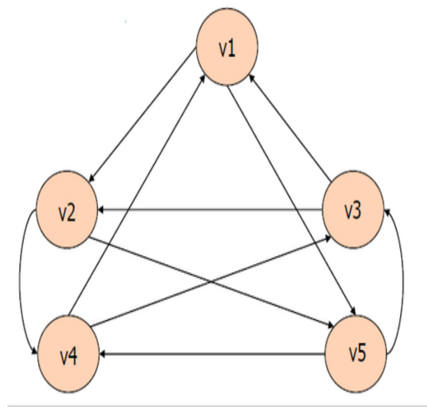
CAPITULO V

Grafos II. Grafos dirigidos. Flujo de red.

5.1 Grafos dirigidos

Definición

Son aquellos en los que las aristas que conectan los nodos son de sentido único. También se les denomina como DIGRAFOS. A menudo, la dirección de las aristas reflejan algún tipo de relación de precedencia en la aplicación que se está modelando. Por ejemplo, un grafo dirigido puede utilizarse como modelo para una cadena de fabricación, los nodos corresponden a las tareas a realizar.

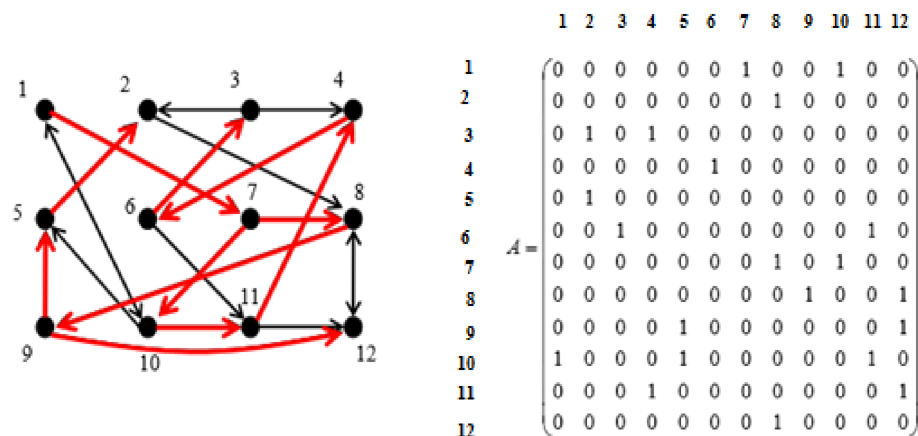


El orden en el que aparecen los vértices al especificar las aristas tiene mucha importancia: la notación V_1V_2 describe una arista que apunta de V_1 hacia V_2, pero no de V_2 hacia V_1.

5.1.1 Búsqueda en profundidad

Los algoritmos de búsqueda en profundidad estudiados anteriormente pueden ser utilizados exactamente de igual forma con los grafos dirigidos.

De hecho, opera de una manera un poco más directa que con los grafos no dirigidos dado que no se necesita tener en cuenta las dobles aristas entre nodos, a menos que estén incluidas explícitamente en el grafo.



$$A = \begin{pmatrix} 0&0&0&0&0&0&1&0&0&1&0&0 \\ 0&0&0&0&0&0&0&1&0&0&0&0 \\ 0&1&0&1&0&0&0&0&0&0&0&0 \\ 0&0&0&0&0&1&0&0&0&0&0&0 \\ 0&1&0&0&0&0&0&0&0&0&0&0 \\ 0&0&1&0&0&0&0&0&0&0&1&0 \\ 0&0&0&0&0&0&0&1&0&1&0&0 \\ 0&0&0&0&0&0&0&0&1&0&0&1 \\ 0&0&0&0&1&0&0&0&0&0&0&1 \\ 1&0&0&0&1&0&0&0&0&0&1&0 \\ 0&0&0&1&0&0&0&0&0&0&0&1 \\ 0&0&0&0&0&0&0&1&0&0&0&0 \end{pmatrix}$$

1	2	3	4	5	6	7	8	9	10	11	12
7 10	8	2 4	6	2	3 11	8 10	9 12	5 12	1 5 11	4 12	8

1 7 8 9 5 2 12 10 11 4 6 3

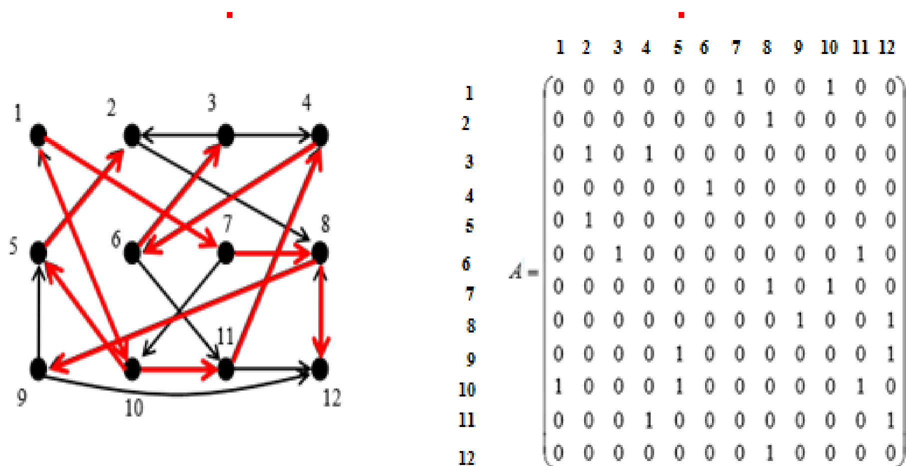
Como en el caso de los grafos no dirigidos, existe gran interés en conocer las propiedades de conectividad de los grafos dirigidos. Debería ser posible responder a preguntas tales como:

¿Existe un camino dirigido desde el vértice **x** al vértice **y**?,

¿Qué vértices son accesibles desde el vértice **x** por medio de un camino dirigido? y

¿Existe un camino dirigido desde el vértice **x** al vértice **y** y un camino dirigido desde **y** a **x**?

5.1.2 Búsqueda en Amplitud



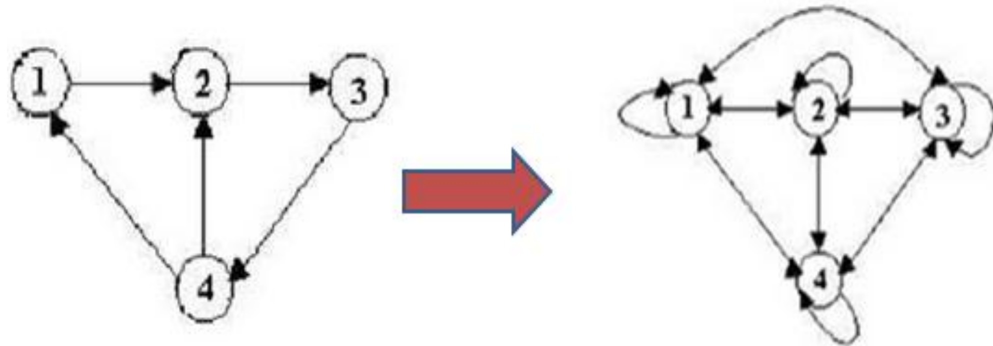
1	2	3	4	5	6	7	8	9	10	11	12
7 10	8	2 4	6	2	3 11	8 10	9 12	5 12	1 5 11	4 12	8

1 7 10 8 5 11 9 12 2 4 6 3

5.1.3 Clausura transitiva

El concepto de transitividad es el mismo que se utiliza en la teoría matemática, el cual postula que si **a < b < c ➔ a < c**, en el caso de la clausura transitiva, el concepto se refiere a que existe un camino que une el vértice x con el vértice y, no importando que para llegar desde x a y tengamos que visitar otros vértices del grafo (**camino más corto**).

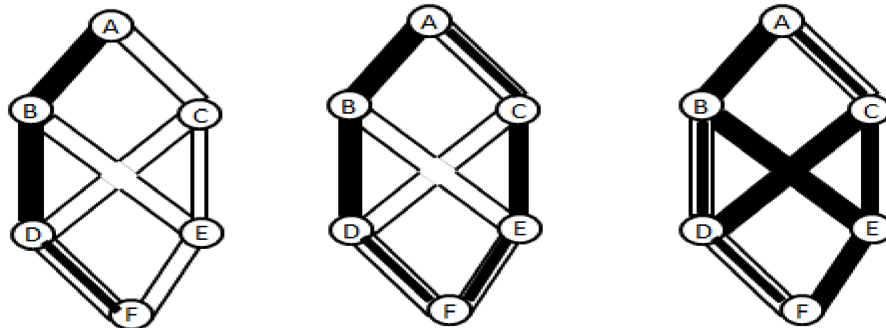
Consiste en completar el grafo con todas las aristas que "**acortan**" el camino entre dos nodos **x** y **y**, si es que se puede llegar de **x** a **y**.



En digrafos muy grandes, el problema de encontrar caminos entre pares de vértices es demasiado complicado, sólo siguiendo el trazado del dibujo del mismo, de ahí la importancia de este algoritmo y el interés que generó su estudio.

5.2 Flujo de red

5.2.1 El problema del flujo de red



En la gráfica idealizada anterior de la pequeña red de oleoducto, las canalizaciones tienen una capacidad fija proporcional a su tamaño y el petróleo solamente puede fluir en forma descendente. Además las válvulas de cada derivación controlan la capacidad de petróleo que va en cada dirección.

Sin importar la forma en la que estén puestas las válvulas, el sistema alcanza un **estado de equilibrio** cuando la capacidad de petróleo que

fluye al sistema por arriba es igual a la cantidad que fluye hacia afuera por la parte inferior y cuando la cantidad de petróleo que fluye hacia cada derivación es igual a la que sale de ella.

El objetivo es desarrollar un algoritmo que pueda encontrar el reglaje de las válvulas “adecuado” para cualquier red y de esta manera evitar “embotellamientos”. Además, se desea estar seguros de que ningún otro reglaje dará un flujo mayor.

Red:
Se define como un grafo dirigido ponderado con dos vértices principales: uno, que no tiene aristas que apunte a el (la fuente), y otro que no tiene aristas que apunten afuera de el (el pozo). Los pesos de las aristas, que se supone que no son negativos, se denominan **capacidades de las aristas**.

Flujo:
Se define como un conjunto de pesos en las aristas tal que el flujo en cada una de ellas es igual o menor que la capacidad, y el flujo que entra en cada vértice es igual al que sale de el. El valor del flujo es el que entra en la fuente (o sale del pozo).

“El problema del flujo de red consiste en encontrar un FLUJO DE VALOR MÁXIMO para una red dada”

5.2.2 Método de Ford-Fulkerson

El algoritmo de Ford-Fulkerson propone buscar caminos en los que se pueda aumentar el flujo, hasta que se alcance el flujo máximo. **La idea es encontrar una ruta de penetración con un flujo positivo neto que una los nodos origen y destino**.

Consideraremos las capacidades iniciales del arco que une el nodo **i** y el nodo **j** como **Cij** y **Cji**. Estas capacidades iniciales irán variando a medida que avanza el algoritmo, denominaremos capacidades residuales a las capacidades restantes del arco una vez pasa algún flujo por él, las representaremos como **cij** y **cji**.

Para un nodo **j** que recibe el flujo del nodo **i**, definimos una clasificación **[aj,i]** donde **aj** es el flujo del nodo **i** al nodo **j**.

PASOS:

1. Identificar los nodos origen y destino
2. Identificar la capacidad mas alta que sale del nodo origen
3. Identificar el nodo intermediario con $[a_f,i]$ ($\mathbf{a_f}$ es el flujo máximo de ingreso y **i** el nodo de donde proviene dicho flujo máximo)
4. Repetir paso 3, como si el nodo intermediario fuera el nodo origen
5. Actualizar los flujos: C_{ij} , C_{ji} = (C_i – k, C_j + k) donde:
   C = Capacidad
   i , j = índices de los nodos
   K = flujo mínimo del camino seleccionado
6. Retornar al paso 2 si al menos hay una salida de flujo del nodo fuente
7. La suma de los **K** corresponde al Flujo Máximo.

REFERENCIAS BIBLIOGRAFICAS

1. Ormen, Thomas H. “Introduction to Algorithms”. Ed. Mc Graw-Hill. 2001. 2da. Edición. EEUU.

2. William Stalings. “Cryptography and Network Security: Principles and Practice”. Ed. Prentice Hall. 2002. 3ra. Edición. EEUU.

3. Wang Paul S. “Java: con Programación Orientada a Objetos y Aplicaciones en la WWW”. Ed. Internacional Thomson. 2000. España.

4. Stuart McClure, Joel Sambay and George Kurtz. “Hacking Exposed: Network Security Secrets and Solution”. Ed. Osborne Mc Graw-Hill. 3ra. Edición. 2002. EEUU.

5. Sergio M. Fernandez S. “Fundamentos del diseño y la programación orientado a objetos”. Ed. Mc Graw Hill. 1995. España.