01168
2ej. 2.

DIVISION DE ESTUDIOS DE POSGRADO
FACULTAD DE INGENIERIA

ESTRUCTURA DE DATOS APLICADA A REDES DE FLUJO

MARIO GUTIERREZ LAGUNES

TESIS

Presentada a la División de Estudios de
Posgrado de la
FACULTAD DE INGENIERIA
de la
UNIVERSIDAD NACIONAL AUTONOMA DE MEXICO
como requisito para obtener
el grado de

MAESTRO EN INGENIERIA
INVESTIGACION DE OPERACIONES

CIUDAD UNIVERSITARIA
OCTUBRE, 1988

TESIS CON
FALLA DE ORIGEN

Universidad Nacional
Autónoma de México

Biblioteca Central

Dirección General de Bibliotecas de la UNAM

UNAM – Dirección General de Bibliotecas
Tesis Digitales
Restricciones de uso

DERECHOS RESERVADOS ©
PROHIBIDA SU REPRODUCCIÓN TOTAL O PARCIAL

Todo el material contenido en esta tesis esta protegido por la Ley Federal del Derecho de Autor (LFDA) de los Estados Unidos Mexicanos (México).

El uso de imágenes, fragmentos de videos, y demás material que sea objeto de protección de los derechos de autor, será exclusivamente para fines educativos e informativos y deberá citar la fuente donde la obtuvo mencionando el autor o autores. Cualquier uso distinto como el lucro, reproducción, edición o modificación, será perseguido y sancionado por el respectivo titular de los Derechos de Autor.

ESTRUCTURA DE DATOS APLICADA A REDES DE FLUJO

INDICE

Introducción General. 1

Capítulo I. Análisis de un Algoritmo.
1. Introducción. 1
2. Definición de Algoritmo. 2
3. Complejidad Computacional. 6
4. Estudio de un Algoritmo. 21
5. Algoritmos de Ordenación. 26
6. Análisis de la Ordenación Rápida. 28

Capítulo II. Estructuras de Datos Básicas.
1. Introducción. 30
2. La Necesidad para Estructurar en Conjuntos de Datos. 31
3. Estructuras de Datos. 32
4. Estructuras de Datos Estáticas y Dinámicas. 37
5. Estructura Modular de Programas. 62
6. Recursividad. 64

Capítulo III. Estructura de Datos en Redes.
1. Introducción. 70
2. Conceptos Básicos. 72
3. Arboles Binarios. 75
4. Búsqueda Binaria. 77
5. Búsqueda Secuencial. 79
6. Montículos. 80
7. Análisis de la Ordenación por Montículos. 89

CAPÍTULO IV. PROBLEMAS DE APLICACIÓN.
1. INTRODUCCIÓN. 90
2. CONECTIVIDAD DE UNA RED. 91
3. FLUJO MÁXIMO. 94
4. CORTE MÍNIMO. 99
5. PINTADO DE LA RED EN FLUJO MÁXIMO. 104

CAPÍTULO V. PROGRAMAS DESARROLLADOS.
1. CONECTIVIDAD DE UNA RED. 110
2. FLUJO MÁXIMO. 114
3. PINTADO DE LA RED EN FLUJO MÁXIMO. 117

CAPÍTULO VI. CONCLUSIONES. 123

APÉNDICE :
PRUEBA DE HIPÓTESIS ENTRE LA DIFERENCIA DE MEDIAS DE LOS ALGORITMOS DE FORD-FULKERSON Y EL PINTADO DE LA RED PARA FLUJO MÁXIMO. 126

ANEXOS :
PROGRAMA: CONECTIVIDAD DE UNA RED. 130
PROGRAMA: FLUJO MÁXIMO. 134
PROGRAMA: PINTADO DE LA RED. 139

BIBLIOGRAFIA.

INTRODUCCION

La información es sólo un proceso de intercambio. El proceso de recibir y utilizar información consiste en ajustarnos a las contingencias de nuestro medio y de vivir de manera efectiva dentro de él. Las necesidades y la complejidad de la vida moderna plantean este fenónemo del intercambio de información demandas más intensas que en cualquier otra época; la prensa, los museos, los laboratorios científicos, las universidades, las bibliotecas y los libros de texto han de satisfacerlas. Vivir de manera efectiva significa poseer la información adecuada. El lugar que ocupa en la actualidad el análisis de datos no es trivial, ni fortuito, ni nuevo, por lo que se ha generado un avance paralelo en la computación para satisfacer las necesidades que nos implica la diaria existencia.

Se puede decir que la década de los sesenta en el campo del "software" fue la época de los lenguajes, ya que en ella se difundió el Fortran (diseñado en 1954), y se desarrollaron otros lenguajes considerados fundamentales, como Algol 60 y 68, COBOL, PL/1, etc. En la década de los 70's hubo un cambio,y se desarrolla la programación estructurada que, encabezado por E.W. Dijkstra, puso de relieve la dificultad que la creciente complejidad de los lenguajes representaba para la construcción de los programas, proponiendo como nuevo criterio de valoración de un lenguaje sus posibilidades para crear, a partir de sus componentes básicos, programas fiables e inteligibles. Con base a estas ideas, se constituyó el diseño del lenguaje PASCAL, realizado por N. Wirth (1971), el cual también participó con su metodología de refinamiento gradual para la construcción de programas.

Por otra parte, la extensa actividad conocida por Investigación de Operaciones ha tenido un avance también explosivo, debido principalmente porque la gente de esta Área ha estado trabajando en diversas circunstancias. Recuérdese el inicio de la Investigación de Operaciones, lo que aportaron los científicos a los problemas operativos fue su aspecto científico. Y de hecho, esta fue su contribución principal. Aportaron ideas nuevas, desconfiaron de los conceptos preconcebidos y obraron sólo ante la evidencia. En la actualidad, los nuevos problemas no son tan accesibles como lo fueron muchas de las operaciones militares.

El objetivo general de este trabajo es sintetizar y describir los conceptos fundamentales de la estructura de datos, y su aplicación a redes de flujo, con particular énfasis en la complejidad computacional como una medida de eficiencia de los algoritmos. Otro aspecto fundamental de este trabajo es la implantación de algunos algoritmos típicos, en especial, el algoritmo del pintado de la red para la solución del Flujo Máximo. Cabe mencionar que este algoritmo dibuja y pinta los arcos de la red, además de hallar el Flujo Máximo.

Este trabajo pretende unificar las ideas trascendentales que existen sobre el tema de algoritmos, sus principios fundamentales y haciendo hincapié en los conceptos de diseño de algoritmos para que sean más fácilmente pensados. Así, el decidir es propio del hombre, ya que la decisión implica la selección consciente entre varias soluciones posibles.

DESCRIPCION.

En el capítulo uno, Análisis de un Algoritmo, se muestra todo lo relacionado para que se realice un análisis completo de un algoritmo, sus ventajas y su relevancia en la época actual. En el capítulo dos, Estructuras de Datos Básicas, se describen las principales estructuras de datos y su importancia que tienen para la eficiencia de algoritmos. En el capítulo tres, Estructura de Datos en Redes se definen algunos conceptos en la teoría de redes, y se da el análisis de ellos. En el capítulo cuatro, Problemas de Aplicación, se tienen algunos problemas de aplicación como son la conectividad de una red, el flujo máximo (Ford-Fulkerson) y el Pintado de la Red, también para obtener el Flujo Máximo. En el capítulo cinco, Programas Desarrollados, se describen los programas desarrollados para las aplicaciones anteriores. Por último, en el capítulo seis, Conclusiones, se dan las conclusiones que tuvo este trabajo, señalando su contribución hacia la comunidad y sus posibles usos.

Capítulo I. Análisis de un Algoritmo.

1. Introducción.
2. Definición de Algoritmo.
3. Complejidad Computacional.
4. Estudio de un Algoritmo.
5. Algoritmos de Ordenación.
6. Análisis de la Ordenación Rápida.

INTRODUCCION

El estudio de algoritmos es el corazón de la ciencia de la computación. En los últimos años se ha caracterizado por un avance significativo en el campo de los algoritmos; dichos avances se han desarrollado principalmente hacia algoritmos más rápidos, así como a ciertos problemas para los cuáles los algoritmos eran ineficientes.

Los resultados de estos avances han sido de gran interés en el estudio de los algoritmos, substancialmente en el área de diseño y análisis de algoritmos. El uso de modelos matemáticos en la actualidad es cada vez mayor, y naturalmente, los algoritmos usados deben de ser buenos, por lo que es necesario medir su eficiencia, independientemente del equipo que se use, así como de su comprensión y claridad. Este capítulo trata estos temas, y valoriza la importancia que tienen los algoritmos en la resolución de problemas.

DEFINICION DE ALGORITMO.

La notación de un algoritmo es básica para todo programa de computación, y así comenzamos con un análisis cuidadoso de este concepto.

La palabra 'algoritmo' es ella misma interesante; en primer instancia puede ser pensada como un intento para escribir 'logaritmo' pero en diferente orden las primeras cuatro letras.

La palabra algoritmo aparece después de 1957 en el Webster's New World Dictionary; primero se tenía noticia de la forma de 'algorismo', que tiene un significado antiguo, es decir, es el proceso de hacer aritmética usando métodos arábigos. Siguiendo en la edad media, el origen de la palabra algorismo fue dudosa, pretendiendo adivinar su derivación haciendo combinaciones como algiros (trabajoso) + arithmos (número); otras personas opinan que la palabra vino de "King Algor of Castile".

Por último, los historiadores de las matemáticas encontraron que el origen verdadero de la palabra algorismo vino del nombre de un famoso autor persa, Abu Ja'far Mohammed ibn Musa al-Khowarizmi,que literalmente dice: "Padre de Ja'far, Mohammed,hijo de Moses,nativo de Khowarizm". Al-Khorarizmi escribió el libro célebre "Kitabaljabr w'al-muqabala" (reglas de restauración y reducción); otra palabra,'álgebra',capítulo de dicho libro, aunque no fue muy algebraico, dió origen al álgebra, tal como se le conoce en la actualidad.

Gradualmente la forma y significado de 'algorismo' se fue modificando, fue 'erróneamente ajustada' por confusión erudita con la palabra aritmética. El cambio de algorismo por algoritmo no es difícil de comprender en vista que, de hecho, la gente olvidó la derivación original de la palabra.

En 1747, un diccionario alemán de matemáticas da la definición de Algorithmus: "Bajo esta designación son combinadas las nociones de los 4 tipos de cálculos aritméticos: adición, multiplicación, sustracción y división". Asimismo, la frase en latín "algorithmus infinitesimalis" fue usada como 'los caminos de cálculo con cantidades muy pequeñas, inventadas por Leibnitz'.

Por 1950, la palabra algoritmo fue más frecuentemente asociada con el 'algoritmo de Euclides' como un proceso para encontrar el Máximo Común Divisor de 2 números enteros positivos.

IMPORTANCIA DEL ALGORITMO.

El moderno significado de algoritmo es completamente similar al de fórmula, proceso, método, técnica, procedimiento, rutina, etc., excepto que la palabra algoritmo connota algunas cosas un poco diferentes. En esencia, es un conjunto de reglas que da una secuencia de operaciones para resolver un tipo específico de problema.

Un algoritmo tiene 5 características importantes:

1. FINITO. Un algoritmo debe siempre terminar después de un número finito de pasos.
2. DEFINICION. Cada paso de un algoritmo debe estar bien definido. Las acciones a ser llevadas a cabo deben ser rigurosamente y sin ambiguedad especificadas para cada paso.
3. ENTRADA. Un algoritmo tiene cero o más entradas, es decir, cantidades que son dadas para inicializarse antes de que el algoritmo comience. Estas entradas son tomadas de conjuntos específicos de objetos.
4. SALIDA. Un algoritmo tiene una o más salidas, es decir, cantidades que tienen una relación específica con las entradas.
5. EFECTIVIDAD. Un algoritmo es también generalmente pensado para ser efectivo. Esto significa que todas las operaciones a ser ejecutadas en el algoritmo deben ser suficientemente básicas, tales que ellas puedan desde un principio estar dadas exactamente y en un tiempo finito por el hombre usando papel y lápiz.

Se hace la observación que la característica de la de 'finito' debe ser fuertemente usada. Un algoritmo útil debe requerir no solamente un número finito de pasos, sino un número finito de pasos razonable. Por ejemplo, existe un algoritmo que determina cuando o no el juego de ajedrez es una victoria forzada por las piezas blancas. Aquí está un algoritmo que puede resolver un problema de interés grande para la gente, salvo que nunca en nuestro tiempo de vida se conocerá la respuesta a este problema, ya que el algoritmo requiere una cantidad fantásticamente grande de tiempo para su ejecución, aunque el sea "finito".

En la práctica no sólo se quieren algoritmos, sino que se desean buenos algoritmos en algún sentido. Un criterio de bondad es la longitud de tiempo que tarda la ejecución del algoritmo, esto puede ser expresado en términos del número de veces que cada paso es ejecutado. Otros criterios pueden ser la adaptabilidad del algoritmo a las computadoras, su simplicidad, su elegancia, etc.

En ocasiones se tienen varios algoritmos para el mismo problema, y se tiene que decidir cuál es el mejor. Esto nos permite introducirnos al importante campo del análisis de algoritmos, es decir, dado un algoritmo, el problema consiste en determinar sus características de ejecución.

"Análisis de algoritmos" es el nombre que se da para describir investigaciones como estas. La idea general es el de tomar un algoritmo particular y determinar su comportamiento promedio, así también cuando o no un algoritmo es óptimo en algún sentido.

COMPLEJIDAD COMPUTACIONAL.

Para estudiar la eficiencia de algoritmos, se necesita un modelo de computación. Una posibilidad es la de desarrollar la definición de complejidad en términos operacionales.

Históricamente, el primer modelo de máquina propuesto fue la máquina de Turing. En su forma simple, una máquina de Turing consiste de un estado de control finito, un camino bipartito de memoria infinita en la cinta, dividida en cuadros, cada uno de los cuales puede tomar uno de un número finito de símbolos, y leer/escribir en la cabecera. En un paso la máquina puede leer el contenido de una casilla de cinta, escribir un nuevo símbolo en la casilla, mover la cabeza una casilla a la izquierda o a la derecha, y cambiar el estado de control.

La simplicidad de las máquinas de Turing hacen que sean muy útiles en un alto nivel teóricamente .de la complejidad computacional, pero no son lo bastante realista para permitir análisis correctos de algoritmos prácticos. Para este propósito un mejor modelo es la máquina de acceso aleatorio. Una máquina de acceso aleatorio consiste de un programa finito, una colección finita de registros, en la que cada uno puede guardar un entero sencillo o un número real, y una memoria consistente de un arreglo de n palabras, cada una de las cuales tiene una única dirección entre 1 y n (inclusive), y puede tomar un número entero sencillo o un número real. En un paso, una máquina de acceso aleatorio puede ejecutar una operación aritmética sencilla u operación lógica sobre los contenidos especificados de registros, busca en un registro específico el contenido de una palabra cuya dirección está en un registro, o guarda el contenido de un registro en una palabra cuya dirección está en un registro.

Las máquinas arriba mencionadas tienen dos propiedades: son secuenciales, y son determinísticas, esto es, el comportamiento futuro de la máquina es únicamente determinado por su configuración presente.

Habiendo tomado un modelo de máquina, se debe ahora seleccionar una medida de complejidad. Una posibilidad es medir la complejidad de un algoritmo por la longitud de sus programas. Esta medida es estática, es decir, independiente de los valores de entrada. La longitud del programa es la medida relevante si un algoritmo sólo se va a correr una o pocas veces, y esta medida tiene un uso teórico interesante, sin embargo, para nuestro propósito, una mejor medida de complejidad es dinámica, tal como el tiempo de ejecución o el espacio de memoria como una función del tamaño de la entrada.

Se usa el tiempo de ejecución como nuestra medida de complejidad, asimismo se considera que la mayoría de los algoritmos tienen un espacio limitado que tiene una función lineal del tamaño de la entrada.

En el análisis del tiempo de ejecución se ignoran los factores constantes. Esto no sólo simplifica el análisis, sino permite ignorar detalles del modelo de máquina, dando así una medida de complejidad que es independiente de la máquina. En la figura 1.1 se ilustra para problemas de tamaño bastante grande la eficiencia relativa de dos algoritmos que depende de su tiempo de ejecución como una función asintótica del tamaño de entrada, independientemente de los factores constantes. Aunque el término 'bastante grande' es relativo; para algunos problemas que se pueden resolver por métodos sencillos, por ejemplo por una multiplicación de matrices, la mayor eficiencia asintótica conocida de los algoritmos descarta tales métodos simplistas para problemas de tamaño astronómico.

Se usa la siguiente notación para tiempos de ejecución asintótica: si f y g son funciones con variables no-negativas $n, m, \ldots$, se dice que f es $O(g)$ si existen constantes positivas c_1 y c_2 tales que

Así también, se dice que f es $\Omega(g)$ si g es $O(f)$, y por último, se dice que f es $\Phi(g)$ si f es $O(g)$ y $\Omega(g)$.

TAMAÑO / COMPLEJIDAD	20	50	100	200	500	1000
1000n	.02 seg	.05 seg	.1 seg	.2 seg	.5 seg	1 seg
1000nlgn	.09 seg	.3 seg	.6 seg	1.5 seg	4.5 seg	10 seg
$100n^2$	.04 seg	.25 seg	1 seg	4 seg	25 seg	2 min
$10n^3$	.02 seg	1 seg	10 seg	1 min	21 min	2.7 hrs
n^{lgn}	.4 seg	1.1 hrs	220 días	125 siglos	5×10^8 siglos	
$2^{n/3}$	.0001 seg	.1 seg	2.7 hrs	3×10^4 siglos		
2^n	1 seg	35 años	3×10^4 siglos			
3^n	58 min	2×10^9 siglos				

Figura 1.1 Estimación del tiempo de corrida.

Es importante señalar que una computadora realiza en promedio un millón de instrucciones por segundo.

Generalmente se debe medir el tiempo de ejecución de un algoritmo como una función del peor caso de datos de entrada. Un análisis provee una garantía de ejecución, pero puede dar una estimación muy pesimista de la ejecución actual si el peor caso ocurre rara vez. Una alternativa es un análisis del caso promedio sobre las entradas posibles, sin embargo tal análisis es generalmente más difícil que el análisis del peor caso, y se debe tener cuidado en que nuestra probabilidad de distribución refleje fielmente la realidad.

Un primer tipo de promedio del tiempo es la amortización. La amortización es apropiada en situaciones en donde los algoritmos particulares son aplicados repetidamente, como ocurre con las operaciones sobre estructuras de datos. Para promediar el tiempo por operación sobre la secuencia de operaciones del peor caso, algunas veces se obtiene un tiempo arriba del límite que el tiempo del peor caso por operación multiplicado por el número de operaciones. Se usa esta idea repetidamente.

Por un algoritmo eficiente se entiende aquél cuyo tiempo de ejecución del peor caso es limitado por una función polinomial del tamaño de entrada. Los algoritmos eficientes usualmente corresponden a alguna estructura significativa en el problema, mientras que los algoritmos ineficientes frecuentemente buscan la cantidad de la fuerza bruta.

La mayoría de los problemas de optimización de redes son más sencillos que cualquier otro problema cuyo límite exponencial ha sido dado. Ellos son de la clase de problemas solubles en un tiempo polinomial en una máquina de Turing no determinística.

Una manera más intuitiva es aquella en que el problema pertenece a esa clase de problemas solubles si puede ser descrito por preguntas del tipo 'sí-no', tal que si la pregunta es 'sí' entonces existe una prueba del tamaño del polinomio para tal problema. Como por ejemplo, un problema de esta clase es el problema de la ruta mínima: dadas n ciudades y sus respectivas distancias entre ellas, encontrar una ruta que pase por cada ciudad una vez y cuya longitud total sea mínima. Se puede expresar este problema como preguntas del tipo 'sí-no' para responder si hay una ruta de longitud a lo más de x, y si es que existe, se puede verificar exhibiendo una ruta apropiada.

La teoría de la complejidad computacional puede dar información importante acerca del comportamiento práctico de los algoritmos, lo cual es fundamental para estar atentos a sus limitaciones.

O - GRANDE.

Una notación conveniente para proceder con aproximaciones fue introducida por Bachmann en 1892. Esta notación es la O-Grande la cual permite reemplazar el signo " $\cong$ " por " = ".

Por ejemplo:

$$Hn = \ln n + \gamma + O(1/n)$$

En general, la notación O(f(n)) puede ser usada cuando f(n) es una función de un entero positivo n; la O-Grande resiste una cantidad que no es explícitamente conocida, excepto que su magnitud no es muy grande. Todo aspecto de O(f(n)) significa esto:
existe una constante positiva M tal que el número Xn representado por O(f(n)) satisface la condición $|Xn| \leq M\ |f(n)|$ para todo $n \geq n_0$. Nótese que no se dice qué constantes M y n_0 son, y en realidad estas constantes son frecuentemente diferentes paca cada apariencia de O.

Así por ejemplo,

$$Hn = \ln n + \gamma + O(1/n) \text{ significa que}$$
$$|Hn - \ln n - \gamma| \leq M/n$$

la constante M no está especificada, pero aún si se desea conocer su valor, se sabe que la cantidad O(1/n) debe ser arbitrariamente pequeña si n es muy grande.

Veamos algunos ejemplos. Se tiene que

$$1^2 + 2^2 + \ldots + n^2 = (1/3)n(n + 1/2)(n + 1)$$
$$= (1/3)n^3 + (1/2)n^2 + (1/6)n$$

se sigue que

$$1^2 + 2^2 + \ldots + n^2 = O(n^3) \quad (1)$$

$$1^2 + 2^2 + \ldots + n^2 = (1/3)n^3 + O(n^2) \quad (2)$$

La ecuación (2) es más poderosa que la ecuación (1). Para justificar estas ecuaciones se debe probar que si

$$P(n) = a_0 + a_1 n + \ldots + a_m n^m$$

es cualquier polinomio de grado menor o igual a m, $P(n) = O(n^m)$. Esto es porque

$$|P(n)| \leq |a_0| + |a_1|n + \ldots + |a_m| n^m$$
$$= (|a_0| / n^m + |a_1| / n^{m-1} + \ldots + |a_m|) n^m$$
$$\leq (|a_0| + |a_1| + \ldots + |a_m|) n^m$$

cuando $n \geq 1$. Por lo que se puede tomar

$$M = |a_1| + |a_2| + \ldots + |a_m| \quad \text{y} \quad n_0 = 1 .$$

La notación O-Grande es una gran ayuda en la aproximación del trabajo, ya que brevemente describe un concepto que ocurre frecuentemente y suprime detalles de la información que son irrelevantes. Por consiguiente, la notación O-Grande puede ser manipulada álgebraicamente por caminos conocidos, teniendo sólo un poco de cuidado en cuanto a su utilización.

Varias de las reglas del álgebra pueden ser usadas al mismo tiempo con la notación O-Grande, pero ciertas diferéncias importantes deberán ser mencionadas. La principal consideración es la idea de un camino de igualdad:

se escribe $(1/2)n^2 + n = O(n^2)$

pero nunca se escribe

$$O(n^2) = (1/2)n^2 + n$$

O en otro caso, puesto que $(1/4)n^2 = O(n^2)$, se puede llegar a la relación absurda de $(1/4)n^2 = (1/2)n^2 + n$.

Siempre se usa la convención de que el lado derecho de una ecuación no debe dar más información que el lado izquierdo. Esta convención acerca del uso del signo de '=' puede ser establecida más precisamente como sigue: "Las fórmulas que involucren la notación $O(f(n))$ pueden ser vistas como conjuntos de funciones de n. El símbolo $O(f(n))$ es útil para el conjunto de todas las funciones g tales que existe una constante M con

$|g(n)| \leq M\,|f(n)|$ para toda n grande.

Si S y T son conjuntos de funciones, entonces

$$S + T \text{ denota al conjunto } \{ g + h \,/\, g \in S \text{ y } h \in T \}$$

De una manera similar se definen $S + c$, $S - T$, $S\,T$, $\log S$, etc.

Si $\alpha(n)$ y $\beta(n)$ son fórmulas que involucran la notación $O(f(n))$, entonces la notación $\alpha(n) = \beta(n)$ significa que el conjunto de funciones denotada por $\alpha(n)$ está contenida en el conjunto denotado por $\beta(n)$.

Consecuentemente, se pueden realizar la mayoría de las operaciones que se acostumbran hacer con el signo de '='.

Si $\alpha(n) = \beta(n)$ y $\beta(n) = \gamma(n)$, entonces $\alpha(n) = \gamma(n)$.

También, si $\alpha(n) = \beta(n)$ y si $\delta(n)$ es una fórmula resultado de la substitución de $\beta(n)$ por alguna ocurrencia de $\alpha(n)$ en una fórmula $\gamma(n)$, entonces $\gamma(n) = \delta(n)$.

Estas dos afirmaciones implican, por ejemplo, que si
$g(x_1, x_2, \ldots, x_m)$ es cualquier función real, y si
$\alpha_k(n) = \beta_k(n)$ para $1 \le k \le m$ entonces
$g(\alpha_1(n), \alpha_2(n), \ldots, \alpha_m(n)) = g(\beta_1(n), \beta_2(n), \ldots, \beta_m(n))$.

Algunas de las operaciones simples que se pueden realizar con la notación O-grande son las siguientes:

- $f(n) = O(f(n))$
- $c\ (O(f(n)) = O(f(n))$, si c es una constante.
- $O(f(n)) + O(f(n)) = O(f(n))$
- $O(O(f(n))) = O(f(n))$
- $O(f(n))\ O(g(n)) = O(f(n)g(n))$
- $O(f(n)g(n)) = f(n)O(g(n))$

La notación O-Grande es también usada en funciones de variable real x. Un rango particular de x es especificado, por ejemplo $a \leq x \leq b$, y se escribe $O(f(x))$ para soportar cualquier cantidad $g(x)$, tal que $|g(x)| \leq M\,|f(x)|$ cuando $a \leq x \leq b$.
(Como antes, M es una constante no especificada).

La notación $O(f(n))$ discutida anteriormente es en el caso especial donde la variable x es restringida a valores enteros positivos; usualmente se llama la variable n en lugar de x en este caso.

Supongamos que $g(x)$ es una función dada por una serie infinita

$$g(x) = \sum_{k \geq 0} a_k x^k \ , \ |x| \leq r$$

donde la suma de valores absolutos $\sum_{k \geq 0} |a_k x^k|$ existe.

Podemos entonces siempre escribir

$$g(x) = a_0 + a_1 x + \ldots + a_m x^m + O(x^{m+1}) \quad , |x| \leq r$$

Para $g(x) = a_0 + a_1 x + \ldots + a_m x^m + x^{m+1}(a_{m+1} + a_{m+2}x + \ldots)$ debemos sólo mostrar que la cantidad que se halla entre paréntesis está limitada cuando $|x| \leq r$, y que fácilmente se ve que

$|a_{m+1}| + |a_{m+2}| r + \ldots$ es una cota superior.

Por ejemplo, considere las funciones generatrices siguientes, las cuales tienen importantes relaciones:

$$e^x = 1 + x + (1/2!)x^2 + \ldots + (1/m!)x^m + O(x^{m+1}),$$
donde $|x| \leq r$, cualquier r fija.

$$\ln(1 + x) = x - (1/2)x^2 + \ldots + (-1)^{m+1}/m \, x^m + O(x^{m+1}),$$
donde $|x| \leq r$, cualquier $r < 1$ fija.

$$(1 + x)^\alpha = 1 + \alpha x + \binom{\alpha}{2}x^2 + \ldots + \binom{\alpha}{m}x^m + O(x^{m+1}),$$
donde $|x| \leq r$, cualquier $r < 1$ fija.

La afirmación que r es fija, significa que r debe tener valores definidos cuando la notación O-Grande es usada. Obviamente se tiene que $e^x = O(1)$ cuando $|x| \leq r$, puesto que $|e^x| \leq e^r$, pero la constante M implicada por la notación O-Grande depende sobre n. De hecho se puede ver fácilmente que si x está sobre el rango de $-\infty < x < \infty$ entonces $e^x \neq O(x^m)$ para cualquier m.

Por ejemplo, considere la cantidad $\sqrt[n]{n}$ cuando n es muy grande, la operación n-ésima raíz tiende a decrementar el valor, pero no es inmediatamente obvio cuándo $\sqrt[n]{n}$ decrementa o incrementa. Se tiene que $\sqrt[n]{n}$ decrementa a la unidad. Considere

ahora una cantidad más complicada, como $n(\sqrt[n]{n} - 1)$. Se tiene que $(\sqrt[n]{n} - 1)$ obtiene valores muy pequeños cuando n se va haciendo muy grande; ¿qué pasa con $n(\sqrt[n]{n} - 1)$?

Este problema se puede resolver fácilmente aplicando las fórmulas anteriores. Se tiene

$$\sqrt[n]{n} = e^{\frac{1}{n}\ln n} = 1 + (\ln n/n) + O((\ln n/n)^2)$$

Esta ecuación prueba de que $\sqrt[n]{n} \to 1$.
Por consiguiente, se dice que

$$\begin{aligned} n(\sqrt[n]{n} - 1) &= n(\ln n/n + O((\ln n/n)^2)) \\ &= \ln n + O((\ln n)^2/n) \end{aligned}$$

así encontramos que $n(\sqrt[n]{n} - 1)$ es aproximadamente igual a $\ln n$; la diferencia es $O((\ln n)^2/n)$, la cual se aproxima a cero cuando n tiende a infinito.

Por otra parte, nuestro interés en la eficiencia importa sobre todo en la resolución de problemas de gran tamaño. Si el arreglo (array) que hay que ordenar contiene sólo 10 elementos, no importa si la ordenación es eficiente o no, puesto que el problema es pequeño y el número de comparaciones es razonablemente pequeño. Pero conforme crece el tamaño del arreglo, el número de comparaciones crece aún más rápido.

Así pues, una aproximación de la relación entre el tamaño del problema y la cantidad de trabajo necesitado para hacerlo es la O-Grande, también llamada orden de magnitud. La O-Grande de una función es el orden del término de la función que crece más rápido con respecto a n (el tamaño del problema).

Por ejemplo, si

$$f(n) = n^4 + 100\,n^2 + 10\,n + 50$$

entonces f(n) es del orden n^4 (en notación O-Grande, $O(n^4)$). Es decir, para grandes valores de n, n^4 dominará la función.

Un tiempo de cálculo constante se escribe O(1). Puesto que el objetivo es minimizar el trabajo, O(1) es mejor que O(n), tiempo lineal, el cual a su vez será mejor que el tiempo cuadrático $O(n^2)$, el tiempo cúbico $O(n^3)$ que es peor, y el tiempo exponencial $O(2^N)$ que es malísimo. Otro tiempo de cálculo que se ve frecuentemente es $O(\log_2 n)$, el cual es mejor que O(n).

La siguiente figura 1.2 muestra el tiempo de ejecución de una computadora que se tardaría si tuviera los siguientes datos, donde N es el número de datos, y la expresión que está en el renglón principal es la complejidad del algoritmo. Dichos tiempos están en unidad de nanosegundos.

N	LOGz N	NLOGz N	N^2	N^3	2^N
1	0	1	1	1	2
2	1	2	4	8	4
4	2	8	16	64	16
8	3	24	64	512	256
16	4	64	256	4096	65536
32	5	160	1024	32768	2147483648
64	6	384	4096	262144	Alrededor de 5 años con las instrucciones de una supercomputadora.
128	7	896	16384	2097152	Alrededor de 600,000 veces mayor que la edad del Universo en nanosegundos (para una estimación de 6 billones de años).
256	8	2048	65536	16777216	NO PREGUNTAR.

Figura 1.2

Se hace mención que el número de comparaciones que puede realizar una computadora en un segundo es de aproximadamente de un millón .

Como puede observarse en la figura 1.2, algunos de los tiempos de cálculo crecen dramáticamente con respecto al tamaño de N. En particular, observe que N y $\log_2 N$ crece mucho más lentamente que N^2. Es interesante percatarse que los valores de la última columna crecen tan rápidamente que el tiempo de cálculo necesitado por problemas de este orden, puede exceder la duración de vida estimada del Universo.

ESTUDIO DE UN ALGORITMO.

Sea M un algoritmo típico. Se desea aplicar algunas técnicas para estudiar este algoritmo.

ALGORITMO M. (Encontrar el Valor Máximo).Dados n elementos X[1] , X[2] , . . . , X [n] , se desea encontrar m y j tales que

$$m = X[j] = \max_{1 \leq k \leq n} \{X[k]\}$$

para el cual j es tan grande como sea posible.

M1. <u>Inicializar</u> . Sea j:= n.
k:= n - 1.
m:= X[n] .

M2. <u>Todos Probados</u> . Si k = 0, el algoritmo termina.

M3. <u>Comparar</u> . Si X[k] ≤ m ir a M5.

M4. <u>Cambiar m</u> . Sea j:= k
m:= X[k]
Ahora m es el valor máximo.

M5. <u>Decrementar k</u> . Decrementar k por una unidad, es decir,
k:= k - 1
Después ir a M2.

El análisis de algoritmos es importante en la programación de computadoras, porque existen varios algoritmos disponibles para una aplicación particular, y se debe saber cuál es el mejor.

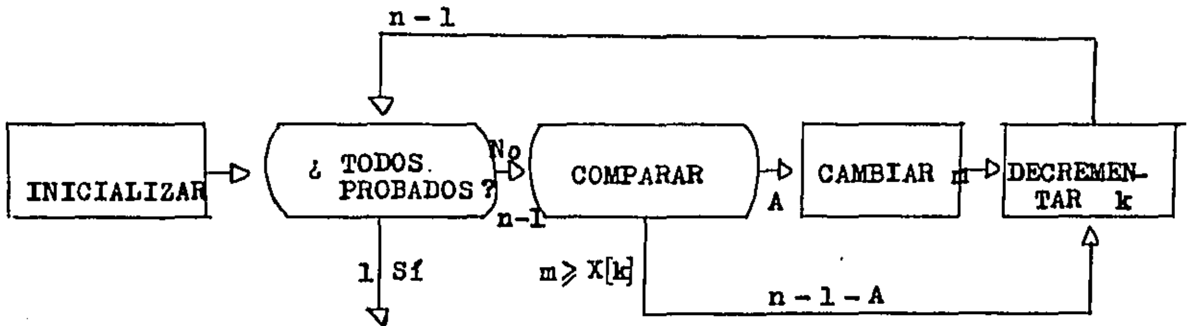


Figura 1.3 Algoritmo M.

De la figura 1.3 del Algoritmo M, las etiquetas sobre las flechas indican el número de veces que cada trayectoria es tomada. Nótese que la 1a. ley de Kirchhoff debe satisfacerse, es decir, la cantidad de flujo que entra en cada nodo debe ser igual a la cantidad de flujo que sale.

El algoritmo M requiere una cantidad fija de memoria, así se debe analizar sólo el tiempo requerido para su ejecución. Para hacer esto se debe contar el número de veces que cada paso es ejecutado. Por ejemplo, de la figura 1.3 se tiene

Paso Número	Número de Veces
M1	1
M2	n
M3	n - 1
M4	A
M5	n - 1

TABLA 1

Conociendo el número de veces en que cada paso es llevado a cabo, nos da la información necesaria para determinar el tiempo de ejecución sobre una computadora en particular.

En la tabla 1, se conocen todas las cosas, excepto la cantidad de A, la cual es el número de veces que se debe cambiar el valor del máximo. Para completar el análisis, se debe estudiar esta interesante cantidad A.

El análisis usualmente consiste en encontrar el valor mínimo de A (para gente optimista), el valor máximo de A (para gente pesimista), el valor promedio de A (para gente probabilística), y la desviación estándar de A (una cantidad indicativa de saber cómo cierra el promedio que se puede esperar del valor de A).

Si el valor mínimo de A es cero, sucede que

$$X[n] = \max_{1 \le k \le n} X[k]$$

Si el valor Máximo de A es n -1, sucede que

$$X[1] > X[2] > \ldots > X[n]$$

Por consiguiente, el valor promedio oscila entre 0 y $n-1$. ¿Será $1/2\ n$? ¿ $1/3\ n$?

Para responder estas preguntas se necesita definir qué significa el promedio (valor medio); y para definir propiamente el promedio, se deben hacer algunas suposiciones acerca de las características esperadas de la entrada de datos X[1], X[2],...X[n] Se debe suponer que las X[k] tienen valores distintos, y que cada una de las n! permutaciones de estos valores es igualmente posible.

En la ejecución del algoritmo M de los valores que tienen las X[k] sólo el orden relativo interviene.

Por ejemplo, supongamos que n = 3. Decimos que cada una de las siguientes 6 posibilidades es igualmente probable.

Situación	Valor de A
X[1] < X[2] < X[3]	0
X[1] < X[3] < X[2]	1
X[2] < X[1] < X[3]	0
X[2] < X[3] < X[1]	1
X[3] < X[1] < X[2]	1
X[3] < X[2] < X[1]	2

TABLA 2

Por lo tanto el valor promedio de A cuando n = 3 es

$$(0 + 1 + 0 + 1 + 1 + 2)/6 = 5/6$$

Es claro que se toma X[1], X[2],..., X[n] por los números 1,2,...n en algún orden, bajo la suposición de que cada una de las n! permutaciones es igualmente posible.

La probabilidad de que A tenga el valor de k es

$$\rho_{nk} = \frac{(\text{número de permutaciones de n objetos para el cual } A = k)}{n\,!}$$

Por ejemplo, en la tabla 2,

$$\rho_{30} = 1/3$$
$$\rho_{31} = 1/2$$
$$\rho_{32} = 1/6$$

El valor promedio (media) es definido como

$$A_n = \sum_k k\rho_{nk}$$

La varianza V_n es definida como el valor promedio de $(k - A_n)^2$ es decir,

$$V_n = \sum_k (k - A_n)^2\rho_{nk} = \sum_k k^2\rho_{nk} - 2\,A_n\sum_k k\rho_{nk} + A_n^2\sum_k \rho_{nk}$$
$$= \sum_k k^2\rho_{nk} - 2\,A_n\,A_n + A_n^2$$
$$= \sum_k k^2\rho_{nk} - A_n^2$$

Finalmente, la desviación estándar σ_n es definida como $\sigma_n = \sqrt{V_n}$

ALGORITMOS DE ORDENACION. EFICIENCIA.

Se ha dicho que el algoritmo de ordenación rápida es un buen algoritmo de ordenación. ¿ Qué se quiere decir con "bueno" ? ¿Cómo se pueden comparar dos algoritmos que hacen la misma tarea ? Para hacer tal comparación, se debe definir primero un conjunto de medidas objetivas que puedan aplicarse a cada algoritmo. El análisis de algoritmos es un área importante de la informática teórica; aquí se examinará una parte pequeña de este tema, suficiente como para determinar cuál de dos algoritmos requiere menos trabajo para realizar una tarea en particular.

¿ Cómo medimos el trabajo que realizan dos algoritmos ? La primera solución es obvia, y es codificar los algoritmos y luego comparar los tiempos de ejecución, ejecutando los dos programas. El que tenga menor tiempo de ejecución será evidentemente el mejor algoritmo. ¿ O no lo es ? Sólo podemos decir realmente que el programa A es más eficiente que el programa B "en tal computadora". Los tiempos de ejecución son específicos de cada computadora. Por supuesto podríamos probar los algoritmos sobre todas las computadoras posibles, pero deseamos una medida más general.

Una segunda posibilidad es contar el número de instrucciones o reglas ejecutadas. Sin embargo, esta medida varía con el lenguaje de programación usado, así como el estilo de cada programador. Para estandarizar de alguna forma esta medida, podríamos contar el número de pasos de un bucle crítico del algoritmo. Si cada iteración conlleva una cantidad constante de trabajo, esta medida nos dará un criterio con sentido de la eficiencia.

Estos pensamientos nos conducen a la idea de aislar una operación particular fundamental del algoritmo y contar el número de veces que se ejecuta esta operación. Por ejemplo, supongamos que estamos buscando un cierto valor en un arreglo. Podemos contar cuántas comparaciones se hacen entre el valor buscado y los elementos del arreglo hasta localizar el valor. Si estábamos sumando los elementos de un arreglo de enteros, podríamos contar las operaciones de suma de enteros que se necesitan. (Observe que esta cuenta es una función del número de elementos del arreglo. Para un arreglo de n elementos, habrá n - 1 operaciones de suma, por lo tanto podemos comparar los algoritmos para el caso general, no para un arreglo de tamaño específico). Si deseamos comparar algoritmos para multiplicar dos matrices de reales, podemos sugerir una medida que combine las operaciones de suma y multiplicación necesitadas en la multiplicación de matrices.

Este último ejemplo nos lleva a una consideración muy interesante: algunas veces una operación dominará el algoritmo de tal forma, que las otras operaciones se desvanecerán como "ruido de fondo". Por ejemplo, si queremos comprar elefantes y peces de colores, sólo necesitaremos realmente comparar los precios de los elefantes; el costo del pez de color es trivial en la comparación. Igualmente, la multiplicación de reales es mucho más cara que la suma en términos de tiempo de computadora, por lo que la operación de suma es un factor trivial en la eficiencia del algoritmo de multiplicación de matrices; podemos perfectamente contar sólo las operaciones de multiplicación, ignorando la suma. En el análisis de algoritmos, encontraremos frecuentemente que una operación domina al algoritmo, relegando efectivamente a las otras al nivel del ruido.

ANALISIS DE LA ORDENACION RAPIDA.

Ahora que se tiene una medida (número de comparaciones) y una forma de decir "mucho más eficiente", se analiza el algoritmo denominado de ordenación rápida.

En la primera llamada, cada elemento del arreglo se compara con el valor de partición, por lo que el trabajo hecho es O(n). El arreglo se divide en dos partes, las cuales se examinan a continuación.

Cada uno de estos segmentos es luego dividido, quedando cuatro partes. Este análisis se ilustra en la figura 1.4. En cada nivel, el número de partes se duplica. ¿ En qué nivel hemos acabado de dividir los elementos ? Si la partición de cada segmento es cada vez, aproximadamente, en la mitad, tendremos log n particiones. En cada partición, hacemos O(n) comparaciones.

Por tanto, en el caso medio la ordenación rápida es $O(n \log_2 n)$, lo cual es más rápido que $O(n^2)$. ¿ Cuándo la ordenación rápida no es rápida ? Considerar una rutina de ordenación rápida cuyo algoritmo de partición utiliza el primer elemento del arreglo (o del segmento del arreglo bajo consideración) como valor para la partición. ¿ Qué sucedería si el arreglo está ya ordenado ? Las particiones serían muy desproporcionadas y las siguientes llamadas al algoritmo ordenarían en un segmento con sólo un elemento y un segmento que contiene todo el resto del arreglo (o segmento del arreglo). Claramente esta situación produciría una ordenación que no es del todo rápida. De hecho, en este caso la ordenación rápida sería $O(n^2)$. La posibilidad de que se presente esta situación es muy pequeña. Por analogía, considere lo raro que sería barajar una

baraja de cartas y que quedaran ordenadas. Por otra parte, en algunas aplicaciones se puede conocer que el arreglo original está probablemente ordenado o casi ordenado. En tales casos, podría usarse una ordenación diferente o un algoritmo de partición diferente para la ordenación rápida.

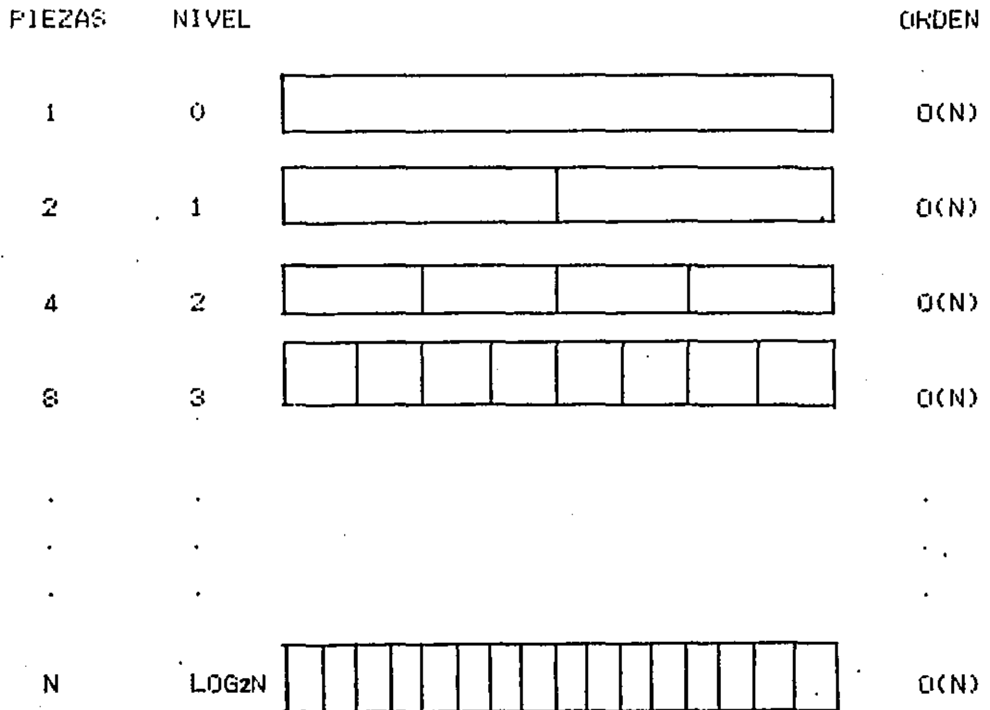


Figura 1.4 Análisis de la ordenación rápida.

CAPITULO II. ESTRUCTURAS DE DATOS BÁSICAS.

1. INTRODUCCIÓN.
2. LA NECESIDAD PARA ESTRUCTURAR EN CONJUNTOS DE DATOS.
3. ESTRUCTURAS DE DATOS.
4. ESTRUCTURAS DE DATOS ESTÁTICAS Y DINÁMICAS.
5. ESTRUCTURA MODULAR DE PROGRAMAS.
6. RECURSIVIDAD.

INTRODUCCION

Las máquinas más antiguas funcionaban como el mecanismo de un reloj, sin admitir variación después de iniciado el movimiento; pero las modernas, poseen órganos sensoriales, es decir, mecanismos de recepción de información que provienen del exterior, tales como los proyectiles teledirigidos, el mecanismo de apertura automática de las puertas, etc. Claro está, la información no se toma en *bruto*, sino que pasa a través de los mecanismos especiales de transformación que posee el aparato. Por consiguiente, los datos adquieren una nueva forma utilizable en las etapas ulteriores de la actividad.

La riqueza de la información que se tiene que manejar en la actualidad da a entender un incremento en el sistema computacional cuyo manejo automático de esta información, deberá asignársele una estructura que facilite al sistema recibir y guardar información, y poder restablecerla para propósitos específicos. En cualquier proceso de gestión de base de datos eficiente, la planificación es un punto vital. El proceso de diseño de una base de datos requiere que se piense primero en la forma en que los usuarios van a preguntar sobre los datos. Así, datos y atributos son los términos importantes en el diseño de programas. Los datos están constituidos por la información que llega al programa. Los atributos son los tipos de datos que conforman el programa.

En este respecto, la necesidad para estructurar la información requiere relaciones intrínsecas del sistema. Aquí, en este capítulo, se estudian estas necesidades y relaciones, y sus diferentes formas de concebirse.

LA NECESIDAD PARA ESTRUCTURAR EN CONJUNTOS DE DATOS.

La abundancia de la información que se tiene que manejar en la actualidad conlleva a la creación de un sistema computacional existente con la realidad representada por dicha información. Así, para el manejo automático de esta información, deberá asignarse una estructura que facilite al sistema recibir y guardar información, y por consiguiente restablecerla para propósitos de procesamiento especificados.

Con este respecto, la necesidad para estructurar en el conjunto de información a ser procesada por un sistema computacional resulta de los siguientes requerimientos:

1. La estructura es necesaria para caracterizar la tecnología dinámica de la actualidad, y así lograr soluciones automáticas de los problemas:

2. La estructura es útil ordenadamente porque permite analizar de manera adecuada la tecnología actual en términos de abstracciones (relaciones entre las componentes del sistema) que faciliten la posibilidad de tomar decisiones acerca de las propiedades esenciales, y por consiguiente, permitir la separación de los aspectos fundamentales de aquéllos que no lo son.

3. La estructura es, por consiguiente, necesaria para facilitar la organización de las características primordiales de la realidad, para que de esta manera exista el desarrollo de nuevas máquinas.

ESTRUCTURAS DE DATOS.

Actualmente las computadoras juegan un papel importante en la comodidad y rapidez para la realización de cálculos complicados y onerosos de tiempo, siendo la característica principal la gran capacidad de almacenamiento y acceso a grandes masas de información. En la mayoría de las aplicaciones, la gran masa de información que es necesario procesar representa una abstracción de la realidad. Es por ello que la información utilizada por la computadora consiste de una selección de datos de la realidad, esto es, un conjunto de datos que es relevante para el problema bajo estudio.

Al resolver cualquier problema, es necesario elegir una abstracción de la realidad guiada por el problema a resolver. Después debe seleccionarse la forma de representar esta información en base a las posibilidades concretas que ofrece la computadora. Generalmente estas dos etapas de diseño no son totalmente independientes una de otra.

Pero ¿ cómo elegir entre dos o más posibles estructuras de datos ? La elección de la representación de los datos es a menudo bastante difícil y no está determinada exclusivamente por los instrumentos disponibles, ya que al igual que en la elección de algoritmos distintos, debe considerarse primero los requerimientos del programa (por ejemplo, la necesidad de ahorrar memoria o la posibilidad de acceder fácilmente a un registro particular). Si no existen diferencias entre las estructuras, entonces debe considerarse la elegancia y claridad relativa de dichas estructuras.

El objetivo de la estructura de datos es el de poder manipular los datos de los programas en función de su representación lógica en lugar de su alamacenamiento físico. En un grupo limitado, los lenguajes de programación suministran estructuras de datos incorporados que enmascaran la colocación física de los datos en memoria.

Un lenguaje de programación es representada por una computadora abstracta capaz de entender los términos utilizados en este lenguaje, que pueden ser más abstractos que los de los objetos utilizados por la máquina real.

La importancia de utilizar un lenguaje que ofrezca un conjunto conveniente de operaciones fundamentales para resolver la mayoría de los problemas que se presentan en el procesamiento de datos, reside principalmente en la confiabilidad de los programasresultantes. Es más fácil diseñar un programa razonando sobre los conceptos familiares de números, conjuntos, sucesiones y repeticiones, que razonando con bits, "palabras", y saltos de secuencia.

Definición. Ahora bien, se define como estructura da datos a la colección de datos cuya organización se caracteriza por las funciones de acceso que se usan para almacenar y acceder a elementos individuales de datos.

Cada estructura de datos puede analizarse desde tres perspectivas:

1. NIVEL ABSTRACTO O LOGICO. En este nivel, se esquematiza la organización y se especifican los procedimientos y funciones generales de acceso.

2. NIVEL DE IMPLANTACION. Aquí se examinan las formas de representación de los datos de memoria y cómo implantar los procedimientos y funciones de acceso en el lenguaje de programación. Se examinan las distintas formas en que pueden implantarse las estructuras de datos.

3. NIVEL DE APLICACION O USO. En este nivel se presentan ejemplos relativos a los niveles anteriores, y se examinan en detalle algunos casos en los que la estructura de datos representa con precisión las relaciones entre los datos.

ELECCION DE UNA ESTRUCTURA DE DATOS.

A usted le han dado un problema que requiere que obtenga y almacene algunos datos de entrada, procese los datos e imprima o guarde los resultados. ¿ Cómo puede saber qué estructura conceptual es apropiada para los datos ? ¿ Por dónde empezar ? Un lugar por dónde comenzar se sugiere en la pregunta: " ¿ Cómo lo haría a mano ?". La respuesta a esta pregunta puede conducirle a los primeros borradores de la solución del problema y de las estructuras de datos.

La elección de las estructuras de datos es tan importante como la elección de los algoritmos. De hecho, se necesita un conocimiento de los algoritmos a aplicar a los datos para hacer una acertada elección de las estructuras de datos. Inversamente, la elección de buenos algoritmos requiere el conocimiento de las estructuras de datos usadas. Una de las ventajas del método de diseño descendente sobre el diagrama de flujo, es que las estructuras de datos para el programa pueden desarrollarse de una forma análoga y están integradas en el diseño. Los diagramas de flujo tienden a preocuparse más del flujo de control y ponen menos énfasis en los datos que se manejan en el programa.

Las ventajas y desventajas de las diferentes organizaciones de datos e implantaciones deben considerarse con respecto a las operaciones requeridas por el problema. Por ejemplo, una de las desventajas de usar listas enlazadas en vez de listas secuenciales en un arreglo, es la limitación de tener que hacer una búsqueda secuencial; sin embargo, si el programa requiere muchas inserciones por el comienzo de la lista, esta desventaja es irrelevante. Si se decide usar una representación enlazada de una lista, debe decidirse si debe ser lineal o circular, simplemente enlazada o doble. Los enlaces dobles son más caros (hay que poner un campo apuntador extra), pero si el programa requiere muchas supresiones de la lista, el gasto puede merecer la pena.

¿ Deben implantarse las listas enlazadas en un arreglo o en nodos asignados dinámicamente con variables apuntadores (suponiendo que el lenguaje fuente tenga variables apuntadores) ?

Para esta decisión hay que tomar en consideración si se puede predecir o no el número de registros que se necesitan. Si no se puede, un arreglo puede ser peligroso (estimado demasiado pequeño) o derrochador (estimado demasiado grande).

La pregunta " ¿ Qué estructura de datos ?" es difícil de responder; la respuesta depende de los factores que hay que considerar en cada programa específico. A continuación se dan como guías unos cuantos criterios para hacer una elección de estructura de datos:

- La estructura de datos debe reflejar los requerimientos de las operaciones a ejecutar sobre los datos.
- Si uno de los tipos de datos incorporados (es decir, los tipos primitivos, arreglo o registro) es adecuado para solucionar el problema, debe usarse. Si no, la estructura de datos diseñada debe parecerse al dibujo conceptual de los datos. Los tipos primitivos son los números enteros, reales y los caracteres lógicos. Pensar sobre cómo resolvería el problema a mano.
- La implantación debe dejarse a los subprogramas de niveles inferiores, transparente a la parte de aplicación del programa. Esto hace que el código sea más fácilmente legible, así como modificable.
- Si la eficiencia es una consideración importante, la elección de la estructura de datos y de la implantación debe reflejar el origen del factor limitante. La limitación de memoria puede requerir una estructura de datos menos complicada. Por otra parte, rigurosos requerimientos de tiempo pueden hacer que las estructuras de datos sean más complicadas para aumentar la velocidad del algoritmo. Normalmente, los requerimientos de tiempo y espacio no pueden a la vez satisfacerse completamente.

Sin embargo, en muchos casos la eficiencia del programador puede ser el factor limitante, requiriendo que la estructura de datos soporte la simplificación del diseño del programa.

- Finalmente, resistir la tentación de sobrecargar el diseño. Si el programa realmente necesita una lista enlazada lineal simple, no usar una lista doblemente enlazada, con cabeceras y finales para una buena medida. Esto claramente desperdicia espacio, tiempo y esfuerzo. Diseñar siempre las estructuras de datos, así como los algoritmos, de tal forma que reflejen las especificaciones del programa.

ESTRUCTURAS DE DATOS ESTATICAS Y DINAMICAS.

En la declaración de una variable arreglo, la especificación del rango del índice es un tipo, y por lo tanto no puede ser una variable. En otras palabras, el compilador debe saber cuántas posiciones asignar al arreglo. Los arreglos y todas las estructuras de datos almacenadas en memoria principal, son variables estáticas; su tamaño se fija en tiempo de compilación.

Una variable estática existe mientras se esté ejecutando la parte del programa en la que está declarada (bloque).

El Pascal también tiene un mecanismo para crear variables dinámicas. Esto significa que se puede definir un tipo en tiempo de compilación, pero realmente no crear una variable de ese tipo hasta el tiempo de ejecución. Estas variables dinámicas pueden crearse o destruirse en cualquier momento durante la ejecución del programa. Pueden definirse de cualquier tipo simple o estructurado. Referenciamos a una variable dinámica no por su nombre sino mediante un apuntador. Un apuntador es una variable que contiene la dirección (posición) de memoria de la variable dinámica a la que referencia. Cada nueva variable dinámica creada tiene un apuntador asociado para referenciarla (seleccionarla).

UTILIDAD DE LA VARIABLE DINAMICA.

Las variables estáticas tales como arreglos y registros para estructurar nuevos datos funcionan muy bien para muchas aplicaciones, pero presentan algunas desventajas. Un registro puede contener componentes de diferentes tipos de datos y frecuentemente es una forma lógica de describir nuestros datos. Sin embargo, normalmente necesitamos una colección (o lista) de

estos registros, para lo que usábamos un arreglo de registros.

Se puede mantener una lista de datos en un archivo o, cuando se trabaja con la lista en memoria principal, en un arreglo. Se puede pensar en una lista como un arreglo uni-dimensional, independientemente de lo complejas que sean las componentes del arreglo.

La forma normal es declarar un arreglo lo bastante grande como para almacenar la cantidad máxima de datos que lógicamente cabe esperar. Puesto que normalmente se tienen menos datos que ese máximo, se anota la longitud del sub-arreglo en el que se tienen almacenados los valores, y así sólo se accede a esa parte del arreglo con valores. Este sub-arreglo puede variar en longitud durante la ejecución desde cero componentes al máximo del arreglo.

¿ Qué se puede hacer si no se sabe cuántas componentes se necesitan ? Se puede siempre dar una longitud y prever en el programa que pueda ocurrir un error al tratar de almacenar datos cuando el arreglo está lleno.

¿ Qué hacer si se desea que la lista de componentes esté almacenada de una forma ordenada ? ¿ Cómo insertar o suprimir una componente ? La inserción o supresión de componentes de una lista supone el desplazar parte de la lista en una u otra dirección. Si la lista es muy larga, la inserción o supresión de componentes necesitará una cantidad significativa de tiempo.

Se puede utilizar variables dinámicas para evitar estos problemas de las variables estáticas tales como el arreglo. Se

pueden crear nuevas componentes para nuestra lista sólo cuando se necesiten, usando variables dinámicas como componentes. Haciendo que cada componente contenga el enlace o apuntador a la siguiente componente de la lista, se puede crear una estructura de datos dinámica que se expande o contrae conforme se ejecuta el programa.

Veáse la figura 2.1 para visualizar dicha estructura dinámica.

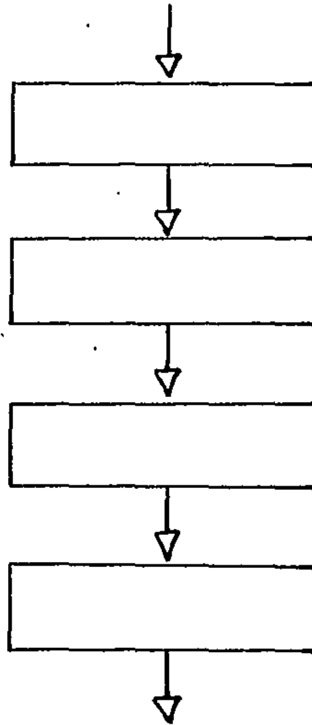


Figura 2.1 Estructura de Datos Dinámica.

No se tiene que saber de entrada qué tan grande será la lista. Ahora la única limitación es la cantidad de memoria disponible. Se puede cambiar fácilmente el orden de las componentes cambiando sólo los valores de los apuntadores.(ver figura 2.2). La inserción y supresión de componentes son más fáciles y más rápidas; simplemente hay que cambiar uno o dos valores de los apuntadores (ver figura 2.2).

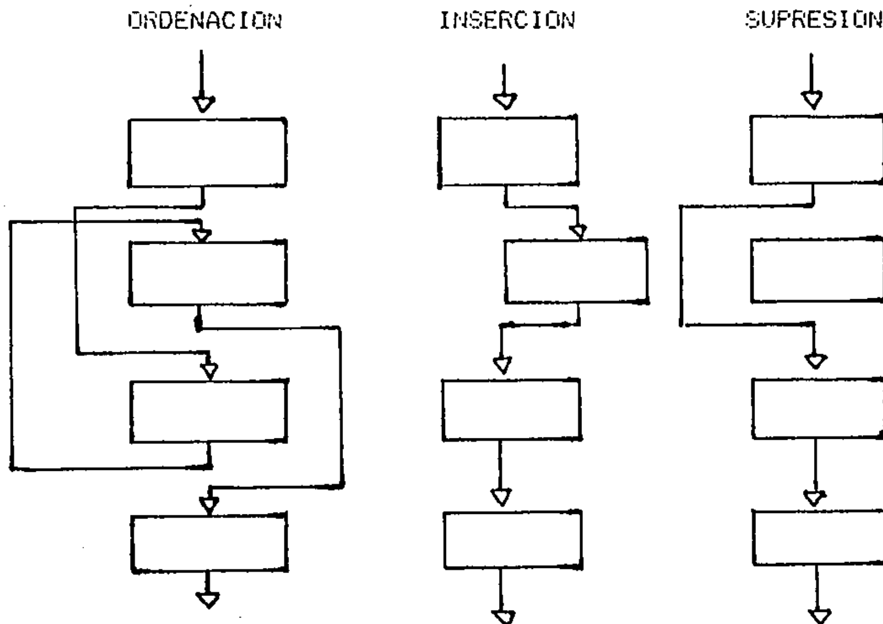


Figura 2.2. Manipulación de una estructura de datos dinámica.

Se pueden usar variables dinámicas y apuntadores para crear listas y estructuras de datos más complejas que puedan expandirse o contraerse durante la ejecución del programa. Estas estructuras de datos son extremadamente flexibles, permitiendo una más fácil inserción y supresión de las componentes. Algunas de estas estructuras de datos son: listas enlazadas, pilas, colas, árboles binarios, etc.

LA ESTRUCTURA ARREGLO (ARRAY)

El arreglo es, probablemente, la estructura de datos más conocida, debido a que en muchos lenguajes es la única estructura explícitamente disponible. Un arreglo es una estructura homogénea, formado por una colección finita de elementos ordenados, todos del mismo tipo. El acceso se realiza mediante un índicé que permite especificar cuál es el elemento deseado dando su posición en la colección.

La sintaxis y semántica del 'lenguaje especifica la función de acceso. En Pascal se declara un tipo de datos que define cómo debe ser un arreglo uni-dimensional.

El arreglo se denomina también estructura de acceso aleatorio, es decir, todas sus componentes pueden seleccionarse arbitrariamente y son igualmente accesibles. Para designar una componente aislada, el nombre de la estructura total se amplía con el denominado índice de selección de la componente. El índice debe ser un valor del tipo definido como el tipo índice del arreglo. Por tanto, la definición de un tipo arreglo T especifica tanto un tipo base To como un tipo índice I.

```
type T = array [ I ] of To
```

Ejemplos:

```
type  Fila     = array [1..5]  of real
type  Tarjeta  = array [1..80] of char
type  Alfa     = array [1..10] of integer
type  Nota     = array [1..50] of boolean
```

ARREGLOS BI-DIMENSIONALES.

Además de los arreglos uni-dimensionales, la mayoría de los lenguajes de programación suministran arreglos bi-dimensionales. Los arreglos bi-dimensionales pueden definirse de dos formas:

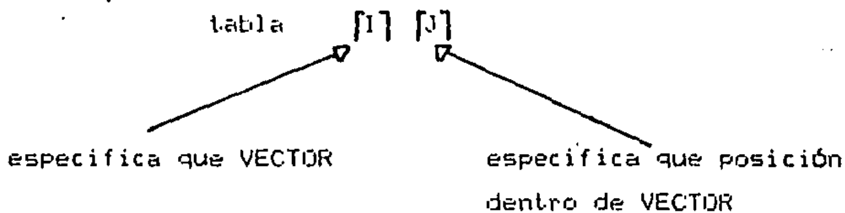
1. Un arreglo bi-dimensional es un arreglo uni-dimensional en el que el tipo de los datos de cada componente es a su vez un arreglo uni-dimensional.
2. Un arreglo bi-dimensional es un tipo de datos estructurado formado por una colección finita de elementos, todos del mismo tipo de dato. Cada elemento está ordenado en dos dimensiones. El acceso se hace usando un par de índices que permiten especificar a qué elemento de la colección se desea acceder. El primer índice referencia a la posición del elemento en la primera dimensión; el segundo índice se refiere a la posición del elemento en la segunda dimensión.

De nuevo, la sintaxis y semántica del lenguaje de programación que se use especifica la función de acceso. En Pascal, se puede declarar un arreglo bi-dimensional de acuerdo con cualquiera de las anteriores definiciones.

Por ejemplo:

```
type vector     = array [1..3] of integer;
     tipotabla  = array [1..5] of vector;
var  tabla      : tipotabla;
```

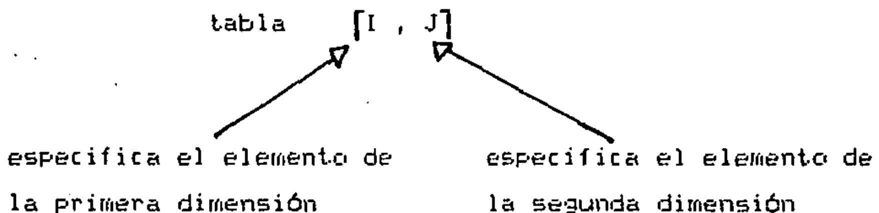
La función de acceso es como sigue:



La segunda forma de definir el arreglo bi-dimensional TABLA es:

```
type tipotabla  =  array [1..5,1..3]  of integer;
var  tabla      :  tipotabla;
```

La función de acceso para la segunda definición es como sigue:



Aunque conceptualmente estas estructuras son muy diferentes, las estructuras de datos definidas por estos dos métodos son idénticas.

Un arreglo bi-dimensional es la estructura de datos ideal para representar datos que estén estructurados lógicamente como una tabla con filas y columnas. La primera dimensión representa las filas y la segunda dimensión representa las columnas. Cada celda del arreglo puede contener un valor de algún tipo escalar. Cada dimensión representa una relación.

LA ESTRUCTURA REGISTRO.

El Pascal tiene un tipo de datos incorporado adicional, que es el registro. Esta es una estructura muy útil que no está disponible en todos los lenguajes de programación, por ejemplo en Fortran. Sin embargo, el lenguaje de programación Cobol, utiliza extensivamente los registros.

Un registro es un tipo de datos estructurado formado por una colección finita de elementos no necesariamente homogéneos, llamados campos. El acceso se realiza mediante un conjunto de selectores de campos.

Un ejemplo, en proceso de datos, es la forma de describir a las personas mediante unas pocas características relevantes, como su nombre y apellidos, así como su fecha de nacimiento, sexo y estado civil.

En general, el tipo registro T se define de la manera siguiente:

```
type   T   =   record   S1  :  T1  ;
                        S2  :  T2  ;
                           . . .
                        Sn  :  Tn
               end
```

Ejemplos:

```
1.  Type complejo  =  record   re :  real;
                                im :  real
                      end;
2.  Type fecha     =  record  dia :  1..31;
                              mes :  1..12;
                              año :  1..2000
                      end;

3.  type persona   =  record  apellido    :  alfa;
                              nombre      :  alfa;
                              nacimiento  :  fecha;
                              sexo        :  (varon,hembra);
                              ecivil      :  (sol,cas,viu,div)
                      end;
```

LA ESTRUCTURA CONJUNTO (SET)

Los conjuntos son un tipo de datos estructurados, únicos en el Pascal entre los lenguajes de programación comúnmente usados. En matemáticas, un conjunto es una colección, grupo o clase de objetos. Los conjuntos en Pascal son lo mismo, con la restricción de que los objetos de un conjunto deben ser del mismo tipo. Asimismo, en matemáticas un conjunto puede tener cualquier tamaño; sin embargo, en Pascal el tamaño de un conjunto va a estar limitado por la capacidad de la computadora.

Definición. Un conjunto es un tipo de datos estructurado compuesto por una colección de elementos distintos escogidos entre los valores del tipo base, y se declara de la siguiente forma:

```
type T = set of To
```

Ejemplos:

```
1.  type  entconj      =   set of 0..30;
2.  type  carconj      =   set of char;
3.  type  estadocinta  =   set of excepción;
```

Por otra parte, los operadores elementales que se definen en las estructuras de tipo conjunto son:

*	intersección de conjuntos
+	unión de conjuntos
-	diferencia de conjuntos
in	pertenencia a un conjunto

APUNTADORES.

Un apuntador es un tipo de datos simples, constando de un conjunto ilimitado de valores, que direcciona o indica la posición de una variable de un determinado tipo. Un apuntador es un tipo simple, como el entero, el real y el booleano, sin embargo el apuntador no es un identificador estándar.

La declaración de los apuntadores es como sigue:

```
type      enlace   =  ↑ objeto;
```

y se lee así:
" el tipo enlace es un apuntador que señala hacia objeto", donde la flecha (↑) es la que nos dice que enlace es un tipo apuntador.

Una variable del tipo objeto puede ser asociada con un apuntador del tipo enlace. Algunas veces tenemos un apuntador para el cual no existe variable del tipo objeto para asociar, y en este caso se escribe

```
l:= nil
```

El símbolo nil es una palabra reservada en Pascal. Es importante entender la diferencia entre apuntadores y las cosas a las cuáles estos apuntan.

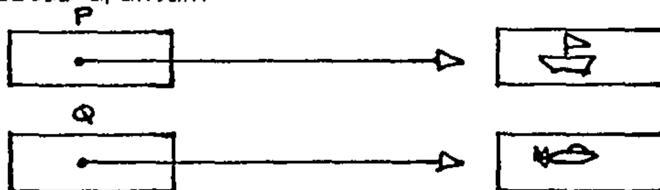


Figura 2.3

La figura 2.3 muestra dos variables P y Q, de tipo enlace, apuntando a diferentes objetos.

La asignación p:= q tiene el efecto de asignar el valor del apuntador q al apuntador p. Tal situación se muestra en la figura 2.4

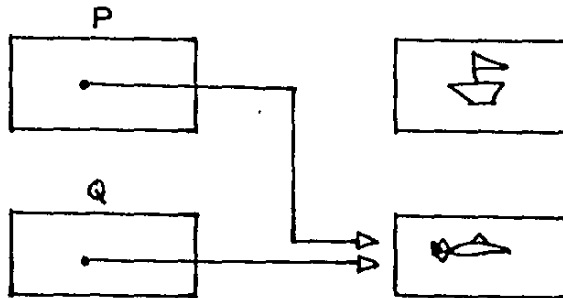


Figura 2.4

Ambos apuntadores apuntan hacia el plano, y el barco se ha perdido (a menos que existiera otro apuntador a él, dos apuntadores pueden apuntar a la misma cosa). Ahora bien, la proposición p↑ := q↑ tiene un efecto bastante diferente. Se está ahora copiando el valor del objeto q↑ al objeto p↑ , cuyo resultado se muestra en la figura 2.5

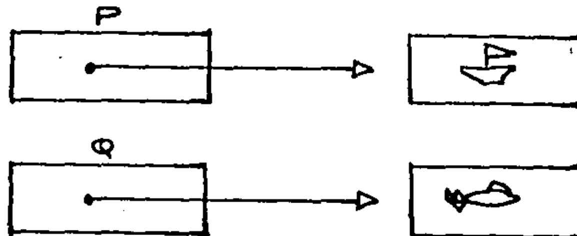


Figura 2.5

SINTAXIS DE LA FLECHA HACIA ARRIBA.

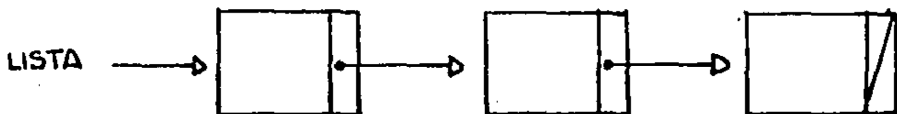
El símbolo ↑ , seguido de un identificador de tipo, define un apuntador, aunque también se usa de otra forma.

NOMBRE DE VARIABLE APUNTADOR ↑ . Denota la variable a la que la variable puntero está apuntando. Por ejemplo, la diferencia entre LISTA y LISTA ↑ es que la primera es una variable puntero (una dirección), mientras que la segunda (LISTA ↑) es el dato real al que está apuntando (un registro).

NOMBRE DE VARIABLE APUNTADOR ↑ . CAMPO Denota el contenido del campo nodo al que apunta la variable puntero. Por ejemplo,

WRITE (LISTA ↑ .INFO) imprime el contenido del campo INFO del nodo apuntado por lista. Nótese la diferencia entre LISTA y LISTA ↑ .INFO .LISTA es un registro completo, mientras que LISTA ↑ .INFO es el dato de un campo del registro.

Se pueden hacer algunas expresiones con punteros complicadas. Por ejemplo, para imprimir el valor del campo INFO del tercer nodo de la lista enlazada



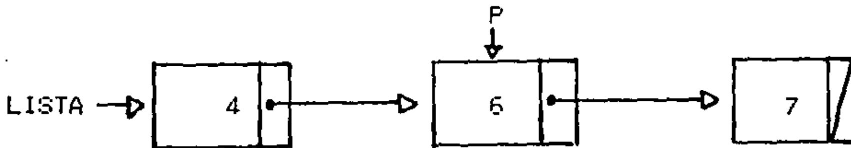
se puede escribir WRITE(LISTA ↑ .NEXT ↑ .NEXT ↑ .INFO)

Por otro lado, para crear un nuevo nodo, se utiliza un procedimiento, incorporado en Pascal, llamado NEW. Este procedimiento tiene un parámetro: el nombre del puntero al nuevo nodo. Por ejemplo, para crear un nuevo nodo, apuntado por P, se puede escribir NEW(P).

Asimismo, existe un procedimiento para eliminar un nodo. Este procedimiento, llamado DISPOSE, tiene un parámetro (un puntero al nodo que se quiere quitar).

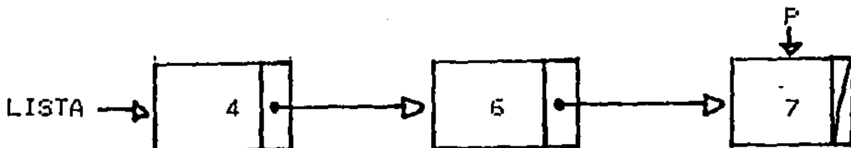
MANIPULACION DE VARIABLES PUNTEROS

Dado el diagrama de una lista enlazada, donde P y LISTA son apuntadores, se pueden manipular para hacer varias cosas:



i) Hacer que P apunte al siguiente nodo, con la instrucción

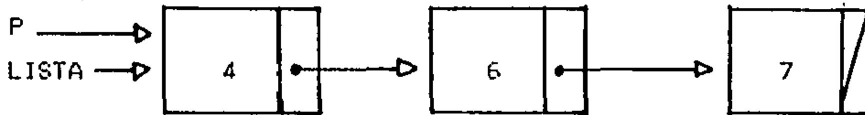
P:= P↑ .SIG



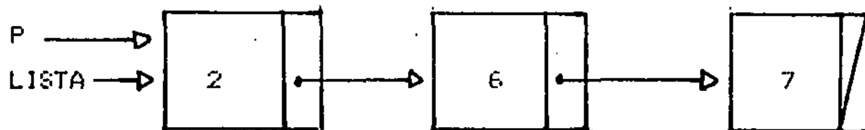
ii) Hacer que P apunte al comienzo de la lista, con la instrucción

P:= LISTA

donde P y LISTA son ambos del mismo tipo.



iii) Poner el valor del campo INFO del primer nodo de la lista a 2, con la instrucción LISTA ↑ .INFO := 2



iv) Poner otro puntero, Q, apuntando al segundo nodo de la lista, con la instrucción Q := LISTA ↑ .SIG

EJEMPLOS.

1) Mostrar lo que escriben los siguientes segmentos de códigos:

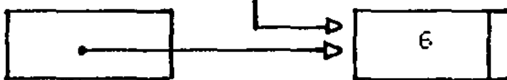
```
NEW(P);
NEW(Q);
P ↑ .INFO := 5;
Q ↑ .INFO := 6;
P:= Q;
P ↑ .INFO := 1;
WRITELN (P ↑ .INFO, Q ↑ .INFO);
```

P

Q

P

Q

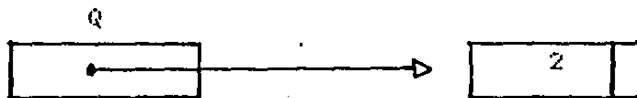


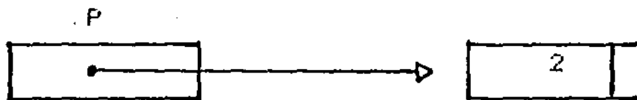
P

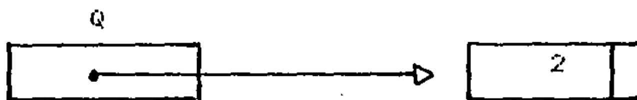
SOLUCION: 1 1

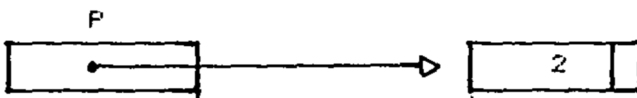
Q

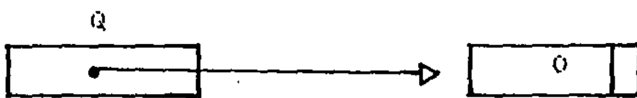
```
2)  NEW(P);     P↑ .INFO :=  3;
    NEW(Q);     Q↑ .INFO :=  2;
    NEW(P);     P↑ .INFO := Q↑ .INFO;
    Q↑ .INFO := 0;
    WRITELN (P↑.INFO, Q↑.INFO);
```











SOLUCION : 2 0

LISTA ENLAZADA.

Una lista de elementos es algo que la mayoría de la gente utiliza frecuentemente en su vida diaria. Se está ya familiarizado con el concepto de almacenar los elementos secuencialmente en una lista, como si se estuviera escribiéndolos uno detrás de otro en un trozo de papel.

A menudo se usa un arreglo para hacer este papel en las soluciones por computadora de problemas que usan listas. Los elementos de la lista se almacenan en posiciones consecutivas del arreglo.

Ahora bien, pensemos en el siguiente caso. Supongamos que una gran compañía va a tener una reunión de accionistas. Habrá 600 asistentes, alojados en 5 hoteles diferentes. Los organizadores de la reunión necesitan tener listas de asistentes por hotel, y se desea escribir un programa con esa finalidad.

Como se observa, se podría usar un arreglo para cada hotel, pero entonces se necesitarían usar 5 arreglos, cada uno capaz de almacenar al número máximo de asistentes, aunque cada accionista puede escoger cualquiera de los 5 hoteles, es posible que todos elijan el mismo.

El uso de un arreglo para cada hotel supone un desperdicio de memoria, porque

5 hoteles x 600 accionistas = 3000 celdas de memoria

Sobrarían 2400 celdas de memoria.

Idealmente, la información sobre cada accionista se debería almacenar de tal forma que sólo se necesitara declarar una estructura y solicitar espacio cuando realmente se vaya a usar; porque de hecho, los inconvenientes de utilizar un sistema de almacenamiento secuencial para representar pilas y colas son:

a) Las cantidades fijas de almacenamiento permanecen asignadas a la pila o a la cola aún cuando la estructura esté realmente utilizando una pequeña cantidad de almacenamiento.

b) No puede asignarse más de la cantidad fijada de almacenamiento, porque se puede presentar la posibilidad de sobreflujo.

Así pues, volviendo al problema de los accionistas, no existe una forma sencilla y práctica de almacenar los elementos en orden secuencial. Si se hallase una forma de conectar los elementos de cada arreglo, se podría almacenarlos todos juntos. Los enlaces entre elementos de una lista particular mantendría unida a la lista sin depender de la contiguidad secuencial en memoria. Podría declararse espacio de memoria para 600 asistentes y las listas de los hoteles estarían enlazadas. Aún si se considerara la memoria extra necesaria para alamcenar los enlaces, se ahorraría mucho espacio.

Se visualiza este problema así:

HOTEL 1	HOTEL 2	HOTEL 3	HOTEL 4	HOTEL 5
1	1	1	1	1
2	2	2	2	2
3	3	3	3	3
.	.	.	.	.
.	.	.	.	.
.	.	.	.	.
600	600	600	600	600

Figura 2.6 Espacio total para 3000 registros, de los cuales se desperdician 2400.

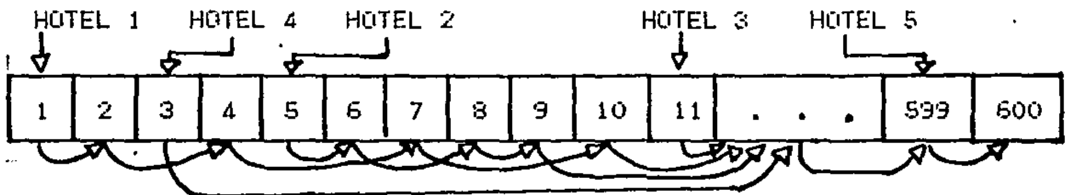


Figura 2.7 Espacio total para 600 registros, no hay desperdicio.

El concepto de enlace entre los elementos de una lista, permite almacenarlos físicamente en cualquier orden conveniente. Los enlaces preservan el orden lógico de los elementos.

Aunque el arreglo como tal es una estructura de acceso directo (se puede localizar cualquier elemento directamente mediante su índice), sólo se puede acceder a los elementos de la lista enlazada según imponga el esquema de apuntadores. Es decir, si la lista enlazada está almacenada en un arreglo llamado DATOS, no se sabe nada acerca de la posición relativa de los elementos de la lista accediendo directamente a posiciones del arreglo. Datos[5] puede ser el primero, quinto o cualquier otro elemento de la lista o puede que no pertenezca a la lista. Sólo da sentido a la estructura el uso de un apuntador externo para acceder al comienzo de la lista y luego seguir los apuntadores elemento a elemento.

LA ESTRUCTURA PILA

Definición. Una pila es una estructura de datos en la que los elementos se añaden y se quitan sólo por un extremo; esto es, una pila es una estructura "último en entrar, primero en salir" (last in,first out: LIFO).

A nivel lógico, una pila es un grupo ordenado de elementos. La supresión de los elementos de la pila y la adición de nuevos elementos se hace sólo por el tope o cabeza de la pila. En cualquier momento, dados los elementos de una pila, un elemento está más alto que el otro; es decir, uno está más cerca de la cabeza de la pila. En ese sentido, la pila es un grupo ordenado de elementos, y puesto que los elementos de una pila pueden cambiar constantemente, se considera una estructura dinámica.

Debido a que los elementos se añaden y suprimen de la cabeza de la pila, el último elemento añadido es el primero en quitar. Existe una regla nemotécnica para recordar este comportamiento de la pila: una pila es una lista LIFO (last in, first out), es decir, último en entrar, primero en salir.

Por consiguiente, la función de acceso a una pila es buscar los elementos sólo por la cabeza de la pila, así como la asignación de nuevos elementos a la pila también se hace por la cabeza.

LA ESTRUCTURA COLA.

Se sabe por experiencia que muchas listas operan de manera completamente inversa a como lo hace una pila. Esta forma de operar inversa de estas listas se le conoce como cola.

Definición. Una cola es un grupo ordenado de elementos en el que los elementos se añaden por un extremo (el final) y se suprimen por el otro extremo (el frente). Las colas también se les conoce como listas FIFO (first in, first out: primero en entrar, primero en salir).

La función de acceso a una cola es:

1. Para añadir elementos, se accede por el final de la cola.
2. Para suprimir elementos, se accede por el frente de la cola.

Por consiguiente, se tiene que los elementos de por en medio son inaccesibles a nivel lógico, aunque fácilmente se almacene la cola en una estructura de acceso directo como el arreglo.

Existen muchas aplicaciones en las que las colas figuran como la estructura de datos prominente. Una de ellas es la simulación por computadora de una situación del mundo real. Así por ejemplo, antes de que los astronautas fueran al espacio, gastaron muchas horas en un simulador de vuelo espacial, un modelo físico de un vehículo espacial en el que podían experimentar los astronautas todas las cosas que les sucederían en el espacio.

Este simulador de vuelo espacial es un modelo físico de otro objeto, y la técnica que usaban se llama simulación. En informática se usa la misma técnica para construir modelos en la computadora de objetos y sucesos en vez de modelos físicos.

Un modelo puede verse como una serie de reglas que describen el comportamiento de un sistema del mundo real. Se cambian las reglas y se ve el efecto de estos cambios en el comportamiento de lo que se está observando.

En una simulación por computadora, cada objeto del sistema del mundo real se representa normalmente como un objeto de datos. Las acciones del mundo real se representan como operaciones sobre los objetos de datos. Las reglas que describen el comportamiento determinan las acciones a realizar.

De hecho, el sistema del mundo real se llama un sistema de colas. Un sistema de colas está formado por servidores y colas de objetos a servir, y el comportamiento que se examina es el tiempo de espera.

Los objetivos de un sistema de colas son utilizar a los servidores tan completamente como sea posible, y mantener el tiempo medio de espera dentro de unos límites razonables. Estos objetivos necesitan frecuentemente de un compromiso entre coste y satisfacción del cliente.

Esto a un nivel personal, es de que a nadie le gusta esperar en una fila. Si hubiera una cajera por cada cliente en el supermercado, el cliente estaría encantado, pero el supermercado no duraría como negocio mucho tiempo. Por tanto hay que hacer un

compromiso: el número de cajeras debe de estar dentro del presupuesto del comercio, y el cliente no debe esperar por término medio demasiado tiempo. Esta es una aplicación idónea para una simulación por computadora para examinar este compromiso de un sistema de colas.

ESTRUCTURA MODULAR DE PROGRAMAS.

Existen varios niveles para definir la estructura de datos por el usuario. En el nivel uno, se definen las estructuras lógicamente: ¿ Cuál es su "forma" y cuáles son las operaciones lógicas sobre ellos ? Al siguiente nivel, se construyen una o más implantaciones posibles de las estructuras de datos, usando declaraciones en Pascal y escribiendo subprogramas para desarrollar las funciones de acceso y otras operaciones útiles. En un tercer nivel se usan las estructuras en varios ejemplos de aplicaciones.

Es importante mantener el muro entre los diferentes niveles, ya que esta separación simplifica el diseño, permitiendo aplazar las decisiones sobre los detalles de implantación tanto tiempo como sea posible. Esta ocultación de la información nos permite concentrarnos en el diseño global de la estructura de datos, sin caer en los niveles superiores del diseño de programa, en la fangosidad de los detalles.

Por otra parte, si se separan los procedimientos que usan una estructura de datos de los procedimientos y funciones de utilidad que los implantan, se tiene la libertad de cambiar el nivel de **implantación** sin una modificación sustancial de los niveles más altos del programa, quizás con sólo cambiar parte de las declaraciones. Este encapsulamiento de las estructuras de datos hace a nuestros programas más fácilmente modificables.

Definición: Por encapsulamiento de datos se entiende a la separación de la representación de los datos de las aplicaciones que usan a los datos en el nivel lógico.

Asimismo, otra ventaja de separar la implantación de las estructuras de datos de su uso es la transportabilidad. Si se desea usar la misma estructura de datos en un programa diferente, se puede ahorrar el esfuerzo de la implantación de los procedimientos y funciones. Se puede usar la estructura de datos para una aplicación completamente diferente, pero dejando sin cambiar las funciones de acceso y las operaciones.

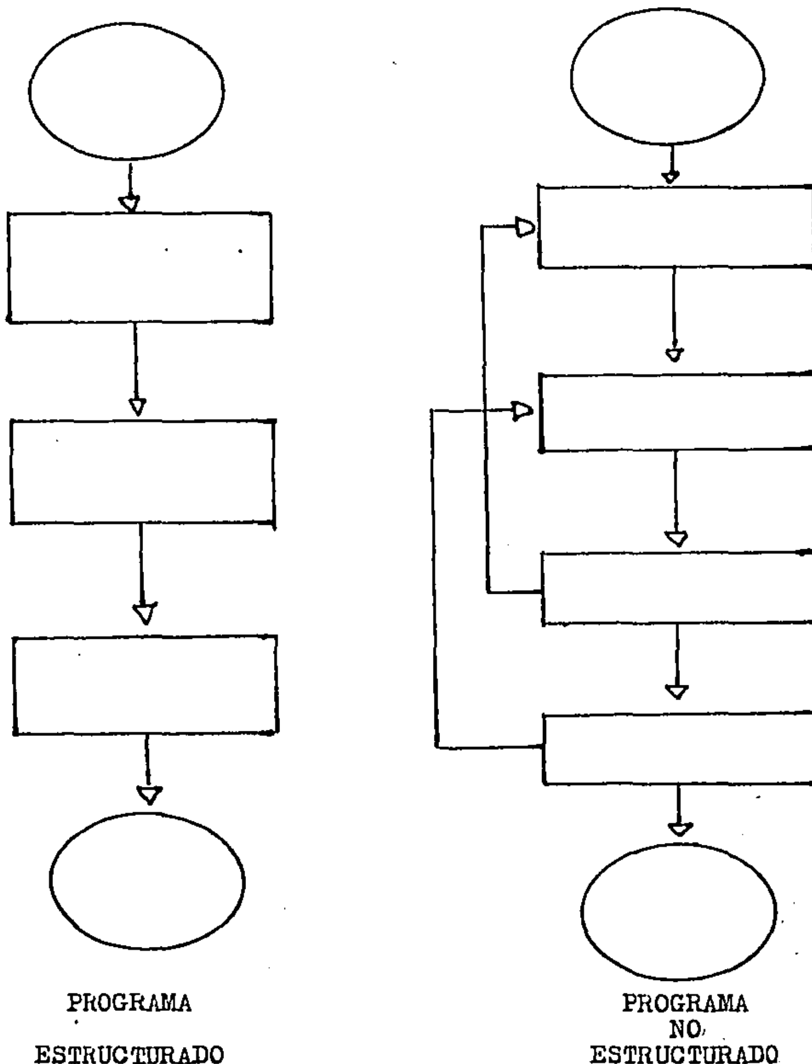


FIGURA 2.8 MODULACION DE UN PROGRAMA

RECURSIVIDAD.

En matemáticas se definen muchos objetos presentando un proceso para producir dicho objeto. Un ejemplo de una definición especificada por un proceso es el de la función factorial, función muy importante en matemáticas y estadística. Dado un entero positivo n, el factorial de n se define como el producto de todos los enteros entre n y 1, y se define de la siguiente forma:

$$n! = \begin{cases} 1 & \text{si } n = 0 \\ n * (n-1)! & \text{si } n > 0 \end{cases}$$

Esta definición puede parecer un poco extraña, puesto que define la función factorial en términos de sí misma. Esto parece ser una definición circular y totalmente inaceptable hasta que nos damos cuenta que la notación matemática es solamente una forma concisa de escribir el número infinito de ecuaciones necesarias para definir n! para cada n. 0! se define directamente como igual a 1. UNa vez definido 0! , el definir 1! como 1 * 0! ya no es de ninguna manera circular. Igualmente se ha definido 1! , definir 2! como 2 * 1! es una forma directa. Se puede argumentar que está última notación es más precisa que la definición de n! como

$$n * (n-1) * (n-2) * \ldots * 1 \qquad \text{para } n > 0$$

puesto que no le deja al lector la oportunidad de llenar por intuición lógica los espacios que están punteados. Tal definición, que define un objeto en términos de una forma simple de él mismo, se denomina <u>definición recursiva</u>.

Obsérvese que la recursión aparece no sólo en matemáticas, sino también en la vida diaria, como en los anuncios que se contienen a sí mismos. Otros ejemplos conocidos de la recursión son los números naturales y las estructuras de árbol:

a) Números Naturales:

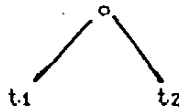
   A. 1 es un número natural.

   B. El siguiente de un número natural es un número natural.

b) Estructuras árbol:

   A. o es un árbol (denominado el árbol vacío).

   B. Si t_1 y t_2 son árboles, entonces



es un árbol.

La potencia de la recursión reside evidentemente en la posibilidad de definir un número infinito de objetos mediante un enunciado finito. De igual forma, un número infinito de operaciones de cálculo puede describirse mediante un programa recursivo finito, incluso si este programa no contiene repeticiones explícitas. De todas maneras, los algoritmos recursivos son apropiados principalmente cuando el problema a resolver, o la función a calcular, o la estructura de datos a procesar, están ya definidos en forma recursiva.

El instrumento necesario y suficiente para expresar los programas recursivamente es el <u>procedimiento</u> (procedure) o subrutina, ya que permite dar un nombre a una instrucción por el

cual ésta puede ser llamada. El uso de la recursión puede no hacerse patente de forma inmediata en el texto del programa. Ahora bien, para que una función o procedimiento recursivo funcione, debe de verificarse:

1. Exista una salida no recursiva del procedimiento o función, y que además funcione correctamente.
2. Cada llamada al procedimiento o función se refiera a un caso más pequeño del problema. Más pequeño significa que hay una parte disminuida del problema que se deja para examinar.
3. Suponiendo que las llamadas recursivas funcionan correctamente, probar que, efectivamente funcione todo el procedimiento o función.

Por otra parte, las aseveraciones usadas para los procedimientos o funciones recursivas, pueden también utilizarse como guía para escribir procedimientos o funciones recursivas. El método siguiente puede usarse para escribir cualquier rutina recursiva:

1. Obtener una definición exacta del problema que hay que resolver. Desde luego, éste es el primer paso en cualquier problema de programación.

2. Determinar el tamaño del problema completo que hay que resolver. Así se determinarán los valores de los parámetros en la llamada inicial al procedimiento o función.

3. Resolver el caso base en el que el problema pueda expresarse no recursivamente. Esto asegurará que exista una salida no recursiva del procedimiento o función.

4. Por último, resolver correctamente el caso general, en términos de un caso más péqueño del mismo problema (una llamada recursiva). Esto asegura las afirmaciones 2 y 3 de las verificaciones anteriores.

EFICIENCIA DE LA RECURSIVIDAD.

En general una versión no recursiva de un programa será realizada más eficientemente en términos de tiempo y espacio, que una versión recursiva. Esto es debido a que la sobrecarga o sobretrabajo comprendido al entrar y salir de un bloque es eliminado en la versión no recursiva. Es posible identificar un buen número de variables locales y temporales que no requieren ser guardadas y restauradas a través del uso de una pila. En un programa no recursivo esta necesidad de apilar puede ser eliminada. Sin embargo, en un procedimiento recursivo el compilador no es capaz de identificar este tipo de variables, y por consiguiente, son colocadas en una pila y vaciadas de la pila para asegurarse de que no se presenta ningún problema.

Algunas veces una solución recursiva es la más natural y lógica para resolver algún problema; por tanto, la recursividad es una herramienta que puede ayudar a reducir la complejidad de un programa ocultando algunos detalles de la implantación. Conforme el costo del tiempo y espacio de memoria de las computadoras disminuye, y aumenta el costo del tiempo del programador, puede ser útil usar soluciones recursivas para tales problemas. Aunque en general, si la solución no recursiva no es mucho más larga que la versión recursiva, usar la no recursiva.

Por otra parte, usar una solución recursiva o una solución no recursiva para resolver algún problema, nos crea un conflicto entre la eficiencia de la máquina y la eficiencia del programador. Teniendo en cuenta que el costo de programación aumenta continuamente y el costo de la computación disminuye, hemos llegado al punto en el cual en muchos casos no se justifica

gastar tiempo de programación para construir una solución no recursiva a un problema que puede ser resuelto en forma más natural a través de un sistema recursivo. Por supuesto, un programador que no tenga mucha experiencia puede resultar con una solución recursiva muy complicada para un problema simple, el cual podía ser resuelto directamente por un método no recursivo. Sin embargo, si un programador con experiencia identifica una solución recursiva como la más simple y como el método más directo para resolver algún problema en particular, no se justifica perder tiempo y esfuerzo al tratar de descubrir algún otro método más eficiente. Aunque claro, este no siempre es el caso.

Si hay que correr un programa muy frecuentemente de tal manera que aumenta la eficiencia en la velocidad de ejecución significativamente, la inversión adicional en tiempo de programación se justifica. Aún en esos casos, es mejor crear una versión no recursiva mediante una transformación y simulación de la solución recursiva a partir del mismo enunciado del problema.

Para realizar esto de una forma más eficiente, lo que se requiere primero es escribir la rutina recursiva y luego la versión simulada. La versión final es una forma refinada del programa original y con seguridad más eficiente. Por supuesto la eliminación de todas esas operaciones superfluas y redundantes mejoran la eficiencia del programa resultante. La extensión en la cual una solución recursiva puede ser transformada en una solución directa dependerá en un buen grado del problema en particular y de la habilidad del programador.

Asimismo, la recursión puede simplificar mucho el diseño y codificación de operaciones sobre algunas estructuras de datos, como por ejemplo, el árbol binario.

CAPITULO III. ESTRUCTURA DE DATOS EN REDES.

1. INTRODUCCIÓN.
2. CONCEPTOS BÁSICOS.
3. ARBOLES BINARIOS.
4. BÚSQUEDA BINARIA.
5. BÚSQUEDA SECUENCIAL.
6. MONTÍCULOS.
7. ANÁLISIS DE LA ORDENACIÓN POR MONTÍCULOS.

INTRODUCCION

Con el transcurso del tiempo surgieron nuevas aplicaciones. Durante este proceso la matemática pura fue siempre por delante, tanto en forma como en contenido. Los griegos incrementaron enormemente lo que habían heredado de los babilonios e incluso añadieron algo nuevo. Transformaron la matemática en un sistema lógico, que comienza con ciertas hipótesis fundamentales y prosigue mediante deducciones lógicas, llamadas premisas, hasta llegar a conclusiones. Esta idea de las matemáticas ha permanecido hasta hoy en día. En matemáticas en muchas ocasiones es suficiente conocer únicamente que una cosa procede de otra de una cierta manera; cualquiera puede convencerse racionalmente de la validez de tales deducciones.

Por otra parte, el tipo de modelo más importante dentro de la Investigación de Operaciones es el modelo simbólico o matemático. Tales modelos son adecuados para el análisis matemático, lo cual usualmente crea la posibilidad de encontrar la mejor solución por medio de herramientas matemáticas convenientes. Para el desarrollo de algunos aspectos de la Investigación de Operaciones, fue necesario el crecimiento de otras áreas de las matemáticas aplicadas, como fue el caso de la Teoría de Redes. La Teoría de Redes se utiliza mucho como modelo en las distintas ramas del conocimiento humano. En muchos casos las gráficas que se desarrollan permiten visualizar los distintos elementos que constituyen una situación, y las relaciones que hay entre ellos.

La teoría de redes aporta una ayuda muy eficaz en ciertos problemas de carácter combinatorio que aparecen en diversos dominios. Por la riqueza que encierra esta área, es fundamental en la concepción de la estructura de datos y sus múltiples aplicaciones. En este capítulo, se definen las ideas y los conceptos principales de la teoría de redes, así como su relación con la estructura de datos.

CONCEPTOS BASICOS.

La teoría de redes puede aportar una ayuda muy eficaz en el tratamiento de ciertos problemas de carácter combinatorio que aparecen en diversos dominios económicos, sociológicos o tecnológicos; asimismo, por su contenido, la teoría de redes tiene un sitio muy importante en la enseñanza.

PUNTOS Y ARCOS PARA REPRESENTAR ESTRUCTURAS.

Consideremos un conjunto de puntos finitos, distintos, como los puntos 1, 2, 3, . . . Esos puntos que llamaremos también vértices o nodos están unidos por líneas , ya sea orientadas o no ,las que llamaremos arcos. Puede pensarse a la red como un conjunto de arcos y nodos, tal que existe una función f definida de la siguiente manera:

$$f: \; N \times N \longrightarrow A$$
$$f(a,b) = c \qquad \forall \; a,b \in N, \; c \in A$$

donde N es el conjunto de nodos y A el conjunto de arcos.

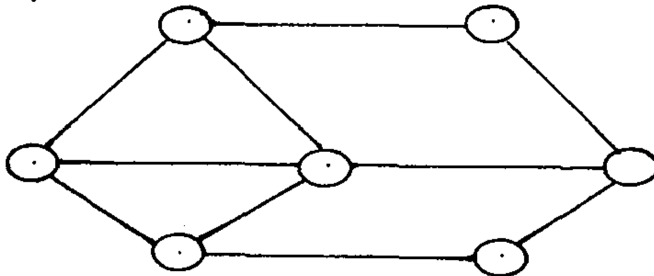


Figura 3.1 Red.

Podemos representar estructuras muy diversas mediante una red, como por ejemplo:

i) Un Sistema de caminos o de calles.

ii) Un Sistema Eléctrico.

iii) Un grupo humano.

iv) La circulación de información de un sistema.

v) Las relaciones de parentesco entre un grupo de individuos.

vi) La superioridad de los participantes en un torneo.

Debemos de considerar que la red de cada uno de los ejemplos anteriores no debe de confundir el concepto al que está asociado: es solamente la estructura para la cual admitimos que una descripción con la ayuda de nodos y arcos es plenamente adecuada en lo que concierne a ciertas propiedades interesantes.

CONCEPTOS ORIENTADOS.

CAMINO. Es una sucesión de arcos adyacentes que permiten pasar de un vértice a otro siguiendo los arcos.

CIRCUITO. Es un camino en el cual el nodo inicial coincide con el nodo final.

LONGITUD DE UN CAMINO O CIRCUITO. Es el número de arcos en el camino o circuito.

LAZO. Es un circuito de longitud uno.

RED SIMETRICA. Si un nodo X está conectado a un nodo Y, entonces Y debe estar conectado con X. Si se cumple para todos los nodos entre los que existe una correspondencia, entonces la red es simétrica.

RED FUERTEMENTE CONECTADA. Cualesquiera que sean los nodos X y Y ($X \neq Y$) considerados, existe un camino de X a Y.

CONCEPTOS SIN ORIENTACION.

ARISTA. Existe una arista entre dos vértices X y Y si hay un arco de X a Y y/o Y a X.

CADENA. Es una secuencia de aristas consecutivas.

CICLO. Es una cadena cerrada.

RED CONECTADA. Cualesquiera que sean los vértices X y Y considerados, existe una cadena entre X y Y.

Definición de Arbol. Es una red conectada que no contiene ningún ciclo.

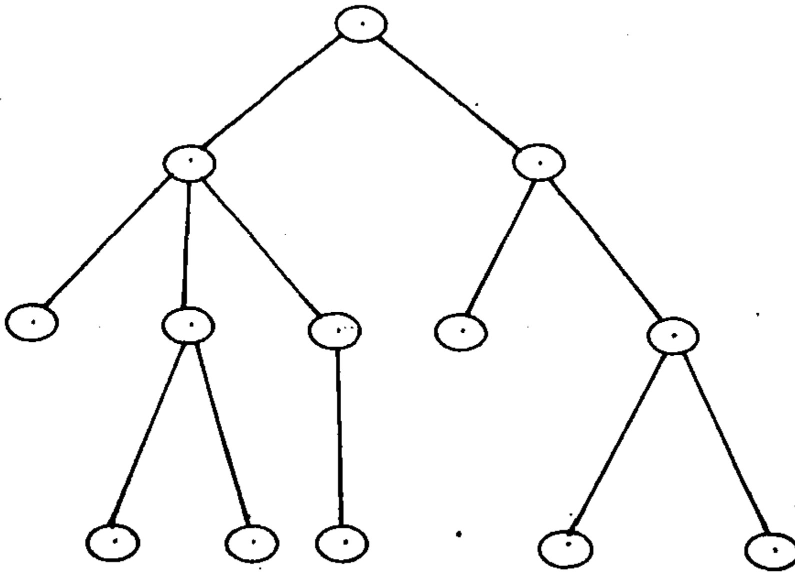


FIGURA 3.2 Arbol en una red

ARBOLES BINARIOS.

El concepto de lista enlazada puede extenderse a estructuras que contienen nodos con más de un campo apuntador. Una de estas estructuras se conoce como árbol. Se llama árbol binario al árbol en el cual cada nodo tiene como máximo dos descendientes.

El árbol se referencia mediante un apuntador externo a un nodo especial llamado raíz. La raíz tiene dos apuntadores: uno a su hijo izquierdo y otro a su hijo derecho. Cada hijo tiene a su vez dos apuntadores: uno a su hijo izquierdo y otro a su hijo derecho. Los árboles binarios de búsqueda son árboles que tienen la propiedad de que la información de cualquier nodo es mayor que la información de cualquier nodo de su hijo izquierdo y de cualquier nodo de sus hijos, y menor que la información de su hijo derecho y cualquier nodo de sus hijos.

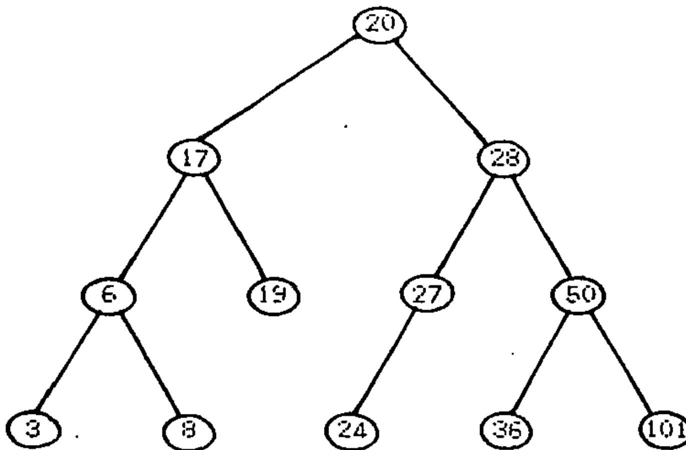


Figura 3.3

El árbol anterior es un ejemplo de árbol binario de búsqueda. La razón de su utilidad está en que si se busca un cierto número, se puede saber después de una comparación en qué mitad del árbol está . Con otra comparación se puede saber en qué mitad de esa mitad estaría. Así se continúa hasta que se encuentre el elemento, o se sepa que no está en el árbol.

Probemos con el número 50.

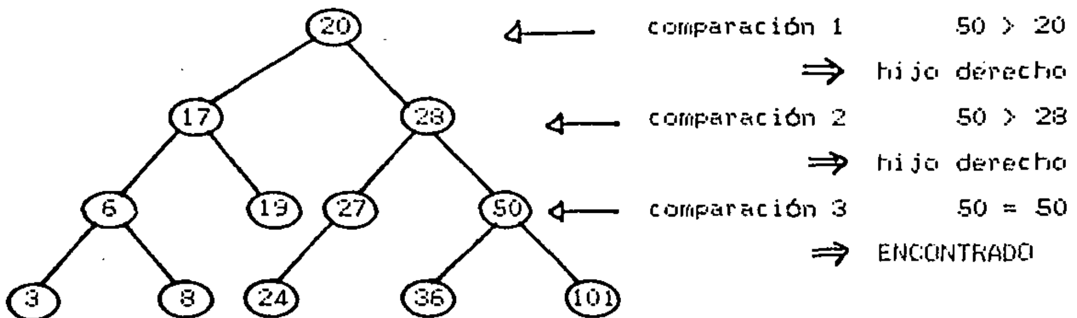


Figura 3.4

Veamos ahora para el 18, un número que no está en el árbol.

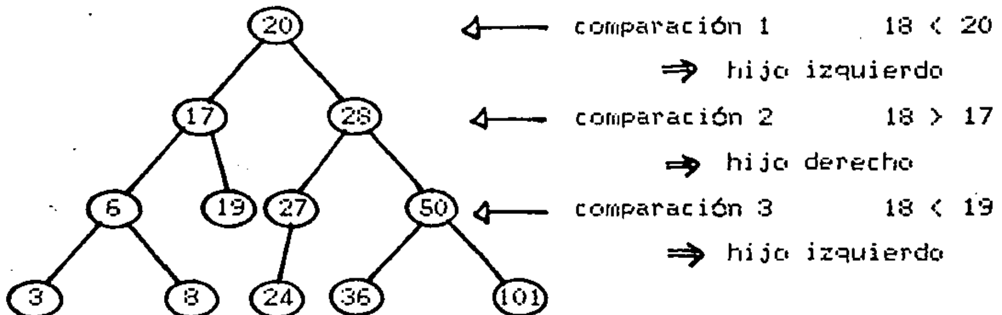


Figura 3.5

El hijo izquierdo de 19 es NIL (Vacío), por lo que el número 18 no está en el árbol. No sólo sabemos que no está allí, sino que si queremos insertarlo, estamos en el lugar en el que se insertaría.

BUSQUEDA BINARIA.

La ventaja de la búsqueda secuencial es su simplicidad. En el peor caso, habrá que hacer N comparaciones, puesto que sólo se examina un registro cada vez y puede tener que buscarse el último registro de la lista. Sin embargo, si la lista está ordenada y almacenada en un arreglo, se puede mejorar el tiempo de búsqueda en el peor caso hasta $O(\log_2 N)$. Desde luego, se mejorará la eficiencia a expensas de la simplicidad.

La idea de una búsqueda binaria se describe mejor recursivamente. Veámoslo en un pseudocódigo

```
Examinar el elemento mitad de la lista
IF el elemento mitad contiene la clave deseada
   THEN parar la búsqueda
   ELSE IF el elemento mitad es
      mayor que la clave deseada
     THEN búsqueda binaria en la primera mitad de la lista
     ELSE (el elemento mitad es más pequeño que la clave )
         búsqueda binaria en la segunda mitad de la lista.
```

Por ejemplo, consideremos cómo podría usarse este algoritmo para encontrar la página con el nombre de "Diana" en la guía telefónica. Abrimos la guía telefónica por la mitad y vemos que los nombres que hay allí comienzan con M. M es mayor que D, por lo que hacemos una búsqueda binaria desde A hasta M. Volvemos al punto mitad y vemos que los nombres de allí comienzan con G. G es mayor que D, por lo que hacemos una búsqueda binaria desde A hasta G.

Escogemos de nuevo la página mitad y encontramos que los nombres que hay allí comienzan con C. C es más pequeño que D, por lo que hacemos una búsqueda binaria en la segunda mitad (es decir, desde C hasta G). Y así sucesivamente hasta que damos con la página que contiene el nombre de "Diana".

Por consiguiente, la búsqueda como la ordenación, es un tema que está ligado estrechamente al objetivo de la eficiencia. Hablamos de la búsqueda secuencial como una búsqueda O(N), puesto que necesita N comparaciones para localizar un elemento (N se refiere al número de registros de la lista). La búsqueda binaria está considerada O(log2N) y es apropiada sólo en arreglos. Asimismo, para permitir la búsqueda binaria en estructuras enlazadas, puede usarse un árbol binario de búsqueda.

BUSQUEDA SECUENCIAL.

ESTA TESIS NO DEBE
SALIR DE LA BIBLIOTECA

La técnica de búsqueda más sencilla es la búsqueda secuencial. Se comienza por el principio de la lista y se va buscando el registro deseado, examinando los registros secuencialmente hasta que se encuentra el registro deseado o se recorre toda la lista. Esta técnica es apropiada tanto para listas secuenciales como enlazadas. La lista no tiene que estar ordenada, aunque la eficiencia de la búsqueda puede mejorarse si la lista está ordenada.

Basándondos en el número de comparaciones, es evidente que está búsqueda es O(N), donde N es el número de elementos. En el peor caso, en el que estamos buscando el último registro de la lista o un registro no existente,. tendremos que hacer N comparaciones. En el caso medio, suponiendo que existe la misma probabilidad de buscar cualquier elemento de la lista, haremos N/2 comparaciones, esto es, en el caso medio tendremos que buscar la mitad de la lista.

MONTICULOS.

Un montículo se define como una secuencia de llaves
$h_1, h_2, \ldots, h_n$ tales que

$$h_i \leq h_{2i}$$
$$h_i \leq h_{2i+1}$$

para todos los valores de $i = 1, 2, \ldots, n/2$. Si se representa un árbol binario como en la figura 3.6, se deduce que los árboles de ordenación de las figuras 3.7 y 3.8 son montículos, y que en particular el elemento h_1 de un montículo es el elemento mínimo.

$$h_1 = \min \{ h_1, h_2, \ldots, h_n \}$$

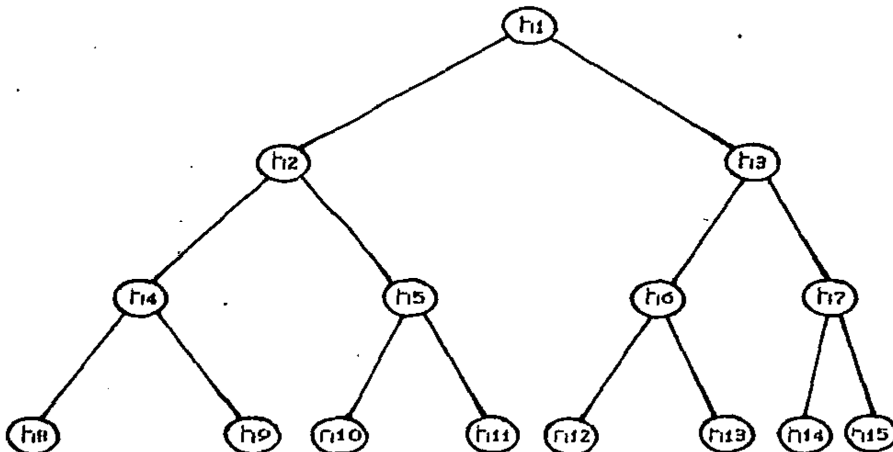


Figura 3.6. El arreglo H en forma de árbol binario.

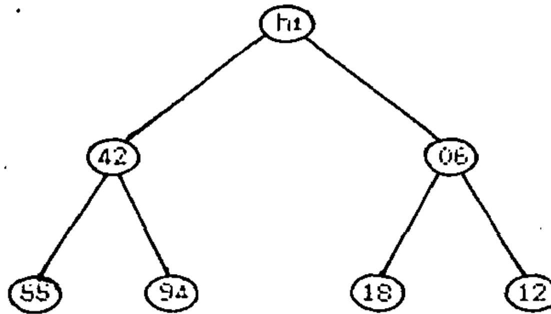


Figura 3.7. Monticulo con siete elementos.

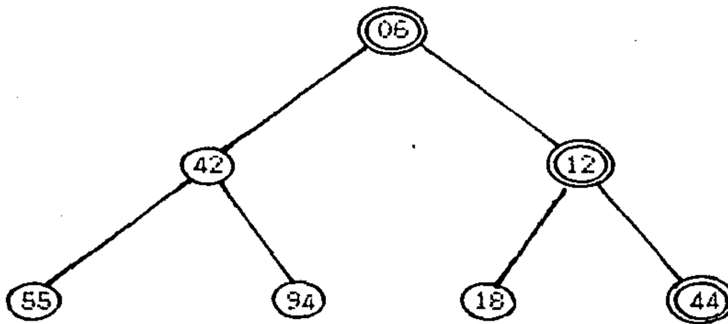


Figura 3.8 . Hundimiento de la llave 44 a través del montículo.

De acuerdo a la definición anterior, un montículo es una estructura de datos abstracta consistente de una colección de datos, cada uno asociado a una llave. Dos operaciones son posibles en un montículo:

1. INSERTAR. Inserta el dato i en el montículo h, i previamente no contenido en h.
2. ELIMINAR. Elimina y retorna un dato de llave mínima del montículo h; si h es vacío retorna 'nulo'.

Asimismo, la siguiente operación crea un nuevo montículo:

3. CREAR. Construye y retorna un nuevo montículo cuyos datos son los elementos del conjunto s.

Por consiguiente, estas tres operaciones algunas veces nos permiten otras, como son:

4. ENCONTRAR. Retorna pero no elimina un dato de llave mínima desde el montículo h; si h es vacío retorna 'nulo'.

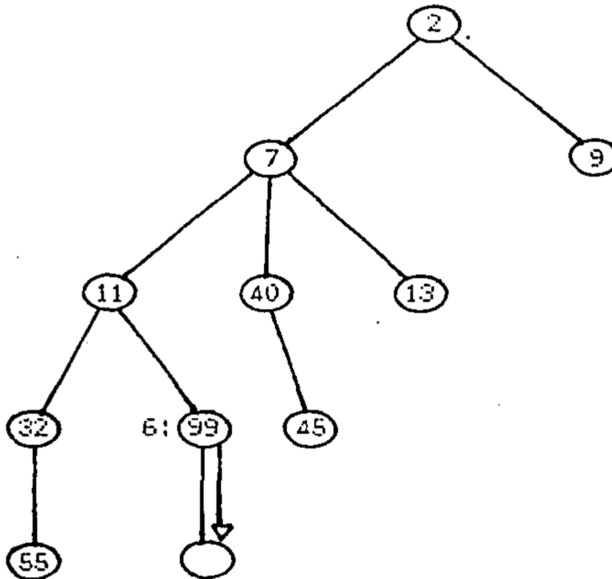
5. ELIMINAR. Elimina el dato i del montículo h.

6. MEZCLA. Retorna el montículo formado por la combinación disjunta de los montículos h_1 y h_2. Esta operación destruye h_1 y h_2.

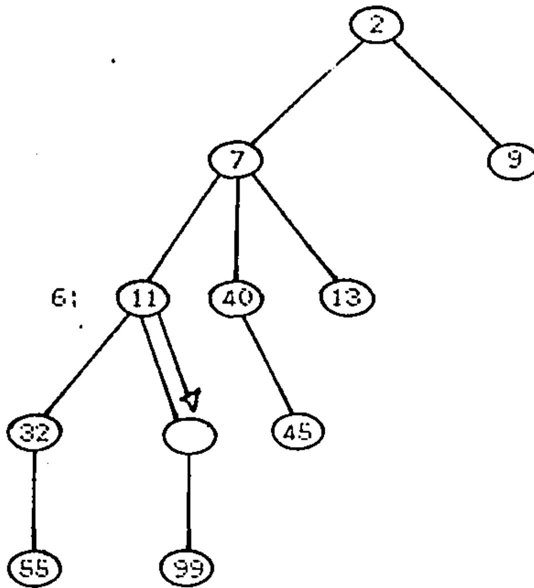
Veamos el caso de insertar. Se puede insertar un nuevo dato como sigue:
Se crea un sitio para el dato i, se añade un nuevo nodo x vacante en el árbol; el padre de x, p(x), es arbitrario, pero x no debe tener hijos.
Se almacena i en x; si el padre de x, p(x), contiene un dato cuya llave exceda al de la i, se viola el orden del montículo, sin embargo puede remediarse haciendo un examen hacia arriba como sigue:

Mientras que p(x) esté definido y contenga un dato cuya llave exceda a la llave de i, guardamos en x el dato que está en p(x), después se reemplaza el nodo vacante x por p(x), y se repite. Cuando el examen para, se almacena i en x. (ver figura 3.9).

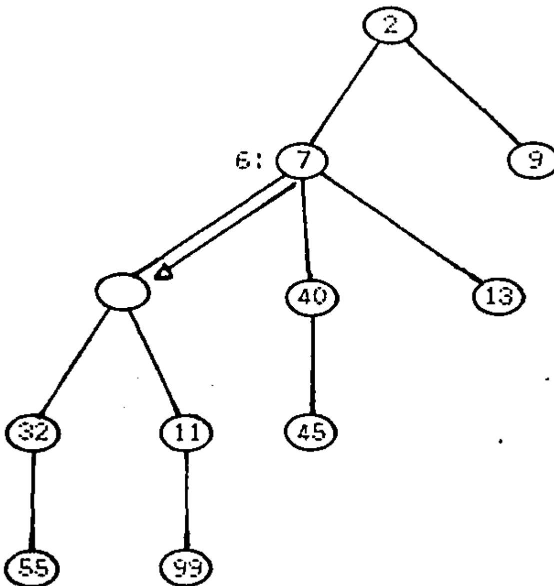
Figura 3.9 Inserción de la llave 6 por examen superior.



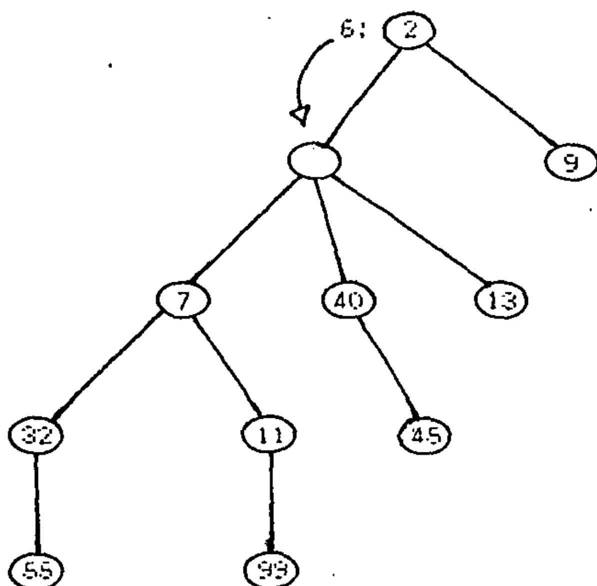
a) Creación de una nueva hoja. Puesto que 6 < 99, 99 se mueve hacia el nuevo nodo.



b) Puesto que 6 < 11, 11 se mueve hacia el nodo vacante.



c) Puesto que 6 < 7, 7 se mueve hacia el nodo vacante.

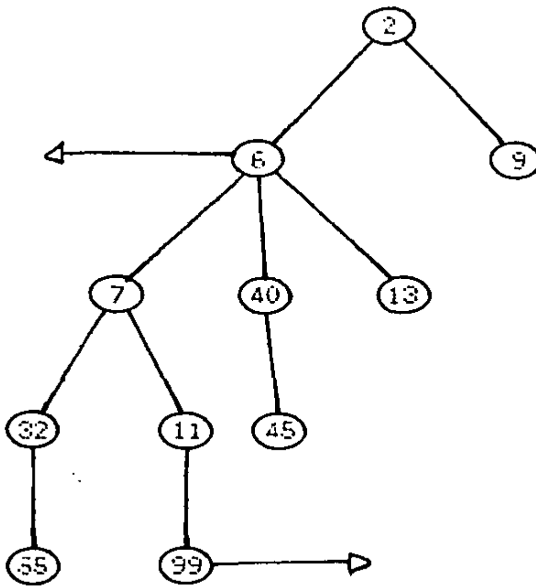


d) Puesto que 6 ≥ 2, 6 se mueve hacia el nodo vacante y el examen superior se detiene.

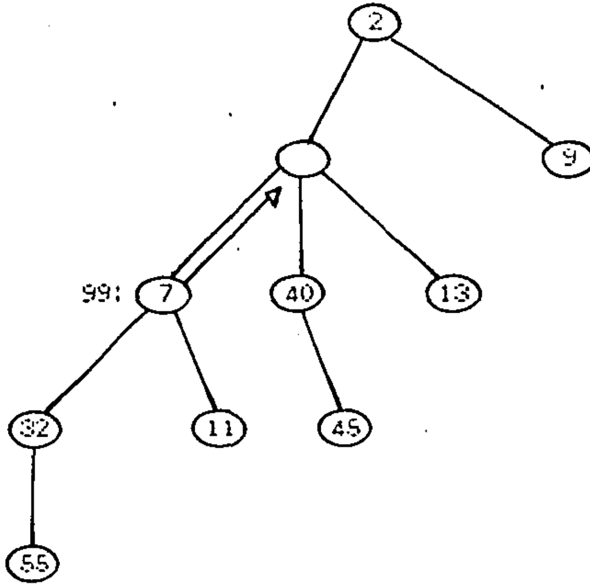
La eliminación de un dato es más complicado que la inserción. Para eliminar un dato i, comenzamos por buscar un nodo y que no tenga hijos. Se remueve el dato, digamos j, desde y y se elimina y del árbol. Si $i = j$ hemos acabado; en otro caso, se remueve i del nodo, digamos x, conteniéndolo, y se procura el reemplazo de él por j. Si la llave(j) ≤ llave(i), reinsertamos j por un examen hacia arriba empezando desde x. Si la llave(j) > llave(i) reinsertamos j por un examen hacia abajo comenzando desde x.

Mientras que la llave(j) sea mayor que la llave de algún hijo de x , se elige un hijo c de x conteniendo un dato de llave minima, almacenamos en x el dato de c, reemplazamos x por c , y se repite el proceso (ver figura 3.10). Cuando el examen se detiene, se almacena j en x .

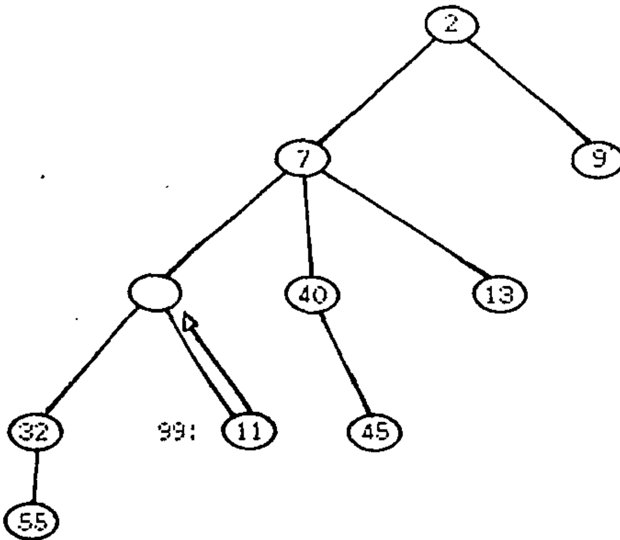
Figura 3.10. Eliminación de la llave 6 por examen inferior.



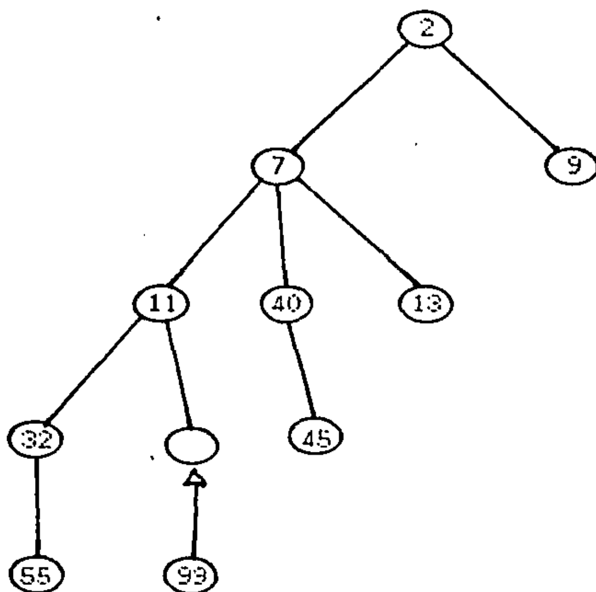
a) El último nodo creado contiene la llave 99. Se destruye la última hoja. La llave 99 puede ser reinsertada.



b) La llave más pequeña de los hijos del nodo vacante es 7. Puesto que 99 > 7, 7 se mueve hacia el nodo vacante.



c) La llave más pequeña de los hijos del nodo vacante es 11. Puesto que 99 > 11, 11 se mueve hacia el nodo vacante.



d) Puesto que el nodo vacante no tiene hijos, 99 se mueve hacia él y el examen inferior se detiene.

ANALISIS DE LA ORDENACION POR MONTICULOS.

Es difícil creer que la ordenación por montículo sea realmente eficiente. Parece extraño tener que mover el menor valor por varios sitios hasta llegar al tope, antes de ponerlo en su lugar final. Y de hecho, para pequeños valores de N, la ordenación por montículo no es muy eficiente.

Sin embargo, para grandes arreglos la ordenación por montículo se convierte en un método muy eficiente. Considerar que un árbol binario completo con N nodos tiene log(N+1) niveles. Aunque cada elemento fuera una hoja y tuviera que pasar por todo el árbol entero, la ordenación sería $O(N \log_2 N)$. Por lo que la ordenación por montículo, a diferencia de la ordenación rápida, es $O(N \log_2 N)$ independientemente del orden inicial de sus elementos.

Capítulo IV. Problemas de Aplicación.

1. Introducción.
2. Conectividad de una Red.
3. Flujo Máximo.
4. Corte Mínimo.
5. Pintado de la Red en Flujo Máximo.

INTRODUCCION

El presente capítulo trata sobre algunas aplicaciones en el flujo de redes. Algunas aplicaciones de la Estructura de Datos a la teoría de redes son: la conectividad de una red, flujo máximo (Ford - Fulkerson), y el Pintado de la Red en flujo máximo. Aunque existen otros problemas de redes tales como el árbol de expansión mínima, flujo a costo mínimo, flujo a costo mínimo en redes con ganancia , transporte-asignación, ruta más corta, etc. estos problemas no se tratan aquí.

Para el Flujo Máximo, aplicando el algoritmo de Ford-Fulkerson, su fundamento radica en la factibilidad de un flujo inicial igual a cero. Con respecto al Pintado de la Red para hallar el flujo máximo, su trascendencia consiste en la búsqueda de trayectorias de un nodo fuente a un nodo destino por medio del pintado de los arcos, teniendo como inicio un flujo factible no negativo.

GENERALIDADES.

En el análisis de redes de flujo, la restricción más común es acotar el flujo a través de un arco; es decir, se exige que el valor del flujo esté en un intervalo cerrado no vacío. C(j) llamado intervalo de capacidad para j. Un flujo x en la red G es factible con respecto a las capacidades se $x(j) \in C(j)$, para toda $j \in A$.

Los intervalos de capacidad se denotarán así:

$$C(j) = \left[C^-(j), C^+(j) \right]$$

en donde $C^-(j)$ y $C^+(j)$ son las capacidades inferior y superior respectivamente para el arco j. Las únicas restricciones que deben satisfacer $C^-(j)$ y $C^+(j)$ son:

$$C^-(j) \leq C^+(j)$$
$$C^+(j) > -\infty$$
$$C^-(j) < \infty$$

Estos intervalos de capacidad tienen como consecuencia indirecta algunos resultados importantes. Uno de ellos restirnge el flujo que puede pasar a través de un corte. El flujo de x a través de Q, donde x es un flujo definido en la red G y Q es un corte de G, se define como la cantidad:

$$e_Q \cdot x = \sum_{j \in Q^+} x(j) - \sum_{j \in Q^-} x(j)$$

CONECTIVIDAD DE UNA RED.

Sea G una red, N el conjunto de nodos, y A el conjunto de arcos de que consta la red. Se dice que una gráfica es conexa si para toda pareja de nodos (i,j) existe una trayectoria P que los une.

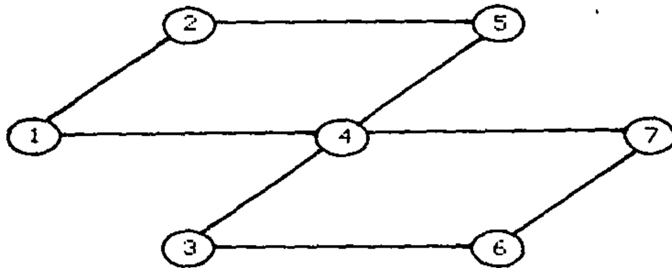


Figura 4.1 Red Conexa

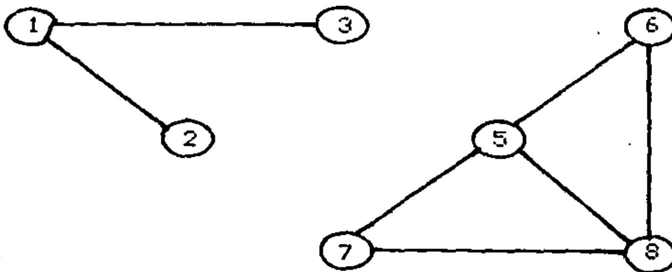


Figura 4.2 Red No conexa.

Definición. La matriz de adyacencia X de la red G se define sobre N x N de la siguiente manera:

$$a(i,j) = \begin{cases} 1 & \text{si } (i,j) \text{ es un arco.} \\ 0 & \text{si } (i,j) \text{ no es un arco.} \end{cases}$$

Aunque es muy fácil determinar desde un diagrama cuando una gráfica es conexa o no, es más dificil cuando las componentes de la red son dadas en forma numérica para su uso computacional.

FLUJO MAXIMO

Considermemos un sistema de tuberías de transporte de agua como se muestra en la figura 4.3 . Cada arco representa un tubo y el número de encima de cada arco representa la capacidad de esta tubería en galones por minuto. Los nodos representan puntos en los cuales las tuberías están unidas y el agua es transportada de un tubo a otro. Dos nodos, S y T, representan la fuente de agua, y un usuario de ésta, respectivamente. Esto significa que el agua nace o se origina en S y es llevada a través de la tubería del sistema hasta T. El agua fluye a través de la tubería en una sola dirección y no existen tuberías entrando a S o saliendo de T. Por consiguiente, la red con los números asociados a cada arco (factores de peso) es ideal para modelar esta situación.

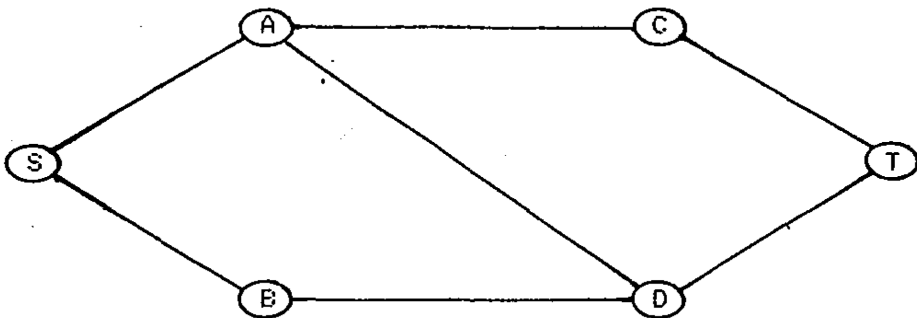


Figura 4.3 Problema de Flujo.

Queremos maximizar la cantidad de agua que fluye desde la fuente hasta su punto de consumo. Aún cuando la fuente puede producir cualquier cantidad de agua a un determinado flujo y el usuario es capaz de consumir agua en la misma proporción, el sistema de tubería puede que no tenga la capacidad suficiente para transportarla desde la fuente hasta el usuario. Esto es, los factores limitantes de todo el sistema es la capacidad de la tubería.

Otros problemas del mundo real son análogos en cuanto a su naturaleza, como por ejemplo, una red eléctrica, un sistema de transporte, un sistema de comunicaciones, o cualquier otro sistema de distribución en el cual uno desea maximizar la cantidad de algún elemento que es enviado desde un punto hasta otro.

Ahora bien, consideremos los resultados dados a continuación referentes al flujo máximo en una red.

CONCEPTOS SOBRE TRAYECTORIAS DE UNA RED.

Una trayectoria P en una red G es una secuencia finita de la forma:

$$i_0 , j_1 , i_1 , j_2 , \ldots , j_r , i_r \quad \text{para } r > 0$$

donde $i_k \in N$, para todo $k = 1,2,\ldots,r$ y $j_k \sim (i_{k-1}, i_k)$ ó $j_k \sim (i_k , i_{k-1})$. Es decir, no importa la dirección del arco j_k .

Si el arco es de la forma $j_k \sim (i_{k-1} , i_k)$ se dice que j_k se recorre positivamente; si por el contrario $j_k \sim (i_k , i_{k-1})$ se dice que j_k se recorre negativamente. Una trayectoria para la cual todos los arcos son recorridos positivamente es una trayectoria positiva; si todos los arcos son recorridos negativamente, entonces se tiene una trayectoria negativa.

En una trayectoria P se puede particionar el conjunto de arcos en los recorridos positivamente y los recorridos negativamente, denotándose estos conjuntos como P^+ y P^- respectivamente.

Sea N^+ el conjunto de nodos donde inician las trayectorias de P, y sea N^- el conjunto de nodos donde finalizan las trayectorias de P.

Sea $S' = [N/S]$ (Complemento de S en N).

Se definen los siguientes conjuntos de la manera siguiente:

$$Q^+ = [S, S']^+ = (j \in A / j \sim (i , i'), i \in S , i' \in S')$$

$$Q^- = [S, S']^- = (j \in A / j \sim (i' , i), i \in S , i' \in S')$$

En una red pueden distinguirse cuatro clases de arcos, según su color: verde, negro, blanco y rojo. La clase de arcos verde es aquélla que se puede recorrer en ambos sentidos, la clase negra es aquélla que se recorre en sentido contrario al suyo, la clase blanca es la que puede ser recorrida sólo en su sentido, y finalmente, la clase roja es la que no puede recorrerse en ningún sentido. De esta manera se define una partición de un conjunto de arcos A en cuatro suoconjuntos llamada coloración de A.

Con el concepto de coloración es posible caracterizar las restricciones de una determinada trayectoria; por ejemplo, que ésta sea positiva. Asimismo, se desea determinar una trayectoria en la cual se respeten las restricciones de recorrido para cada arco. En términos de la coloración de A, el problema puede establecerse de la siguiente manera:

Trayectoria compatible con la coloración: Es una trayectoria consistente en los colores Verde, Blanco, Negro y Rojo tal que

$$P : N^{+} \longrightarrow N^{-}$$

es decir, $P : i \longrightarrow ir$ donde $i \in N^{+}$ & $i \in N^{-}$ y todo arco de P^{+} es Verde o Blanco, y todo arco de P^{-} es Verde o Negro.

ALGORITMO DE ENRUTAMIENTO.

Sea $\theta : S/N^{+} \longrightarrow A$ la cual asigna o etiqueta a cada nodo del dominio un arco. Esta función recibe el nombre de enrutamiento de S con base en N^{+} y debe satisfacer:

a) Para cada nodo i de S/N^{+}, (i) es un arco que une i con algún nodo de S.

b) Cuando se genera una secuencia i_0 , $\Theta(i_0)$, i_1 , $\Theta(i_1)$, ..., i_k en donde i es el otro extremo de $\Theta(i_k)$, eventualmente se alcanza un nodo de N^+ . El reverso de esta secuencia es entonces una trayectoria de N^+ a i .

Propósito: Determinar una trayectoria de N^+ a N^- compatible con la coloración dada.

Descripción:

Paso 1. Sea $S = N^+$ y sea Θ vacio.

Paso 2. Determínese el conjunto $Q = [S, N/S]$

- Si existe $j \in Q^+$ tal que j es Verde o Blanco ó si existe $j \in Q^-$ tal que j es Verde o Negro ir al paso 3
- Si no existe tal j terminar. En este caso no existe solución al problema.

Paso 3. Sea $\Theta(i) := j$, en donde $i \notin S$.
Sea $S := S \cup \{i\}$. (El enrutamiento es compatible con la coloración.

- Si $i \in N^-$ terminar. Los arcos del enrutamiento forman una trayectoria P de N^+ a $i \in N^-$ compatible con la coloración.
- Si $i \notin N^-$ se tiene $S \cap N^- = \phi$. Ir al paso 2.

CORTE MINIMO.

PROBLEMA DE FLUJO MAXIMO.

Maximizar el flujo de N^+ a N^- sobre todos los flujos x factibles con respecto a las capacidades. El supremo en el problema de flujo máximo es la mínima cota superior del conjunto de valores de flujo de N^+ a N^- de todos los posibles flujos x. Si este supremo es finito entonces el flujo x que produzca este valor es la solución requerida del problema de flujo máximo.

Como se puede observar, todas las trayectorias de N^+ a N^- utilizan algún arco de cualquier corte $Q : N^+ \longrightarrow N^-$ y de este modo los cortes constituyen "cuellos de botella" para el valor del flujo de N^+ a N^-. Es decir

<u>Proposición</u>. Sea x un flujo que satisface las restricciones del problema y sea $Q : N^+ \longrightarrow N^-$ un corte que separa N^+ de N^- entonces

$$\left[\text{flujo de } x \text{ de } N^+ \text{ a } N^-\right] \leqslant c^+(Q)$$

El problema de corte mínimo es el problema dual del flujo máximo.

PROBLEMA DE CORTE MINIMO.

Minimizar $c^+(Q)$ sobre todos los cortes $Q : N^+ \longrightarrow N^-$
Nótese que un corolario de la proposición anterior es la desigualdad siguiente

$$\begin{bmatrix}\text{Supremo en problema de} \\ \text{flujo máximo}\end{bmatrix} \leq \begin{bmatrix}\text{Mínimo en problema de corte} \\ \text{mínimo}\end{bmatrix}$$

La relación más importante entre los dos problemas es el hecho de que la igualdad se cumple en los óptimos.

Una trayectoria $P : N^+ \longrightarrow N^-$ es aumentante para el flujo x si $x(j) < c^+(j), \forall\, j \in P^+$ & $x(j) > c^-(j), \forall\, j \in P^-$.
Claramente, el flujo x puede mejorarse a través de una trayectoria aumentante. En efecto, puede garantizarse la existencia de un número $\alpha > 0$ tal que

$$x'(j) = x(j) + \alpha\, e_P(j) = \begin{cases} x(j) + \alpha & , j \in P^+ \\ x(j) - \alpha & , j \in P^- \\ x(j) & , j \notin P \end{cases}$$

TEOREMA DE FLUJO MAXIMO - CORTE MINIMO.

Supóngase que existe al menos un flujo x que satisface todas las restricciones del problema. Entonces:

$$\begin{bmatrix}\text{Supremo en problema} \\ \text{de flujo máximo}\end{bmatrix} = \begin{bmatrix}\text{Mínimo en problema} \\ \text{de corte mínimo}\end{bmatrix}$$

Cabe señalar que si existe una trayectoria de capacidad ilimitada, entonces ambos problemas tienen valor $+\infty$; en caso contrario, los valores de ambos, iguales, son finitos y el problema de flujo máximo tiene, por tanto solución.

Una de las conclusiones que pueden obtenerse del teorema de flujo Máximo - Corte Mínimo es que ambos problemas pueden ser resueltos simultáneamente. Debe recordarse que, puesto que estos problemas son duales, a partir de la solución de uno puede obtenerse la solución del otro.

Para construir un flujo aumentante, se comienza con cualquier flujo que cumpla las restricciones y en cada iteración se buscará enviar más flujo de N^+ a N^- a través de trayectorias aumentantes de flujo hasta que esto no sea posible.

Obsérvese que si existe una trayectoria aumentante para el flujo x de capacidad ilimitada, el valor máximo de x no es finito, y por lo tanto, el supremo en el problema de flujo máximo es $+\infty$. Por otro lado, el criterio de terminación se establece mediante la existencia de un corte Q.

ALGORITMO DE FORD Y FULKERSON.

Propósito: Determinar el flujo máximo de N^+ a N^- en una red G en la cual no existen trayectorias de capacidad ilimitada.

Descripción.

Paso 1. Determinar x, un flujo que satisfaga todas las restricciones del problema.

Paso 2. Colorear los arcos de G de acuerdo a:

verde	si $c^-(j) < x(j) < c^+(j)$
blanco	si $c^-(j) = x(j) < c^+(j)$
negro	si $c^-(j) < x(j) = c^+(j)$
rojo	si $c^-(j) = x(j) = c^+(j)$

Paso 3. Determinar una trayectoria compatible con la coloración, es decir, una trayectoria aumentante para x.

Si se determina la trayectoria $P : N^+ \longrightarrow N^-$ compatible con la coloración, calcular

$$\alpha = \mathrm{Min} \begin{cases} c^+(j) - x(j), & j \in P^+ \\ x(j) - c^-(j), & j \in P^- \end{cases}$$

Hacer $x = x + \alpha e_P$ y regresar al paso 2.

Si se determina un corte $Q : N^{+} \longrightarrow N^{-}$ compatible con la coloración; terminar con el flujo máximo x y el corte mínimo Q, ya que este último satisface

$c^{+}(j) = x(j),\ j \in Q^{+}$ & $c^{-}(j) = x(j),\ j \in Q^{-}$

y por lo tanto

$$\text{flujo de } x \text{ a través de } Q = \sum_{j \in Q^{+}} x(j) - \sum_{j \in Q^{-}} x(j) = c^{+}(Q).$$

TRAYECTORIA DE UNA RED.

Una consideración importante en la determinación del flujo máximo en una red es la dirección en que puede circular flujo a través de un arco; es decir, debe tenerse en cuenta si el flujo puede aumentarse, decrementarse, ambas cosas o ninguna de ellas. En este aspecto es necesario "etiquetar" los arcos para que se distinga qué cambios requeridos son factibles para el flujo a través de un arco. Esta etiqueta estará dada por la introducción de una "coloración" del arcos.

CORTE Y DUALIDAD.

Se dice que el corte Q separa N^+ de N^- si es de la forma $[S,N/S]$, para algún $S \subset N$, tal que $N^+ \subset S$ y $N^- \cap S = \emptyset$

PROBLEMA DEL CORTE COLOREADO.

Sean N^+ y $N^- \subset N$ tales que $N^+ \cap N^- = \emptyset$. Sea una coloración en la red con los colores verde, blanco, negro y rojo. El problema es determinar un corte $Q: N^+ \longrightarrow N^-$ tal que todo arco de Q^+ sea rojo o negro, mientras que todo arco de Q^- sea rojo o blanco.

Un corte que cumple las restricciones de color se dice compatible con la coloración, y si además separa N^+ de N^- constituye la solución al problema del corte coloreado. Nótese la dualidad entre las restricciones de color para trayectorias y cortes.

TEOREMA DE LA RED COLOREADA.

Sean $N^+ \subset N$ y $N^- \subset N$, tales que $N^+ \cap N^- = \emptyset$. Entonces, para toda coloración de la red, llámese G, con los colores verde, blanco, negro y rojo, una y sólo una de las siguientes afirmaciones es válida:

1. El problema de la trayectoria coloreada tiene solución P.
2. El problema del corte coloreado tiene solución Q.

Si termina en el paso 2, entonces se ha construido un corte $Q:N^+ \longrightarrow N^-$ compatible con la coloración; la existencia de este corte garantiza entonces la no existencia de una trayectoria compatible con la coloración.

Otro resultado fuertemente relacionado con el teorema de la red coloreada es el lema de Minty y se utiliza frecuentemente de manera constructiva. Este resultado utiliza el concepto de corte elemental paralelo al de trayectoria elemental.

Un corte Q es elemental si al remover sus arcos de la red G, el número de componentes conexas se incrementa en una unidad. Si la red G es conexa, esto equivale a que Q tenga la forma $[S, N/S]$, donde $N \neq S \neq \emptyset$, todo par de nodos de S pueden ser unidos mediante una trayectoria que sólo utilice nodos de S y todo par de nodos en N/S pueden ser unidos con trayectorias cuyos nodos sean todos elementos de N/S.

Dada una coloración en una red y un arco blanco o negro de ella, se cumple que este arco pertenece a un circuito elemental compatible con la coloración o bien a un corte elemental compatible con la coloración. Este resultado se establece en el lema de Minty.

LEMA DE MINTY.

Dada una coloración en la red G con los colores verde, blanco, negro y rojo, y cualquier arco $\bar{J}$ blanco o negro, una y sola una de las siguientes afirmaciones es válida:

1. Existe un circuito (elemental) P compatible con la coloración que usa $\bar{J}$.
2. Existe un corte (elemental) Q compatible con la coloración que usa $\bar{J}$.

Este lema de Minty puede demostrarse a partir del teorema de la red coloreada del siguiente modo:

Sea $\bar{J} \sim (i, i')$ un arco, y denótense sus extremos con s y s' del siguiente modo: si $\bar{J}$ es blanco sean $s = i_2$ y $s' = i_1$; si $\bar{J}$ es negro, sean $s = i_1$ y $s' = i_2$. Por el algoritmo de enrutamiento para establecer una trayectoria $P: N^+ \longrightarrow N^-$ compatible con la coloración, en donde $N^+ = \{ s \}$ y $N^- = \{ s' \}$, sólo son factibles dos posibilidades:

a. No existe solución al problema de la trayectoria coloreada; esto es, se determina un corte $Q = [S, N/S]$ compatible con la coloración. En este caso, puesto que $s \in S$ y $s' \notin S$ se tiene $\bar{J} \in Q$ y por tanto se satisface la segunda condición del lema de Minty. Inversamente, si Q es un corte que satisface todas las condiciones anteriores resuelve el problema del corte coloreado.

b. Existe solución al problema de la trayectoria coloreada; en este caso se determina una trayectoria $P': s \longrightarrow s'$. Supóngase que esta trayectoria es elemental y que no usa $\bar{J}$.

La existençia de P' puede ser garantizada a partir del siguiente hecho: sólo existe una trayectoria elemental de N^+ a N^-, a saber: s, $\bar{J}$, s' ; pero esta trayectoria no es compatible con la coloración. Definase $P = P', \bar{J}, s$. Esta trayectoria es un circuito elemental que contiene a $\bar{J}$ y que es compatible con la coloración. Nótese que las trayectorias P' de esta clase corresponden biunívocamente a los circuitos P de esta clase. Por tanto se deriva la primera alternativa del lema de Minty.

Por otro lado, el teorema de la red coloreada puede ser demostrado a partir del lema de Minty de la siguiente manera:

Sea G una red coloreada con los cuatro colores de costumbre y sean N^+, N^- dos subconjuntos ajenos del conjunto N de nodos de la red. Sea G' una red construida agregando a G el arco $\bar{J} \sim (s',s)$ y los arcos (s,i), para todo $i \in N^+$, y (k,s'), para todo $k \in N^-$ (véase figura 4.4). Coloréense todos los nuevos arcos de blanco. La aplicación del lema de Minty para el arco $\bar{J}$ lleva a dos posibilidades:

1a. Existe un circuito elemental P compatible con la coloración que contiene al arco $\bar{J}$. Puesto que $\bar{J}$ es blanco, este circuito debe ser de la forma $P: s', \bar{J}, s, j_1, P', j_2, s'$ en donde $j_1 \sim (s,i)$, con $i \in N^+$, $j_2 \sim (k,s')$ con $k \in N^-$ y P' es entonces una trayectoria de i a k compatible con la coloración. Con esto se establece una correspondencia biunívoca de los circuitos P' con estas características y las trayectorias $P': i \longrightarrow k$, con $i \in N^+$ y $k \in N^-$, de donde se deriva la primera condición del teorema de la trayectoria coloreada.

2a. Existe un corte elemental Q compatible con la coloración que contiene el arco $\bar{J}$. En este caso Q no puede contener ningún arco nuevo distinto de $\bar{J}$, puesto que, siendo éstos blancos, no se satisfarían las restricciones de color (nótese que $\bar{J} \in Q^-$). De aquí que los demás arcos de Q son arcos de la red original G y por tanto constituyen un corte compatible con la coloración en ella. De nuevo esta correspondencia de cortes es biunívoca, por lo que se deriva la segunda alternativa del teorema de la red coloreada. □

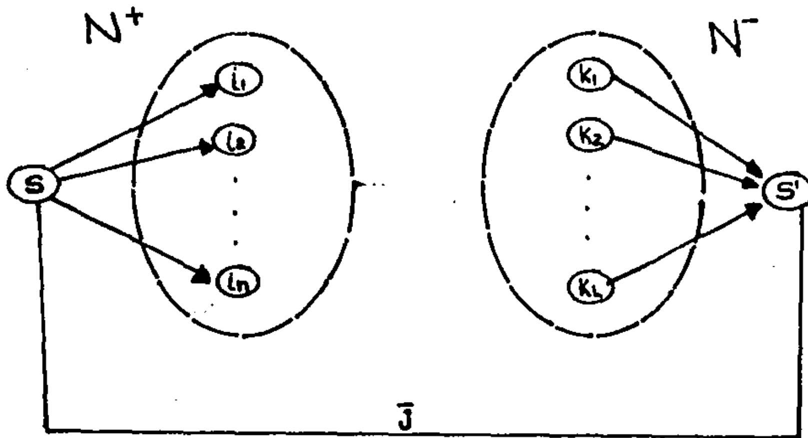


Figura 4.4

Capítulo V. Programas Desarrollados.

1. Conectividad de una Red.
2. Flujo Máximo.
3. Pintado de la Red en Flujo Máximo.

CONECTIVIDAD DE UNA RED.

En este algoritmo se usa la matriz de adyacencia X de la gráfica. Esta es la matriz de orden $n \times n$, donde n es el número de nodos de la gráfica, cuyas entradas son ceros o unos. Para la matriz cuadrada, X^2, las entradas corresponden al número de diferentes trayectorias entre dos nodos los cuáles usan dos arcos. Si la matriz es cúbica, X^3, entonces las entradas en la (i,j)-ésima posición es el número de diferentes trayectorias entre los nodos i & j los cuales usan tres arcos; asimismo, para la matriz X^4, sus entradas es el número de caminos entre dos nodos que usan cuatro arcos. Entonces se tiene muy bien determinada la matriz X^{n-1}, cuyas entradas en la (i,j)-ésima posición indica el número de diferentes trayectorias que existen entre los nodos i & j los cuáles usan n-1 arcos.

Si una gráfica está conectada, es posible encontrar una trayectoria entre toda pareja de nodos los cuáles usan a lo más n-1 arcos. Así al menos, una de las matrices X, X^2, ..., X^{n-1} debe tener una entrada diferente de cero en la posición (i,j)-ésima si existe una trayectoria entre los nodos i & j .

En resumen, para probar cuando una gráfica es conexa o no, la suma

$$X + X^2 + X^3 + \ldots + X^{n-1} = Y$$

es encontrada. Esta matriz deberá tener entradas cero para algunos nodos que no estén conectados, por lo que la gráfica es no-conexa; si la matriz es totalmente diferente de cero (para toda pareja de nodos i,j su entrada en la posición (i,j)-ésima de la matriz Y es diferente de cero), entonces la gráfica es conexa.

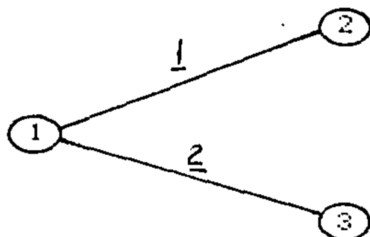


Figura 5.1

En la figura 5.1, se tiene que la matriz de adyacencia es

$$X = \begin{bmatrix} 0 & 1 & 1 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \end{bmatrix} \quad \& \quad X^2 = \begin{bmatrix} 2 & 0 & 0 \\ 0 & 1 & 1 \\ 0 & 1 & 1 \end{bmatrix}$$

Por lo tanto, la matriz que nos indica el número de trayectorias que existen para ir del nodo i al nodo j, que usan a lo más n-1 arcos, es:

$$Y = \begin{bmatrix} 2 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix} \qquad \text{donde} \quad Y = X + X^2$$

Por ejemplo, nótese que del nodo 1 al nodo 1 existen dos trayectorias:

i) Una que sigue esta secuencia: (1)—(2)—(1) (usa 2 arcos)

ii) Otra que usa esta secuencia: (1)—(3)—(1) (usa 2 arcos)

Así también, en este mismo ejemplo, existe una trayectoria que va del nodo i al nodo j, para todo $(i,j) \neq (1,1)$.

EJEMPLOS.

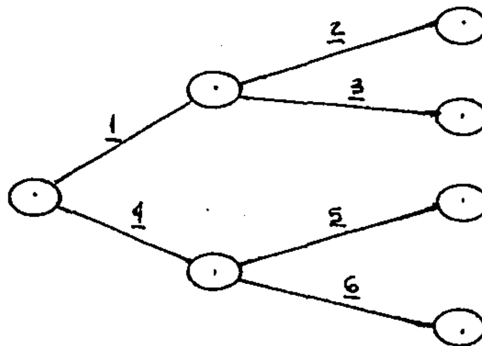


Figura 5.2 GRAFICA CONECTADA

$$Y = \begin{bmatrix} 42 & 21 & 21 & 21 & 21 & 21 & 21 \\ 21 & 49 & 35 & 14 & 14 & 7 & 7 \\ 21 & 35 & 49 & 7 & 7 & 14 & 14 \\ 21 & 14 & 7 & 14 & 14 & 7 & 7 \\ 21 & 14 & 7 & 14 & 14 & 7 & 7 \\ 21 & 7 & 14 & 7 & 7 & 14 & 14 \\ 21 & 7 & 14 & 7 & 7 & 14 & 14 \end{bmatrix}$$

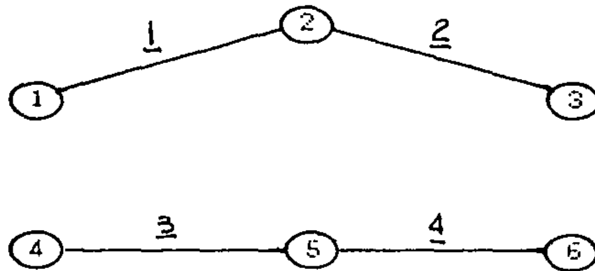


Figura 5.3 GRAFICA NO CONECTADA

$$Y = \begin{bmatrix} 3 & 7 & 3 & 0 & 0 & 0 \\ 7 & 6 & 7 & 0 & 0 & 0 \\ 3 & 7 & 3 & 0 & 0 & 0 \\ 0 & 0 & 0 & 3 & 7 & 3 \\ 0 & 0 & 0 & 7 & 6 & 7 \\ 0 & 0 & 0 & 3 & 7 & 3 \end{bmatrix}$$

donde $X + X^2 = Y$

FLUJO MAXIMO

Antes de formalizar, se comenta la idea central de este algoritmo, que es el algoritmo de Ford y Fulkerson. Se inicia con un flujo básico factible igual a cero. El paso siguiente consiste en encontrar trayectorias del nodo inicial S al nodo final F, y enviar el máximo flujo posible a través de ellas; este paso es iterativo y se interrumpe al no existir trayectoria alguna de S a F.

ALGORITMO DE FORD - FULKERSON

Paso 1. Iniciar con un flujo factible cero, i.e., $f_{ij} = 0$ $\forall$ (i,j) arco.

Paso 2. Encontrar una trayectoria de S a F.

- Se asigna la etiqueta [+F] al vértice S.
- Se etiqueta al nodo i cuyos arcos no estén saturados con [+S].
- Para cada nodo ya etiquetado se examinan:

  j si j no está etiquetado y el flujo de (i,j) no está saturado. Se le asigna la etiqueta [+i].

  k si k no tiene etiqueta y el flujo de (j,k) es mayor que cero. Se la asigna la etiqueta [-i].

Se repite el paso 2 hasta etiquetar F (ya encontramos una trayectoria de S a F). La etiqueta de F nos indica el nodo anterior a F en la trayectoria, y la de este el otro nodo.

Paso 3. (Aumento). Si F ha sido etiquetado se elige:

$$\alpha_i = \text{Min} \begin{cases} c_{ij} - f_{ij} & \text{si } j \text{ tiene etiqueta } [+i] \\ f_{ij} & \text{si } j \text{ tiene la etiqueta } [-i]. \end{cases}$$

y se pasa a (2).

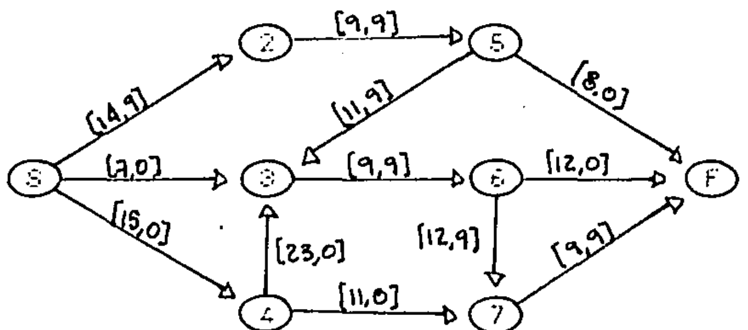
Si F no se puede etiquetar se termina el proceso y por consiguiente, el flujo máximo es:

$$\text{flujo Máximo} = \sum \alpha_i$$

Es de notar que este algoritmo encuentra soluciones óptimas para valores enteros, ya que por construccion del algoritmo, maneja un conjunto de valores enteros finito, y sus operaciones correspondientes.

EJEMPLO.

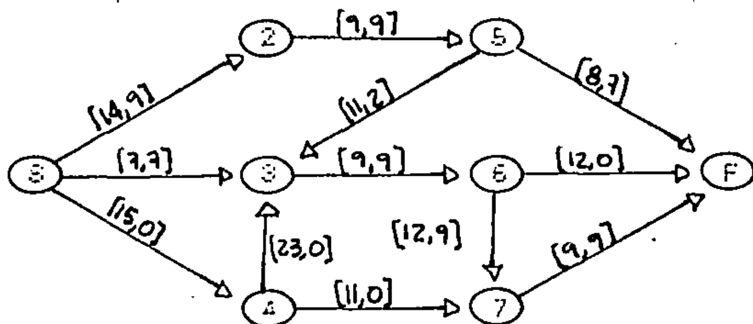
Sea la siguiente red, con sus capacidades y flujo inicial dado respectivamente.



$\alpha_1 = 9$

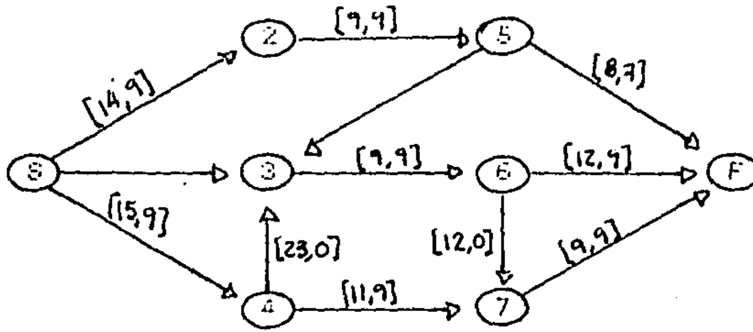
Figura 5.4

Solución:



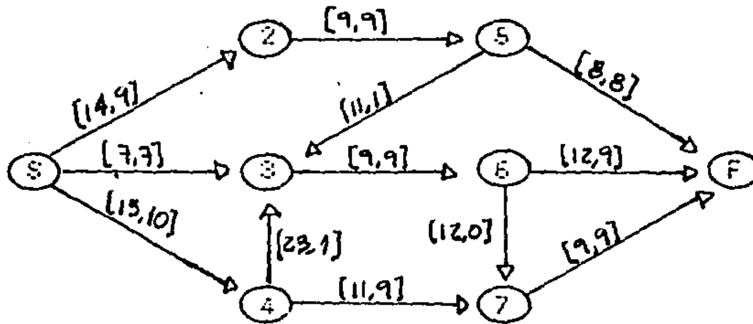
$\alpha_2 = 7$

Figura 5.4 a



$\alpha_3 = 9$

Figura 5.4 b



$\alpha_4 = 1$

Figura 5.4 c

Flujo Máximo = $\sum \alpha_i$ = 9 + 7 + 9 + 1 = 26.

PINTADO DE LA RED EN FLUJO MAXIMO.

Se tiene una red coloreada, cuyo pintado de los arcos significa lo siguiente:

V - verde - es posible enviar flujo por ambos sentidos del arco

B - blanco - es posible enviar flujo solamente en el sentido que tiene el arco

N - negro - es posible enviar flujo únicamente en sentido contrario que tiene el arco

R - rojo - es imposible enviar flujo por el arco.

De acuerdo a esta interpretación, se puede pensar a la red pintada de un sólo color, ya sea de color blanco ó de color negro. Por consiguiente, se elige, sin pérdida de generalidad, el blanco para pintar la red, es decir, se cambian los colores de la red original al color blanco de la manera siguiente:

1) Los arcos pintados de color V (verde) se descomponen en dos arcos de color blanco: uno con el sentido que tiene originalmente, y el otro con sentido inverso al original.

2) El arco pintado de color N (negro) se descompone en un arco blanco con sentido inverso al que tiene inicialmente.

3) El arco pintado de color B (blanco) se deja igual.

Nótese que todavía no existen arcos de color R (rojo); éstos van a existir cuando haya iniciado el proceso del Flujo Máximo.

Por ejemplo, sea la red G que se muestra abajo.

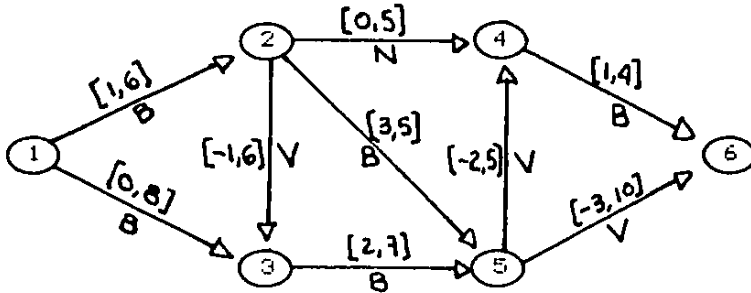


Figura 5.5

Los números que están sobre los arcos es el intervalo de capacidad.

Esta red es equivalente a la siguiente red G':

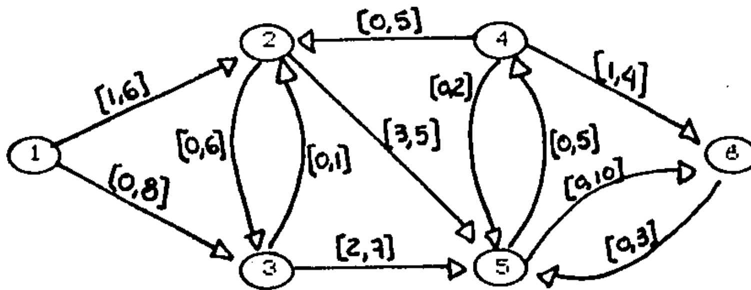


Figura 5.6

Nótese que ahora se tiene una red únicamente con arcos de color B (blanco).

Ahora bien, para cada arco existe un intervalo de capacidad $C(x) = [c^-(x), c^+(x)]$, y un flujo factible f(x). Supongamos que el flujo inicial de la red es factible, y que es posible aumentar ese flujo. Se recuerda que el objetivo es encontrar el flujo máximo de la red de un nodo fuente a un nodo destino.

Tómese la red del ejemplo anterior, y ahora se procede a dar un flujo factible, esto es, se añaden flujo f(x) a todos los arcos, donde $c^-(x) \leq f(x) \leq c^+(x)$. Supongamos que el flujo factible ya está dado, tal como se muestra en la figura. La idea es conocer el funcionamiento del algoritmo del pintado de la red para hallar el flujo máximo. Se hace hincapié en que todos los arcos de la red son de color blanco. Ver la figura 5.7

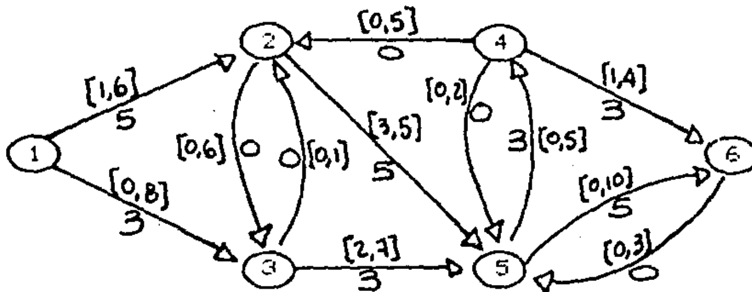
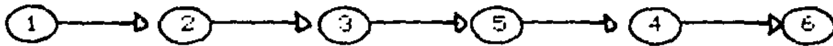


Figura 5.7

ITERACION 1.



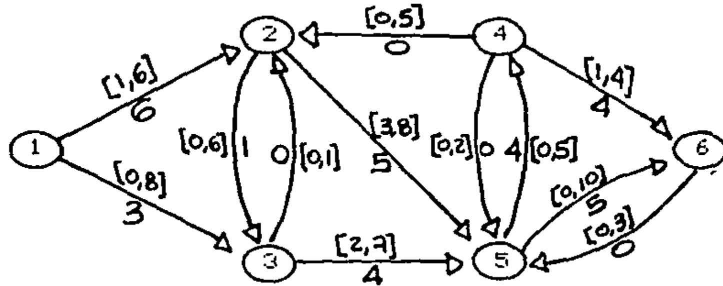
Para cada arco se obtiene la (Capacidad Máxima - Flujo). Después se elige el mínimo, es decir:

ARCO	CAPACIDAD MAXIMA		FLUJO		IGUAL
(1,2):	6	-	5	=	1
(2,3):	6	-	0	=	6
(3,5):	7	-	3	=	4
(5,4):	5	-	3	=	2
(4,6):	4	-	3	=	1

Por lo tanto, el Mínimo de $\{1,6,4,2,1\} = 1$. Así, el flujo aumentante es uno para la trayectoria anterior, es decir, $\alpha = 1$.

Ahora se tiene la nueva red dada a continuación, dónde a cada arco de la trayectoria anterior se le añadió α_1.

Figura 5.8



Como se puede observar en esta red, los arcos saturados (color rojo), son: (1,2) y (4,6). Esto es, $R_F = \{ (1,2), (4,6) \}$.

ITERACION 2.

y análogamente a la primera iteración, se tiene:

ARCO	CAPACIDAD MAXIMA		FLUJO		IGUAL
(1,3):	8	-	3	=	5
(3,2):	1	-	0	=	1
(2,5):	8	-	5	=	3
(5,6):	10	-	5	=	5

Por lo tanto, $\alpha_2 = \text{Min} \{ 5,1,3,5 \} = 1$. El flujo aumentante es 1. A cada arco de esta trayectoria se le añade una unidad a su flujo.

Se tiene ahora la red siguiente:

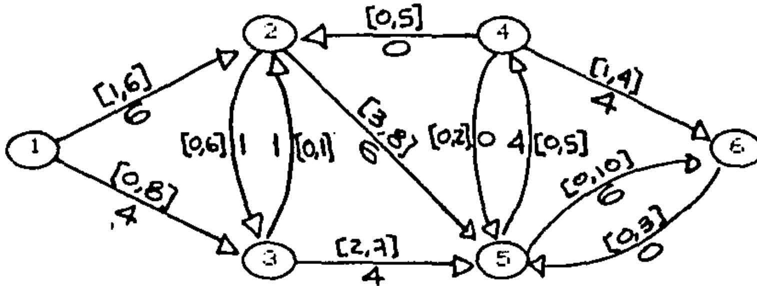
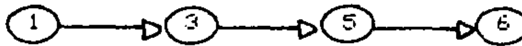


Figura 5.9

Se observa que se satura el arco (3,2), el cual se agrega al conjunto R_F , es decir, R_F = $\{ (1,2), (4,6), (3,2) \}$.

ITERACION 3.



Para esta trayectoria se tiene:

ARCO	CAPACIDAD MAXIMA		FLUJO		IGUAL
(1,3):	8	-	4	=	4
(3,5):	7	-	4	=	3
(5,6):	10	-	6	=	4

Se hace $\alpha_3 = \text{Min} \{ 4, 3, 4 \} = 3$. Por consiguiente, el flujo aumentante es 3, y a cada arco se le aumenta 3 unidades. Con este nuevo flujo, se tiene la red siguiente:

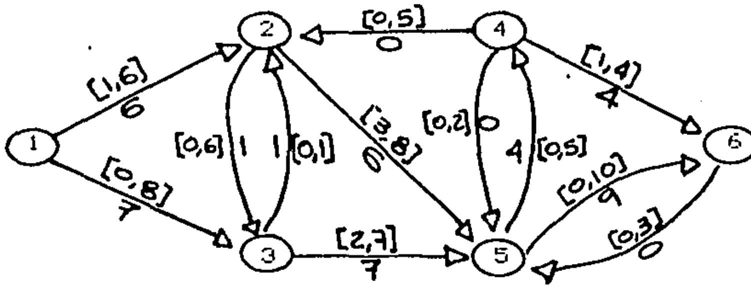


Figura 5.10

Se satura el arco (3,5). El conjunto de arcos rojos, R_F es ahora:

$$R_F = \{ (1,2),\ (4,6),\ (3,2),\ (3,5) \}$$

Nótese en la red que ya no es posible hallar una trayectoria del nodo 1 al nodo 6 . Se ha encontrado el flujo máximo. Este flujo máximo corresponde a:

$$\text{Flujo Máximo} = \text{Flujo Inicial en la Red} + \sum \alpha_i$$

$$\text{Flujo Máximo} = 8 + (1 + 3 + 1) = 13 .$$

CAPITULO VI. CONCLUSIONES

CONCLUSIONES

En el transcurso de su desarrollo las matemáticas han trascendido no sólo más allá de sus propias fronteras, sino también a través de los límites que separan las diferentes partes de la matemática. Antiguamente era posible distinguir, aunque con cierta dificultad, entre la matemática pura y la aplicada; hoy es imposible decir dónde comienza una y dónde termina la otra. Esta creciente imprecisión en las fronteras de las diferentes ciencias es una característica del desarrollo científico. Del mecanismo de una computadora al mecanismo del universo o al mecanismo de nuestros pensamientos hay sólo un pequeño paso.

En la presente década hubo un incremento en el uso de la computación; de hecho, sus aplicaciones siguen extendiéndose a campos jamás imaginables. La importancia de este trabajo reside en haber precisado el alcance que tiene el uso de los algoritmos, realzando sus bondades ya definidas, así como el de señalar la trascendencia que tiene la complejidad computacional como una herramienta útil para la eficiencia de los algoritmos.

En lo referente a la Estructura de Datos, se concluye que es esencial por estar íntimamente ligada a la eficiencia de los algoritmos, destacando su comodidad en el estudio de la teoría de redes, además de sus múltiples aplicaciones en los problemas reales.

Con respecto a las aplicaciones que se llevan a cabo en esta tesis, se destaca la implantación del algoritmo del pintado de la red para hallar el flujo máximo. Cabe señalar que existe mucha teoría sobre redes, sin embargo, la aportación en esta implantación en particular, fue la de crear un algoritmo basado en los colores de una red, y de allí iniciar una búsqueda exhaustiva de todas las trayectorias de un nodo fuente a un nodo destino, con un flujo inicial que puede ser diferente de cero, visualizándose tales caminos en la pantalla.

Otra aportación, fue la conjunción de los aspectos computación-redes, analizándose primeramente por separado, para después combinarlos en las aplicaciones ya descritas, con el enfoque deseado.

Asimismo, los algoritmos presentados pueden enriquecerse con nuevas ideas, como son para la conectividad y Flujo Máximo de Ford-Fulkerson, dibujar la red en pantalla, y para el algoritmo del pintado de la red, que tenga varios nodos fuente y varios nodos destino. Además de los muchos usos que se le pueden dar, ya que la creatividad del hombre es no medible.

Como complemento, se analizó la eficiencia de los algoritmos de Flujo Máximo (Ford-Fulkerson y Pintado de la Red) con base a su tiempo de ejecución. Se hizo una prueba de hipótesis para conocer si existía diferencia significativa entre las medias de los tiempos de estos algoritmos, estando ambos algoritmos en las mismas condiciones: misma computadora, mismos datos de entrada,

mismas redes, etc. Para realizar tal prueba se generaron números aleatorios los cuales construyeron las redes, así como sus características que tenían los arcos. Cabe mencionar que fueron generadas treinta redes para llevar a cabo este análisis. Y habiendo realizado tal prueba se llegó a la conclusión de que el algoritmo del Pintado de la Red es más eficiente con respecto al tiempo de ejecución.

Finalmente, hay que tener en cuenta que los objetivos establecidos de este trabajo se cumplieron, y que en lo personal, todavía queda mucho por hacer en estas áreas (Computación e Investigación de Operaciones) , ya que , como se observa en nuestro diario acontecer, ha habido un cambio interesante: en un mundo probabilístico, ya no manejamos ni cantidades ni afirmaciones relativas a un universo dado, real y específico, sino que hacemos preguntas que pueden hallar respuesta en un gran número de universos similares. Así, este trabajo espera haber contribuido con su granito de arena para el conocimiento del hombre.

COMENTARIOS:

Los programas fueron desarrollados en Turbo Pascal, versión 3.0 y fueron probados en computadora personal.

La Prueba de Hipótesis fue hecha con los programas de Flujo Máximo de Ford - Fulkerson y el Pintado de la Red, que por necesidades propias de la prueba desarrollada, estos programas se pasaron a Turbo Pascal, versión 4.0

Las corridas de la Prueba de Hipótesis fueron hechas en computadora personal 'printaform'.

APÉNDICE :

PRUEBA DE HIPÓTESIS ENTRE LA DIFERENCIA DE MEDIAS DE LOS ALGORITMOS DE FORD-FULKERSON Y EL PINTADO DE LA RED PARA FLUJO MÁXIMO.

ANEXOS :

PROGRAMA: CONECTIVIDAD DE UNA RED.
PROGRAMA: FLUJO MÁXIMO.
PROGRAMA: PINTADO DE LA RED.

COMPARACION ENTRE LOS ALGORITMOS DE FORD-FULKERSON Y EL PINTADO DE LA RED PARA FLUJO MAXIMO.

METODOLOGIA.

1. La construcción de las redes en las que se efectuó la comparación se llevó a cabo de la siguiente manera:

   a) Todas las redes poseen un nodo fuente y un nodo destino, además de sus nodos intermedios. El número de nodos de cada red tiene como máximo 50 nodos, y el número de arcos está entre (n-1) y 60, donde n es el número de nodos de la red.

2. Los datos para los algoritmos van a estar dados de la forma siguiente:

   a) Cada arco va a quedar definido dando su nodo inicial, su nodo final y su capacidad. La capacidad de cada arco puede ser mayor o igual a cero, y va a ser generada aleatoriamente.

3. Generación de números aleatorios.

   a) Se genera aleatoriamente el número de nodos de la que va a estar compuesta la red. Este número aleatorio está entre 2 y 50.

   b) Se genera aleatoriamente el número de nodos que van a estar conectados al nodo fuente. Este número está entre 1 y n-2, y va a estar en función del número de nodos n.

   c) Se genera aleatoriamente el número de nodos que van a estar conectados al nodo destino. De manera similar al nodo fuente, este número generado está en función de n, y se halla entre 1 y n - 2 .

d) Se genera aleatoriamente el número de arcos de que consta la red. Este número en general, se halla en el intervalo $(n-1, \frac{n(n-1)}{2})$ donde n es el número de nodos de la red.

Para nuestros fines, el número de arcos generado va a estar entre n-1 y 60.

e) La capacidad de cada arco es generada aleatoriamewnte.

No.CORRIDA	No.NODOS	No.ARCOS	FLUJO MAXIMO	D U R A C I O N	
				FORD FULKERSON	PINTADO DE LA RED
1	33	49	43	1.710	1.040
2	23	41	43	1.980	0.550
3	10	9	24	0.330	0.060
4	23	38	11	0.770	0.720
5	31	34	74	1.860	0.500
6	13	12	3	0.440	0.110
7	29	35	47	1.760	0.550
8	10	20	3	0.380	0.050
9	24	42	2	0.770	0.610
10	38	43	50	3.310	0.940
11	35	50	161	3.740	0.830
12	44	59	63	1.160	0.490
13	28	37	76	2.410	0.610
14	9	9	11	0.390	0.050
15	7	9	0	0.280	0.110
16	25	45	30	1.100	0.930
17	13	28	45	2.420	0.320
18	11	20	15	0.440	0.220
19	10	13	25	0.660	0.110
20	12	29	8	0.490	0.440
21	28	33	33	0.550	0.220
22	16	37	43	0.930	0.830
23	35	39	106	2.360	0.660
24	14	14	3	0.270	0.220
25	20	47	64	1.530	1.040
26	20	24	11	0.270	0.010
27	46	48	6	1.150	3.460
28	12	20	76	0.930	0.280
29	38	46	11	1.160	1.040
30	20	19	4	0.500	0.110

No.CORRIDA	No.NODOS	No.ARCOS	FLUJO MAXIMO	D U R A C I O N	
				FORD FULKERSON	PINTADO DE LA RED
1	33	49	43	1.710	1.040
2	23	41	43	1.980	0.550
3	10	9	24	0.330	0.060
4	23	38	11	0.770	0.720
5	31	34	74	1.860	0.500
6	13	12	3	0.440	0.110
7	29	35	47	1.760	0.550
8	10	20	3	0.380	0.050
9	24	42	2	0.770	0.610
10	38	43	50	3.310	0.940
11	35	50	161	3.740	0.830
12	44	59	63	1.160	0.490
13	28	37	76	2.410	0.610
14	9	9	11	0.390	0.050
15	7	9	0	0.280	0.110
16	25	45	30	1.100	0.930
17	13	28	45	2.420	0.320
18	11	20	15	0.440	0.220
19	10	13	25	0.660	0.110
20	12	29	8	0.490	0.440
21	28	33	33	0.550	0.220
22	16	37	43	0.930	0.830
23	35	39	106	2.360	0.660
24	14	14	3	0.270	0.220
25	20	47	64	1.530	1.040
26	20	24	11	0.270	0.010
27	46	48	6	1.150	3.460
28	12	20	76	0.930	0.280
29	38	46	11	1.160	1.040
30	20	19	4	0.500	0.110

TABLA 3. TIEMPOS DE EJECUCION EN CENTESIMAS DE SEGUNDO.

PRUEBA DE HIPOTESIS

DIFERENCIA ENTRE DOS MEDIAS.

A : Algoritmo de Ford - Fulkerson.

B : Algoritmo del Pintado de la Red para Flujo Maximo.

$$\bar{X}_A = 35.65/30 = 1.188$$
$$\bar{X}_B = 17.10/30 = 0.570$$

$S_A^2 = 0.809$ $S_A = 0.899$ $n_A = 30$

$S_B^2 = 0.399$ $S_B = 0.632$ $n_B = 30$

Hipotesis nula : $\mu_A = \mu_B$

Hipotesis alterna : $\mu_A < \mu_B$

$$\sigma_{\bar{x}_A - \bar{x}_B} = \sqrt{S_A^2/n_A + S_B^2/n_B} = \sqrt{(0.809)/30 + (0.399)/30}$$
$$= \sqrt{1.208/30} = \sqrt{0.0403} = 0.2007$$

$$Z = \frac{X_A - X_B}{x - x} = \frac{1.188 - 0.570}{0.2007} = \frac{0.618}{0.2007} = 3.079$$

Al nivel de significancia de 5 % el valor de zeta en pruebas de un extremo es de 1.645

Como $Z_c = 3.079 > 1.645$, se recchaza la hipotesis nula, es decir, no se acepta que los dos promedios sean iguales.

Al nivel de significancia del 1 % el valor critico de zeta es de 2.33 en pruebas de un extremo, y se tiene

$Z_c = 3.079 > 2.33$, por lo que tambien es rechazada la hipotesis nula.

Conclusion: Es mas eficiente el algoritmo del Pintado de la Red para encontrar el Flujo Maximo.

PROGRAMA CONECTIVIDAD DE UNA RED

```
PROGRAM CONEXA(INPUT,OUTPUT);
   CONST MAXNODOS = [illegible]0;
   TYPE
        MATRIX = ARRAY[1..MAXNODOS,1..MAXNODOS] OF INTEGER;
        VAR
           X,Y,Z : MATRIX;
           C,I,J,K,NODOS : INTEGER;
           CONNECT : BOOLEAN;

   PROCEDURE MATMULT(VAR A,B : MATRIX; N : INTEGER);

   VAR
      I,J,K : INTEGER;
      C     : MATRIX;
   BEGIN
      FOR I:= 1 TO N DO
      FOR K:= 1 TO N DO
      BEGIN
         C[I,K]:= 0;
         FOR J:= 1 TO N DO
            C[I,K]:= C[I,K] + A[I,J]*B[J,K]
      END;
      FOR I:= 1 TO N DO
      FOR J:= 1 TO N DO
      A[I,J] := C[I,J]
   END (MATMULT);

   PROCEDURE MATADD (VAR A,B: MATRIX;N : INTEGER);
   { EL  CONJUNTO  VA A SER LA SUMA DE A Y B }
   VAR
      I,J: INTEGER;
   BEGIN
      FOR I:= 1 TO N DO
      FOR J:= 1 TO N DO
         A[I,J]:= A[I,J] + B[I,J]
   END (MATADD);

   BEGIN
   (ALGORITMO DE LA CONECTIVIDAD USANDO LOS EXPONENTES DE LA
##    MATRIZ ADYACENTE
   VARIABLES USADAS:
   X  : LA MATRIZ DE ADYACENCIA
   Z  : EL ULTIMO EXPONENTE DE LA MATRIZ DE ADYACENCIA A SER
##  CALCULADA
   Y  : LA SUMA DE LOS EXPONENTES DE X
   C  : CONTADOR DEL NUMERO DE ARCOS
 NODOS: NUMERO DE NODOS EN LA GRAFICA
 I,J,K: VARIABLES LOOP
 CONNECT: VARIABLE QUE SE CONVIERTE EN CIERTA CUANDO LA GRAFICA
##    ES CONEXA )

 REPEAT
    WRITE('CUANTOS NODOS ?      ');
    READLN(NODOS);
    IF (NODOS <= 0) THEN
```

```
    WRITELN('DEBE HABER AL MENOS UN NODO, INTENTE OTRA VEZ')
UNTIL
    (NODOS > 0);
IF (NODOS > MAXNODOS) THEN
   BEGIN
     WRITELN('EL PROGRAMA HA SIDO PUESTO PARA REDES');
     WRITELN(' CON A  LO MAS', MAXNODOS:4);
     WRITELN('NODOS. SI DESEA USAR REDES GRANDES:');
     WRITELN('FAVOR DE CAMBIAR EL PARAMETRO MAXNODOS')
   END
   ELSE
   BEGIN
     FOR I:= 1 TO NODOS DO
     FOR J:= 1 TO NODOS DO
       X[I,J]:= 0;
     WRITELN('INTRODUZCA LOS ARCOS AHORA');
     WRITELN('DANDO SUS NODOS TERMINALES. DAR 0 0 PARA FINALIZAR');
     C:= 0;
   REPEAT
     C:= C + 1;
     REPEAT
       WRITE('ARCO NUMERO',C:2,':     ');
       READLN(I,J);
       IF ((I > NODOS) OR (J > NODOS)) THEN
       WRITELN('EL NODO ES MUY ALTO,INTENTE OTRA VEZ');
       IF (( I<0 ) OR (J < 0)) THEN
       WRITELN('EL NODO ES MUY BAJO,INTENTE OTRA VEZ');
     UNTIL
       ((I IN [0..NODOS]) AND (J IN [0..NODOS]));
       IF (I > 0 ) THEN
          BEGIN
             X[I,J]:= 1;
             X[J,I]:= 1;
          END
       UNTIL
          (I = 0 );
          WRITELN;
       IF (C < NODOS ) THEN
          BEGIN
             WRITELN('ESTA GRAFICA ES NO CONEXA');
             WRITELN('NO HAY BASTANTES ARCOS PARA');
             WRITELN('FORMAR EL ARBOL DE EXPANSION');
          END;
          {          *********
                     PASO    0
                     *********
          }
             FOR I:= 1 TO NODOS DO
             FOR J:= 1 TO NODOS DO
             BEGIN
                Z[I,J]:= X[I,J];
                Y[I,J]:= X[I,J]
             END;
             CONNECT:= FALSE;
             K:= 1;
```

```
{           *********
            PASO    1
            *********
}
   REPEAT
      I:= 0;
      REPEAT
      I:= I + 1;
      J:= 0;
      REPEAT
         J:= J + 1;
      UNTIL
         (( J >= NODOS ) OR ( Y[I,J] = 0 ));
      IF (( J = NODOS ) AND ( Y[I,NODOS] <> 0 )) THEN
         CONNECT := TRUE
      UNTIL (( I>=NODOS) OR CONNECT);
      IF NOT CONNECT THEN
         BEGIN
        {    *********
             PASO    2
             *********
      }
             K:= K + 1;
             IF (K < NODOS) THEN
               BEGIN
                   MATMULT(Z,X,NODOS);
                   MATADD(Y,Z,NODOS)
               END;
         END;
      UNTIL ((K = NODOS) OR (CONNECT));
      {     ***********
            PASO 3 Y 4
            ***********
      }
      IF CONNECT THEN
      BEGIN
         WRITELN('ESTA GRAFICA ES CONEXA');
         WRITELN('LA MATRIZ FINAL Y, ES: ');
         WRITELN;
        {    *********
             PASO    2
             *********
      }
      REPEET
             K:= K + 1;
             IF (K < NODOS) THEN
               BEGIN
                   MATMULT(Z,X,NODOS);
                   MATADD(Y,Z,NODOS)
               END;
       UNTIL (K = NODOS );
         FOR I:= 1 TO NODOS DO
         BEGIN
            FOR J:= 1 TO NODOS DO
            WRITE(Y[I,J]:6);
```

```
        WRITELN;
     END;
  END
  ELSE
  BEGIN
     WRITELN('ESTA GRAFICA NO ES CONEXA');
     WRITELN;
     WRITELN('LA MATRIZ FINAL Y ES: ');
     WRITELN;
     FOR I:= 1 TO NODOS DO
        BEGIN
           FOR J:= 1 TO NODOS DO
           WRITE(Y[I,J]:6);
           WRITELN;
        END;
  END;
END;
END.
```

PROGRAMA FLUJO MAXIMO (FORD-FULKERSON)

```
PROGRAM FLUJO_MAXIMO (INPUT,OUTPUT);
(**** COMPRUEBA EL FLUJO MAXIMO DE UNA RED ****)
CONST
     MAXNODOS     = 50;
TYPE NODEPTR      = 1..MAXNODOS;
     CAPFUNCT     = ARRAY[NODEPTR,NODEPTR] OF INTEGER;
     ARREGLOTIPO = ARRAY [NODEPTR] OF BOOLEAN;
     STR30        = STRING[30];
VAR
   NODOS      : 1..MAXNODOS;      { NODOS TOTALES DE LA RED }
   S,                             { NODO FUENTE }
   T          : 1..MAXNODOS;      { NODO DESTINO }
   I,J        : 1..MAXNODOS;      { VARIABLES AUXILIARES }
   CAPACIDAD : CAPFUNCT;          { ARREGLO QUE CONTIENE LA MAXIMA CAPACIDAD
                                    DE CADA UNO DE LOS ARCOS }
   FLUJO      : CAPFUNCT;         { ARREGLO QUE CONTIENE EL FLUJO QUE PASA
                                    POR CADA ARCO }
   FLUJOTOTAL: INTEGER;           { CONTIENE EL VALOR DEL FLUJO MAXIMO DE LA
                                    RED }
   ARCOS,K    : INTEGER;          { NUMERO DE ARCOS DE LA RED }
   IND        : INTEGER;          { INDICADOR PARA EL FLUJO FACTIBLE }
   RESP       : STR30;
   FLUJOTOT1 : INTEGER;

 { ************************************************************************ }

FUNCTION FLUJO_FACT (VAR RESP: STR30 ; VAR FLUJOTOT1 : INTEGER):BOOLEAN;

TYPE
     FLUJONODO = ARRAY[NODEPTR] OF INTEGER;
     FLUJOFACT = ARRAY[NODEPTR,NODEPTR] OF INTEGER;

VAR
   L,SUMA,
   FLUJOTOT,
   M,ARCFLUJO : INTEGER;
   FLUJOENT,
   FLUJOSAL    : FLUJONODO;
   FACTFLUJ    : FLUJOFACT;
   IND         : INTEGER;

BEGIN { PROCEDURE FLUJOFACTIBLE }
   WRITELN ('DESEAS INICIALIZAR UN FLUJO DIFERENTE DE CERO:  ');
   READLN (RESP);
      IF (RESP = 'SI')OR (RESP = 'si') or (RESP = 'Si') THEN
      BEGIN
          IND:=1;
          FOR I:= 1 TO NODOS DO
            FOR J:=1 TO NODOS DO
               FACTFLUJ[I,J]:= 0;
          WRITELN ('NUMERO DE ARCOS DEL FLUJO FACTIBLE:  ');
          READLN (ARCFLUJO);
          SUMA:=0;
            FOR L:= 1 TO ARCFLUJO DO
```

```
          BEGIN
             WRITELN ('ARCO FACTIBLE NUMERO: ',L);
             WRITE ('NODO INICIAL: ');
             READ (I);
             WRITE ('  NODO FINAL: ');
             READ (J);
             WRITE ('  FLUJO:   ');
             READLN (FACTFLUJ[I,J]);
                FACTFLUJ[I,J]:= FACTFLUJ[I,J];
                FLUJOTOT1:= SUMA + FACTFLUJ[I,J];
          END; { FIN DEL FOR }
       FOR M:=1 TO NODOS DO
          BEGIN
             FLUJOSAL[M]:= 0;
             FLUJOENT[M]:= 0;
          END;
       FOR M:=1 TO NODOS DO
          FOR J:=1 TO NODOS DO
             BEGIN
                FLUJOSAL[M]:=FLUJOSAL[M] + FACTFLUJ[M,J];
                FLUJOENT[M]:=FLUJOENT[M] + FACTFLUJ[J,M];
             END; { FIN DEL FOR }
             IF FLUJOENT[T] <> FLUJOSAL[S] THEN
             BEGIN
             WRITELN ('NO ES FLUJO FACTIBLE EN LOS NODOS FUENTE-DESTINO'
             IND:= 0;
             END;
             FOR M:=2 TO NODOS - 1 DO
                 IF FLUJOENT[M] <> FLUJOSAL[M] THEN
               BEGIN
                 WRITELN ('NO ES FACTIBLE EN EL NODO: ',M);
                 IND:= 0;
               END;
             IF IND = 1 THEN
              BEGIN
                WRITELN ('EL FLUJO INICIAL DADO ES FACTIBLE ');
                FLUJO_FACT:= TRUE;
              END
             ELSE
                FLUJO_FACT:=FALSE;
   END
   ELSE
      FLUJO_FACT := TRUE;
END; { FIN DEL PROCEDURE DE FLUJO FACTIBLE }

{ { ****************************************************************************** } }

PROCEDURE MAXFLUJO (CAP: CAPFUNCT;S,T: NODEPTR; VAR FLUJO: CAPFUNCT;
                    VAR FLUJOTOT:INTEGER);
TYPE
   ARREGLOTIPO = ARRAY[NODEPTR] OF BOOLEAN;
VAR
   PRED,NO,I: NODEPTR;
   X         : INTEGER;
   FLUJOTOT1: INTEGER;
```

```
    PRECEDE    : ARRAY[NODEPTR] OF NODEPTR;
    IMPROVE    : ARRAY[NODEPTR] OF INTEGER;
    FINCAM,
    AVANCE,
    ENCAM      : ARREGLOTIPO;
    RESP       : CHAR;

(****************************************************************************)

FUNCTION ANY (FINCAM : ARREGLOTIPO; NODOS : INTEGER) : BOOLEAN;
VAR
    I : INTEGER;
    FIN : BOOLEAN;
BEGIN
    FIN:= FALSE;
    I:= 1;
    REPEAT
        FIN := FIN OR FINCAM[I];
        I:= I + 1;
    UNTIL ( I > NODOS ) OR FIN;
    ANY:= FIN;
END; (* FIN DE LA FUNCION ANY *)

(****************************************************************************)

BEGIN { PROCEDURE MAXFLUJO }
    FOR NO:= 1 TO MAXNODOS DO
        FOR I:= 1 TO MAXNODOS DO
            FLUJO[NO,I]:=0;
    FLUJOTOT:=0;
  BEGIN
   IF ( RESP = 'SI' ) OR (RESP = 'si') OR (RESP = 'Si') THEN
    FLUJOTOT:=FLUJOTOT1;
  END;
    REPEAT

    { TRATAR DE ENCONTRAR UNA SEMIVIA DESDE S HASTA T }

        FOR NO:= 1 TO MAXNODOS DO
            BEGIN
                FINCAM[NO]:=FALSE;
                ENCAM[NO]:=FALSE;
            END { FOR...DO BEGIN };
        FINCAM[S]:=TRUE;
        ENCAM[S]:=TRUE;
        IMPROVE[S]:=MAXINT;

    { SE ASUME QUE S PUEDE PRODUCIR CUALQUIER FLUJO }

    WHILE (NOT ENCAM[T]) AND ANY(FINCAM,NODOS) DO
        BEGIN

            { TRATE DE EXTENDER UN CAMINO EXISTENTE }

            NO:=1;
```

```
        WHILE NOT FINCAM[NO] DO
            NO:=NO + 1;
            FINCAM[NO]:=FALSE;
        FOR I:= 1 TO MAXNODOS DO
            BEGIN
               IF (FLUJO[NO,I] < CAP[NO,I]) AND (NOT ENCAM[I]) THEN
                   BEGIN
                      ENCAM[I]:=TRUE;
                      FINCAM[I]:=TRUE;
                      PRECEDE[I]:=NO;
                      AVANCE[NO]:=TRUE;
                      X:=CAP[NO,I] - FLUJO[NO,I];
                          IF IMPROVE[NO] < X THEN
                             IMPROVE[I]:=IMPROVE[NO]
                          ELSE
                             IMPROVE[I]:=X
                   END; ( TERMINA EL BEGIN )
             IF (FLUJO[I,NO] > 0) AND  (NOT ENCAM[I]) THEN
                 BEGIN
                    ENCAM[I]:=TRUE;
                    FINCAM[I]:=TRUE;
                    PRECEDE[I]:=NO;
                    AVANCE[NO]:=FALSE;
                        IF IMPROVE[NO] < FLUJO[I,NO] THEN
                           IMPROVE[I]:=IMPROVE[NO]
                        ELSE
                           IMPROVE[I]:=FLUJO[I,NO]
                 END ( TERMINA EL BEGIN )
           END ( TERMINA FOR...DO BEGIN )
    END; ( TERMINA EL WHILE...DO BEGIN )
 IF ENCAM[T] THEN ( SE PUEDE AUMENTAR EL FLUJO EN LA SEMIVIA A T )
    BEGIN
        X:=IMPROVE[T];
        FLUJOTOT:=FLUJOTOT + X;
        NO:=T;
           WHILE NO <> S DO
              BEGIN ( REGRESE SOBRE LA VIA )
                 PRED:=PRECEDE[NO];
                    IF AVANCE[PRED] THEN
                    ( AUMENTE EL FLUJO DESDE PRED )
                       FLUJO[PRED,NO]:=FLUJO[PRED,NO]+X
                    ELSE
                    ( DISMINUYA EL FLUJO HASTA PRED )
                       FLUJO[NO,PRED]:=FLUJO[NO,PRED] - X;
                    NO:=PRED;
              END ( TERMINA EL WHILE...DO BEGIN )
    END ( TERMINA EL BEGIN )
  UNTIL NOT ENCAM[T]
END ( TERMINA EL PROCEDURE DE FLUJOMAX );
PROCEDURE MARIO;
BEGIN ( DE MARIO )
    WRITE ('DAME EL NUMERO TOTAL DE NODOS EN LA RED  ');
    READLN (NODOS);
    WRITE ('PROPORCIONA EL NODO FUENTE  ');
    READLN (S);
```

```
    WRITE ('PROPORCIONA EL NODO DESTINO   ');
    READLN (T);
        FOR I:= 1 TO NODOS DO
            FOR J:= 1 TO NODOS DO
            CAPACIDAD[I,J]:= 0;
            ARCOS:=0;
    WRITE ('NUMERO DE ARCOS=   ',ARCOS);
    READLN (ARCOS);
        FOR K:=1 TO ARCOS DO
            BEGIN
                WRITELN ('ARCO NUMERO:   ',K);
                WRITE ('NODO INICIAL:   ');
                READ (I);
                WRITE ('  NODO FINAL:   ');
                READ (J);
                WRITE ('  CAPACIDAD:   '):
                READLN (CAPACIDAD[I,J]);
                    CAPACIDAD[I,J]:= CAPACIDAD[I,J];
            END; {FIN DEL FOR }
    END; {DE MARIO}
BEGIN {PROGRAMA PRINCIPAL }
    MARIO;
        IF  FLUJO_FACT (RED,FLUJOTOTAL) THEN
          BEGIN
            MAXFLUJO (CAPACIDAD,S,T,FLUJO,FLUJOTOTAL);
            WRITELN ('EL FLUJO MAXIMO DE LA RED ES: ',FLUJOTOTAL);
          END;
END. { TERMINA EL PROGRAMA }
```

PROGRAMA PINTADO DE LA RED (FLUJO MAXIMO)

```
PROGRAM [illegible];
CONST
      MAXNODOS = 50;       { MAXIMO NUMERO DE NODOS }
      KNOD     = 60;       { MAXIMO NUMERO DE ARCOS }
      R = 0;
      V = 1;
      N = 2;
      B = 3;
      ENE = 10;
TYPE
      VELA     = ARRAY [1..KNODO] OF INTEGER;
      NODOTES  = ARRAY [1..MAXNODOS] OF BOOLEAN;
      CONJRUTA = SET OF 1..KNODO;
      CONJUNTO = SET OF 1..MAXNODOS;
      ARRECONJ = ARRAY [1..MAXNODOS,1..MAXNODOS] OF CONJRUTA;
      ARROJO   = ARRAY [1..KNODO] OF INTEGER;
      ARRUTA   = ARRAY [1..KNODO] OF CONJRUTA;
      AVECTOR  = ARRAY [1..MAXNODOS] OF INTEGER;
      NODEFMIR = 1..MAXNODOS;
      ARRCAR   = ARRAY [1..MAXNODOS,1..MAXNODOS] OF CHAR;
      ARRENT   = ARRAY [1..MAXNODOS,1..MAXNODOS] OF INTEGER;
      ARRBOL   = ARRAY [1..MAXNODOS,1..MAXNODOS] OF BOOLEAN;
      FCONJ    = ARRAY [1..MAXNODOS] OF CONJUNTO;
      REGISTRO = RECORD
                   INIC,
                   FINAL,
                   COLOR,
                   CAPMA,
                   CAPMI,
                   FLUJO    : INTEGER;
                 END;
VAR
      VEL                 : VELA;
      NODO                : NODOTES;
      CONJ                : FCONJ;
      ENTEROJO,ROJO       : ARROJO;
      CONJROJO            : ARRUTA;
      CONJFMAX,
      MARIO,ACONJ         : CONJRUTA;
      ARCONJ              : ARRECONJ;
      MATCOLOR,
      MATCOLGRA,
      MATCAPMA,
      MATCAPMI,
      FMATFLU,
      MATFLU              : ARRENT;
      USADO,
      MATBOOL             : ARRBOL;
      VECFLU,X,A,Y,
      ESC,NODOESC         : ARROJO;
      VECTOR,
      FLUJOSAL,
      FLUJOENT            : AVECTOR;
      FN,NO,NODOS         : 1..MAXNODOS;
      GRACONJ             : CONJUNTO;
```

```
     S,T,VF,
     RO,NO,L,INO,
     MINIMO,Z,NU,
     FLUJOMAX,
     NUMCORTE,ACORTE,
     NODOCORTE,NC,AO,
     ARCOS,I,J,AX,BX,C,D,
     MAXIMO,RJ,FMAXCOR,
     NUMARCOS,X0,X1,
     Y0,Y1,COLOR,
     K,M,NUMNODO,
     XF0,XF1,YF0,YF1,
     ABC,BCD              : INTEGER;
     MO,BO,O1,
     ALFA,BETA,
     GAMA,ETA,
     IOTA,KAPA            : REAL;
     REG                  : REGISTRO;
     BJ,ACORT             : ARRAY[1..MAXNODOS,1..2] OF INTEGER;
     BOOLMAX              : ARRAY[1..MAXNODOS,1..1] OF BOOLEAN;
     RUTABB,LISTO,
     RUTABOOL,ROJITO,
     CORRECTO             : BOOLEAN;
     CAR                  : CHAR;

     (*I B:GRAPH.P)

PROCEDURE MAXFLUJO;
BEGIN { MAXFLUJO }
     RUTABOOL:= FALSE;
     ROJITO:= FALSE;
     Z:= 0;
     VF:= 0;
     RJ:= 0;
     FLUJOMAX:= 0;
     K:=0;
     J:=1;
     I:=2;
     A[1]:=1;
REPEAT
     RUTABB:= TRUE;

   { COMIENZA A ELEGIR NODOS EN LA RUTA DEL NODO FUENTE }

   IF ( (I IN CONJ[1]) AND (MATCOLOR[1,I] = B) AND
   (MATBOOL[1,I] = FALSE) AND (USADO[1,I] = FALSE)) THEN
      BEGIN
        J:= J + 1;
        A[J]:= I;
        K:= K + 1;
        FMATFLU[1,I]:= MATCAPMA[1,I] - MATFLU[1,I];
        VECTOR[1]:= FMATFLU[1,I];
        IF I = T THEN
        BEGIN
           RUTABOOL:= TRUE; { EXISTE AL MENOS UNA TRAYECTORIA }
```

```
MATFLU[I,I]:= MATFLU[I,I] + FMATF[I,I] ;
FLUJOMAX:= FLUJOMAX + MATFLU[1,I];
RJ:= RJ + 1;
VF:= VF + 1;
WRITELN;
WRITELN ('ITERACION ',VF,': ');
VFL[VF]:= MATFLU[1,I];
MATCOLOR[1,I]:= R;
ROJO[RJ]:= MATCAPMA[1,I];
CONJROJO[RJ]:= ARCONJ[A[1],I];
BU[RJ,1]:= A[1];
BU[RJ,2]:= I;
WRITELN ('UNA RUTA ES: [1,',',I,']','CON COLOR ',MATCOLOR[1,I]);
READ (KBD,CAR);
RUTAS8:= FALSE;
L:=1;
A[L]:= 1;
A[L+1]:= NODOS;
ROJITO:= TRUE;
IF ROJITO = TRUE THEN
{ EMPIEZA EL PROCEDIMIENTO DE CAMBIO ROJO }
BEGIN { CAMBIO ROJO }
    COLOR:= 1;
    XO:= X[ESC[A[L]]];
    YO:= Y[ESC[A[L]]];
    X1:= X[ESC[A[L+1]]];
    Y1:= Y[ESC[A[L+1]]];
    MO:= (Y1 - YO)/(X1 - XO);
    BO:= (-MO)*XO + YO;
    D1:= (X1 - XO)/NNE;
    FOR K:= 0 TO 10 DO
    { SE DESEAN 10 SEGMENTOS DE ESA RECTA }
    BEGIN
        XFO:= ROUND(XO + K * D1)     ; YFO:= ROUND(MO * XFO + BO);
        XF1:= ROUND(XO + (K+1)*D1)  ; YF1:= ROUND(MO * XF1 + BO);
        IF K MOD 2 = 0 THEN
           DRAW(XFO,YFO,XF1,YF1,MATCOLOR[A[L],A[L+1]]);
        IF K MOD 2 = 1 THEN
           DRAW(XFO,YFO,XF1,YF1,R00);
    END; { FIN DEL FOR K }
    READ (KBD,CAR);
END;     { FIN DEL PROCEDURE CAMBIO_ROJO }
END
ELSE
BEGIN
    RU:=2;
    CREACONJ:= [1];
    CREACONJ:= CREACONJ + [I];
    NR:= 1;
    REPEAT
    IF (RU IN CONJ[NR]) AND (MATCOLOR[NR,RU] = B ) AND
    (MATBOOL[NR,RU] = FALSE) AND NOT(RU IN CREACONJ)
    AND (USADO[NR,RU] = FALSE) THEN
    BEGIN
       CREACONJ:= CREACONJ + [RU];
```

```
J:= J + 1;
A[J]:= RO;
K:= K + 1;
NUMARCOS:= K;
FMATFLU[A[J-1],A[J]]:= MATCAPMA[A[J-1],A[J]]
                       - MATFLU[A[J-1],A[J]];
VECTOR[K]:= FMATFLU[A[J-1],A[J]];
IF  RO = 1 THEN
BEGIN
   RUTABOOL:= TRUE; { EXISTE AL MENOS UNA TRAYECTORIA }
   MINIMO:= VECTOR[1];
   FOR I:= 2 TO K  DO
   BEGIN
      IF (VECTOR[I] < MINIMO ) THEN
          MINIMO:= VECTOR[I];
   END;
   VF:= VF + 1;
   WRITELN;
   WRITELN ('ITERACION ',VF,' : ');
   WRITELN ('UNA RUTA ES: ');
   FOR L:= 1 TO NUMARCOS DO
   BEGIN
      MATFLU[A[L],A[L+1]]:= MATFLU[A[L],A[L+1]] +
                            MINIMO;
      IF L = NUMARCOS then
         IF BOOLMAX[A[L],1] = FALSE THEN
         BEGIN
            FLUJOMAX:= FLUJOMAX + MATFLU[A[L],A[L+1]];
            BOOLMAX[A[L],1]:= TRUE;
         END
         ELSE
         FLUJOMAX:= FLUJOMAX + MINIMO;
         ARCONJ[A[L],A[L+1]]:= ARCONJ[A[L],A[L+1]] + [VF];
      IF MATCAPMA[A[L],A[L+1]] = MATFLU[A[L],A[L+1]] THEN
      BEGIN
         RJ:= RJ + 1;
         CONJROJO[RJ]:= ARCONJ[A[L],A[L+1]];
         BU[RJ,1]:= A[L];
         BU[RJ,2]:= A[L+1];
         MATCOLOR[A[L],A[L+1]]:= R;
         ROJO[RJ]:= MATCAPMA[A[L],A[L+1]];
         WRITE('CAMBIA A ROJO  ');
         ROJITO:= TRUE;
         IF ROJITO = TRUE THEN
         {  EL PROCEDIMIENTO DE CAMBIO ROJO }
         BEGIN { CAMBIO_ROJO }
            COLOR:= 1;
            X0:= X[ESC[A[L]]];
            Y0:= Y[ESC[A[L]]];
            X1:= X[ESC[A[L+1]]];
            Y1:= Y[ESC[A[L+1]]];
            M0:= (Y1 - Y0)/(X1 - X0);
            B0:= (-M0)*X0 + Y0;
            D1:= (X1 - X0)/ENE;
            FOR K:= 0 TO 10 DO
```

```
                        ( SE DESEAN 10 SEGMENTOS DE ESA RECTA )
                          BEGIN
                             XF0:= ROUND(X0 + K * D1);
                             YF0:= ROUND(M0 * XF0 + B0);
                             XF1:= ROUND(X0 + ((K+1)*D1)) ;
                             YF1:= ROUND(M0 * XF1 + B0);
                             IF K MOD 2 = 0 THEN
                             DRAW(XF0,YF0,XF1,YF1,MATCOLOR[A[L],A[L+1]]);
                             IF K MOD 2 = 1 THEN
                                DRAW(XF0,YF0,XF1,YF1,RED);
                          END; ( FIN DEL FOR K )
                          READ (KBD,CAR);
                       END;
                    END;
                  ( FIN DEL PROCEDIMIENTO CAMBIA_ROJO )
                   WRITELN('EL ARCO ',L,' ES: [ ',A[L],', ',A[L+1],' ]')
                END;
                WRITELN ('EL FLUJO AUMENTANTE ES: ', MINIMO);
                RU:= RU - 1;
                RUTA88:=FALSE;
                READ(KBD,CAR);
             END
             ELSE
             BEGIN
                NR:=RU;
                RU:= 1;
             END;
          END;
          RU:= RU + 1;
       UNTIL (RU > NODOS) OR (RUTA88=FALSE);

       IF (RU > NODOS) THEN

       BEGIN
          MATBOOL[A[J-1],A[J]]:= TRUE;
          RUTA88:= FALSE;
       END;
    END;
    IF RUTA88 = FALSE THEN
    BEGIN
       K:= 0;
       I:= 1;
       J:= 1;
       A[1]:= 1;
    END;
 END;
      I:= I + 1;
UNTIL  I > NODOS;

END;
{ fin del procedimiento maxflujo }

PROCEDURE CORTEMINIMO;
BEGIN
   CORRECTO:= FALSE;
```

```
I:=1;                                  ACORTE:= 1;
RCORTE[1,1]:= BU[1,1];                 RCORTE[1,2]:= BU[1,2];
MARIO:= CONJROJO[1];
ACONJ:= [ ];
FOR K:= 1 TO VF DO
ACONJ:= ACONJ + [K];                 { ACONJ = [1,2,...VF] }
REPEAT
    FOR K:= 1 TO RJ DO
    BEGIN
        IF CONJROJO[K] * MARIO = [ ] THEN
        { no existe un elemento en comun }
        BEGIN
            ACORTE:= ACORTE + 1;
            RCORTE[ACORTE,1]:= BU[K,1];
            RCORTE[ACORTE,2]:= BU[K,2];
            MARIO:= MARIO + CONJROJO[K];
        END;
    END;
    IF MARIO = ACONJ THEN
    { se ha llegado el conjunto del corte minimo }
    CORRECTO:= TRUE
    ELSE
    BEGIN  { aun no se tiene el corte minimo }
        I:= I + 1;
        MARIO:= CONJROJO[I];                    ACORTE:= 1;
        RCORTE[1,1]:= BU[I,1];                  RCORTE[1,2]:= BU[I,2];
    END;
UNTIL CORRECTO = TRUE;
WRITELN ('EL CORTE MINIMO ES:   ');
FOR Z:=1 TO ACORTE DO
BEGIN
    { GOTOXY(2,37);   }
    WRITE ('[',RCORTE[Z,1],',',RCORTE[Z,2],']');
    X0:= X[ESC[BU[Z,1]]];                 Y0:= Y[ESC[BU[Z,1]]];
    X1:= X[ESC[BU[Z,2]]];                 Y1:= Y[ESC[BU[Z,2]]];
    IF (X1 - X0) = 0 then
    m0:= 999
    else
   { WRITE('X1 - X0 ES: ',K); }
    M0:= (Y1 - Y0)/(X1 - X0);
    B0:= (-M0)*X0 + Y0;
    D1:= (X1 - X0)/ENE;
    FOR K:= 0 TO 10 DO
    { se desean 10 segmentos de esa recta }
    BEGIN
        XF0:= ROUND(X0 + K*D1);                 YF0:= ROUND(M0 * XF0 + B0);
        XF1:= ROUND(X0 + (K + 1)*D1);           YF1:= ROUND(M0 * XF1 + B0);
        IF K MOD 3 = 0 THEN
        DRAW(XF0,YF0,XF1,YF1,1);
        IF K MOD 3 = 1 THEN
        DRAW(XF0,YF0,XF1,YF1,2);
        IF K MOD 3 = 2 THEN
        DRAW(XF0,YF0,XF1,YF1,3);
    END;
END;
```

```
END; {  fin del procedure cortemínimo }

PROCEDURE  MENSAJE;
BEGIN { mensaje }
    GRAPHCOLORMODE;
    GRAPHBACKGROUND(BLACK);
    GOTOXY(1,1);
    WRITELN ('EL PRESENTE ALGORITMO ENCUENTRA');
    WRITELN ('EL FLUJO MAXIMO DE UNA RED.');
    WRITELN ('SIEMPRE HAY QUE DAR UN FLUJO INICIAL');
    WRITELN ('FACTIBLE EN EL ALGORITMO');
    WRITELN ('EL NODO INICIAL SIEMPRE ES UNO');
    WRITELN ('EL NODO FINAL SIEMPRE');
    WRITELN ('ES EL ULTIMO NUMERO DE NODO');
    WRITELN ('Oprime una tecla para continuar');
    READ (KBD,CAR);
END; { MENSAJE }


PROCEDURE PINTAFLUJO;
BEGIN {PINTAFLUJO}
    CLRSCR;
    WRITELN('DAME EL NUMERO TOTAL DE NODOS');
    WRITELN('EN LA RED: ');
    READLN(NODOS);
    WRITELN('PROPORCIONA EL NODO FUENTE: ');
    READLN(S);
    WRITELN('PROPORCIONA EL NODO DESTINO: ');
    READLN(T);
    FOR J:= 1 TO NODOS - 1 DO
        BOOLMAX[A[J],T]:= FALSE;
    FOR I:= 1 TO NODOS DO
        BEGIN
            CONJ[I]:= [ ];
            BEGIN
                FOR J:= 1 TO NODOS DO
                BEGIN
                    MATCAPMA[I,J]:= 0;
                    MATCAPMI[I,J]:= 0;
                    MATFLU[I,J]:= 0;
                    MATBOOL[I,J]:= FALSE;
                    USADO[I,J]:= FALSE;
                    ARCONJ[I,J]:= [ ];
                END; {FIN DEL FOR J}
            END;
        END; {FIN DEL FOR I}
    WRITE('PROPORCIONA EL NUMERO DE ARCOS: ');
    READLN(ARCOS);
    WRITELN('ESCRIBE LOS REGISTROS EN ESTE ORDEN: ');
    WRITELN('NODO INICIAL NODO FINAL COLOR CAPMAXIMA CAPMINIMA FLUJO ');
    FOR M:= 1 TO ARCOS DO
     BEGIN
        WRITELN('REGISTRO NUMERO: ',M);
        READLN(REG.INIC,REG.FINAL,REG.COLOR,REG.CAPMA,REG.CAPMI,REG.FLUJO);
        IF (REG.FLUJO > REG.CAPMA) OR (REG.FLUJO < REG.CAPMI) THEN
```

```
      BEGIN
          WRITELN(' NO ES FLUJO FACTIBLE ');
          IND:= 0 ( ESTE INDICADOR NOS SIRVE PARA QUE EL PROGRAMA NO HAGA NADA
      END;
      MATCOLGRA[REG.INIC,REG.FINAL]:= REG.COLOR;
      IF REG.COLOR = V THEN
       BEGIN
         MATCOLOR[REG.INIC,REG.FINAL]:= B;
         MATCOLOR[REG.FINAL,REG.INIC]:= B;
         MATCAPMA[REG.INIC,REG.FINAL]:= REG.CAPMA;
         MATCAPMA[REG.FINAL,REG.INIC]:= -(REG.CAPMI);
         MATCAPMI[REG.INIC,REG.FINAL]:= 0;
         MATCAPMI[REG.FINAL,REG.INIC]:= 0;
         CONJ[REG.INIC]:= CONJ[REG.INIC] + [REG.FINAL];
         CONJ[REG.FINAL]:= CONJ[REG.FINAL] + [REG.INIC];
         IF REG.FLUJO >= 0 THEN
                BEGIN
                  MATFLU[REG.INIC,REG.FINAL]:= REG.FLUJO;
                  MATFLU[REG.FINAL,REG.INIC]:= 0;
                END
                ELSE
                BEGIN
                  MATFLU[REG.INIC,REG.FINAL]:= 0;
                  MATFLU[REG.FINAL,REG.INIC]:= -(REG.FLUJO);
                END; (FIN DEL IF)
       END;
          IF REG.COLOR = N THEN
          BEGIN
            MATCOLOR[REG.FINAL,REG.INIC]:= B;
            MATCAPMA[REG.FINAL,REG.INIC]:= REG.CAPMA;
            MATCAPMI[REG.FINAL,REG.INIC]:= REG.CAPMI;
            MATFLU[REG.FINAL,REG.INIC]:= REG.FLUJO;
            CONJ[REG.FINAL]:= CONJ[REG.FINAL] + [REG.INIC];
          END;
          IF REG.COLOR = B THEN
          BEGIN (EL ARCO ES DE COLOR BLANCO)
            MATCOLOR[REG.INIC,REG.FINAL]:= B;
            MATCAPMA[REG.INIC,REG.FINAL]:= REG.CAPMA;
            MATCAPMI[REG.INIC,REG.FINAL]:= REG.CAPMI;
            MATFLU[REG.INIC,REG.FINAL]:= REG.FLUJO;
            CONJ[REG.INIC]:= CONJ[REG.INIC] + [REG.FINAL];
          END;
    END; (  FIN DEL FOR DE ARCOS )
END; ( FIN DEL PROCEDURE PINTAFLUJO )


PROCEDURE GRAFICA;
BEGIN  ( GRAFICA )
   GRAPHWINDOW(3,0,310,150);
   GRAPHCOLORMODE;
   GRAPHBACKGROUND(BLACK);
   PALETTE(3);
   COLOR:= 1;
   GOTOXY(2,30);
   DRAW(3,20,310,20,1);
```

```
DRAW(3,185,310,185,1);
DRAW(3,20,3,185,1);
DRAW(310,20,310,185,1);
X[1]:= 10;
Y[1]:= 99;
NODOESC[1]:= 1;
ESC[1]:= 1;
CIRCLE(X[1],Y[1],7,1);
GOTOXY(X[1] DIV 8,Y[1] DIV 8);
WRITE('1');
X[NODOS]:= 300;
Y[NODOS]:= 99;
NODOESC[NODOS]:= NODOS;
ESC[NODOS]:= NODOS;
CIRCLE(X[NODOS],Y[NODOS],7,1);
GOTOXY(X[NODOS] DIV 8,Y[NODOS] DIV 8);
WRITE(NODOS);
READ (KBD,CAR);
IF FN = 1 THEN
BEGIN
    X[2]:= 40;
    Y[2]:= 38;
    CIRCLE(X[2],Y[2],7,1);
END;
IF FN > 1 THEN
BEGIN  ( LOS NODOS CONECTADOS CON LA FUENTE )
    ABC:= 150 DIV (FN-1);
   ( ES LA SEPARACION ENTRE CADA NODO )
   FOR I:= 2 TO FN + 1 DO
   BEGIN
     ALFA:= 105 + (I - 2) * ABC;
     BETA:= (ALFA * 3.1416)/180;
     X[I]:= ROUND(130 + 70 *  COS (BETA)) ;
     Y[I]:= ROUND(99  + 70 *  SIN (BETA)) ;
     CIRCLE(X[I],Y[I],7,1);
   END;
   READ (KBD,CAR);
END;

( SIGUEN LOS ARCOS QUE VAN A IR AL FINAL )

IF ND = 1 THEN
BEGIN
    X[NODOS-1]:= 270;
    Y[NODOS-1]:= 172;
    CIRCLE(X[NODOS-1],Y[NODOS-1],7,1);
END;
IF ND > 1 THEN
BEGIN
    BCD:= 150 DIV (ND - 1);
    FOR J:= (NODOS - ND) TO (NODOS - 1) DO
    BEGIN
        GAMA:= 285 + (J - (NODOS - ND)) * BCD;
        ETA := (GAMA * 3.1416)/180;
        X[J]:= ROUND(185 + 70 * COS (ETA));
```

```
        Y[J]:= ROUND( 99 + 70 * SIN (ETA));
        CIRCLE(X[J],Y[J],7,1);
      END;
      READ (KBD,CAR);
    END;

{ SIGUEN LOS NODOS QUE VAN A IR ENMEDIO }

    C:= NODOS - (FN + NO+ 2);
    IF C > 0 THEN
      IF C = 1 THEN
        BEGIN
          X[FN + 2]:= 160;
          Y[FN + 2]:=  99;
          CIRCLE(X[K],Y[K],7,1);
        END
      ELSE
      BEGIN
        D:= (360) DIV C ;
        FOR K:= (FN + 2) TO (NODOS - (1 + NO)) DO
        BEGIN
          IOTA:= 45 + (K - (FN + 2)) * D;
          KAPA:= (IOTA * 3.1416)/180;
          X[K]:= ROUND(159 + 60 * COS (KAPA));
          Y[K]:= ROUND( 99 + 60 * SIN (KAPA));
          CIRCLE(X[K],Y[K],7,1);
        END;
      END;
      READ (KBD,CAR);
    { ESTO FUE SOLAMENTE PARA LA POSICION DE LOS NODOS }
END; { FIN DEL PROCEDURE GRAFICA }

    { AHORA VIENE LA TRAZA DE LOS ARCOS }

PROCEDURE TRAZA;
BEGIN {PROCEDURE TRAZA }
    FOR J:= 1 TO NODOS DO
    BEGIN
      FOR I:= 1 TO NODOS DO
      BEGIN
        IF I IN CONJ[J] THEN
        BEGIN
          DRAW(X[ESC[J]],Y[ESC[J]],X[ESC[I]],Y[ESC[I]],MATCOLGRA[J,I]);
          ARC(X[ESC[I]],Y[ESC[I]],20,10,MATCOLGRA[J,I]);
        END;
        IF J IN CONJ[I] THEN
        BEGIN
          DRAW(X[ESC[I]],Y[ESC[I]],X[ESC[J]],Y[ESC[J]],MATCOLGRA[I,J]);
          ARC(X[ESC[J]],Y[ESC[J]],20,10,MATCOLGRA[I,J]);
        END;
      END; { FIN DEL FOR I }
    END; { FIN DEL FOR J }
    READ (KBD,CAR);
END;    { FIN DEL PROCEDURE TRAZA }
```

```
PROCEDURE ESCRIBE;
    { ES LA ESCRITURA DE LOS NODOS.}
BEGIN
    IF FN > 0 THEN
    BEGIN
        FOR K:= 2 TO FN + 1 DO
        BEGIN
            GOTOXY(X[K] DIV 8,Y[K] DIV 8);
            WRITE(NODOESC[K]);
        END;
    END; { FIN DEL IF FN }
    IF ND > 0 THEN
    BEGIN
        FOR K:= (NODOS - ND) TO (NODOS - 1) DO
        BEGIN
            GOTOXY(X[K] DIV 8, Y[K] DIV 8);
            WRITE(NODOESC[K]);
        END;
    END; { FIN DEL IF ND }
    IF (NODOS - (FN + ND + 2)) > 0 THEN
    BEGIN
        FOR K:= (FN + 2) TO (NODOS - (1 + ND)) DO
        BEGIN
            GOTOXY(X[K] DIV 8, Y[K] DIV 8);
            WRITE(NODOESC[K]);
        END;
    END; { FIN DEL IF NODOS - (FN + ND + 2) }
END; { FIN DEL PROCEDURE ESCRIBE }

{ INICIA EL PROCEDIMIENTO DE LA EJECUCION DE LA GRAFICA DE LA RED }

PROCEDURE EJECUTA;
BEGIN { EJECUTA }
    CLRSCR;
    FOR I:= 1 TO NODOS DO
    BEGIN
        NODO[I]:= FALSE;
    END;
    WRITELN('NUMERO DE NODOS CONECTADOS');
    WRITELN('CON EL NODO FUENTE: ');
    READLN(FN);
    WRITELN ('NUMERO DE NODOS CONECTADOS');
    WRITELN('CON EL NODO DESTINO: ');
    READLN(ND);
     { FOR K:= 1 TO ARCOS DO                         }
        BEGIN
          MATCOLOR[REG.INIC,REG.FINAL]:= REG.COLOR;
        END;  FIN DEL FOR DE ARCOS }
   IF FN > 0 THEN
   BEGIN
    WRITELN;
    WRITE ('QUE NODOS ESTAN CONECTADOS');
    WRITELN(' CON EL NODO FUENTE: ');
    FOR K:= 2 TO FN + 1 DO
    BEGIN
```

```
        WRITE ('NODO ',K,' :  ');
        READLN (NUMNODO);
        NODOESC[K]:= NUMNODO;
        ESC[NUMNODO]:= K;
     END;
   END; { FIN DEL IF FN }
   IF ND > 0 THEN
   BEGIN
     WRITELN;
     WRITE ('QUE NODOS ESTAN CONECTADOS');
     WRITELN(' CON EL NODO DESTINO: ');
     FOR K:= (NODOS - ND) TO (NODOS - 1) DO
     BEGIN
        WRITE ('NODO ',K,' :  ');
        READLN (NUMNODO);
        NODOESC[K]:= NUMNODO;
        ESC[NUMNODO]:= K;
     END;
   END; { FIN DEL IF ND }
   IF (NODOS - (FN + ND + 2)) > 0 THEN
   BEGIN
      WRITELN;
      WRITE('QUE NODOS no ESTAN CONECTADOS');
      WRITELN(' CON LOS NODOS FUENTE Y DESTINO: ');
      FOR D:= (FN + 2) TO (NODOS - (1 + ND)) DO
      BEGIN
         WRITE('NODO ',D,' :  ');
         READLN (NUMNODO);
         NODOESC[D]:= NUMNODO;
         ESC[NUMNODO]:= D;
      END;
   END; { FIN DEL IF NODOS - (FN + ND + 2) }
 END; { EJECUTA }


 { EMPIEZA EL PROCEDIMIENTO DE FACTIBILIDAD DE FLUJO }

 PROCEDURE FACTIBILIDAD;
 BEGIN { PROCEDURE FACTIBILIDAD }
     IND:= 1;
     FOR M:= 1 TO NODOS DO
     BEGIN
        FLUJOSAL[M]:= 0;
        FLUJOENT[M]:= 0;
     END;
     FOR M:= 1 TO NODOS DO
     FOR J:= 1 TO NODOS DO
     BEGIN
        FLUJOSAL[M]:= FLUJOSAL[M] + MATFLU[M,J];
        FLUJOENT[M]:= FLUJOENT[M] + MATFLU[J,M];
     END; { FIN DEL FOR J y M }
     IF FLUJOENT[T] <> FLUJOSAL[S] THEN
     BEGIN
        WRITELN ('NO ES FLUJO FACTIBLE EN LOS NODOS FUENTE-DESTINO');
        IND:= 0;
```

```
    END;
    FOR M:= 2 TO NODOS - 1 DO
    IF FLUJOENT[M] <> FLUJOSAL[M] THEN
    BEGIN
       WRITELN ('NO ES FACTIBLE EN EL NODO: ',M);
       IND:= 0;
    END;
    IF IND = 1 THEN
    WRITELN ('EL FLUJO INICIAL DADO ES FACTIBLE ');
END; {  FIN DEL PROCEDURE FACTIBILIDAD }

PROCEDURE RESMAXFLUJO;
BEGIN
    WRITELN ('EL MAXIMO FLUJO ES: ',FLUJOMAX);
END;

BEGIN {PROGRAMA PRINCIPAL}
       PINTAFLUJO;
       FACTIBILIDAD;
       IF IND = 1 THEN
       BEGIN
          MENSAJE;
          EJECUTA;
          GRAFICA;
          ESCRIBE;
          TRAZA;
          MAXFLUJO;
          IF RUTABOOL = TRUE THEN
          BEGIN
             RESMAXFLUJO;
             CORTEMINIMO;
          END
          ELSE
          BEGIN
             WRITELN('NO ES FACTIBLE');
             WRITELN('EN EL PINTADO DE LA RED');
          END;
       END;
END. {TERMINA EL PROGRAMA }
```

B I B L I O G R A F I A

- Aho, A.; Hopcroft, J.; Ullman, J.
  "Data Structures and Algorithms"
  Addison - Wesley, 1983.

- Bazaraa, M.; Jarvis, J.
  "Programación Lineal y Flujo en Redes"
  Limusa, 1981.

- Bondy, J.A.; Murty, U.S.R.
  "Graph Theory with Applications"
  The Mcmillan Press LTD, 1977.

- Dale, N.; Lilly, S.
  "Pascal y Estructura de Datos"
  McGraw-Hill, 1986.

- Dale, N.; Orshalick, D.
  "Pascal"
  McGraw-Hill, 1986.

- Denardo, E.
  "Dynamic Programming"
  Prentice-Hall, Inc., 1982.

- Gondran, M.; Minoux, M.
  "Graphs and Algorithms"
  John Wiley & Sons, 1984.

- Grogono, P.
  "Programación en Pascal"
  Addison-Wesley,1986.

- Hadley, G.
  "Linear Programming"
  Addison-Wesley, 1967.

- Kaufmann, A.
  "Métodos y Modelos de la Investigación de Operaciones"
  CECSA, 1966.

- Kaufmann, A.
  "Métodos y Modelos de la Programación Dinámica"
  CECSA, 1966.

- Keller, A.
  "Programación en Pascal"
  McGraw-Hill, 1983.

- Knuth, C.
  "The Art of Computer Programming, Vol. 1"
  Addison-Wesley, 1973.

- Prawda, J.
  "Métodos y Modelos de Investigación de Operaciones. Vol.1"
  Limusa, 1981.

- Revett, P.
  "Principles of Model Building"
  John Wiley & Sons., 1972.

- Rockafellar, R.T.
  "Network Flows and Monotropic Optimization"
  John Wiley & Sons., 1984.

- Taha, H.
  "Investigación de Operaciones"
  Representaciones y Servicios de Ingeniería, S.A., 1981.

- Tarjan, R.E.
  "Data Structures and Network Algorithms"
  Society for Industrial and Applied Mathematics, 1983.

- Tenenbaum, A.; Augenstein, M.
  "Estructura de Datos en Pascal"
  Prentice - Hall, 1986.

- Wirth, N.
  "Algoritmos + Estructuras de Datos = Programas
  Ediciones del Castillo, Madrid, 1985.