Unidad 4: Abstracciones de Datos y Algoritmia

Tema XI: Algoritmos más complejos.

Divide y Vencerás. Algoritmos Voraces. Programación Dinámica. Vuelta atrás. Ramificación y Poda. Algoritmos Evolutivos.

Programación I (Plan 1999)

Algoritmos y Estructuras de Datos II (Plan 2009)

Mgter. Oscar Adolfo Vallejos
FaCENA - UNNE

Algoritmos Complejos

1. Entender la complejidad es importante porque a la hora de resolver muchos problemas, utilizamos algoritmos ya diseñados.

2. Saber valorar su valor de complejidad puede ayudarnos mucho a conocer cómo se va a comportar el algoritmo e incluso a escoger uno u otro.

3. Existen multitud de problemas de computación que se pueden resolver mediante un algoritmo (aunque algunos pocos no tienen un algoritmo que los solucione).

4. Para resolver cada problema, podemos obtener más de un algoritmo que lo solucione, pero ¿Cual de ellos es el mejor?

DIVIDE Y VENCERÁS

- El término Divide y Vencerás en su acepción más amplia es algo más que una técnica de diseño de algoritmos.

- Sin embargo, en ese contexto, es una técnica de diseño de algoritmos que consiste en resolver un problema a partir de la solución de sub problemas del mismo tipo, pero de menor tamaño.

- Si los sub problemas son todavía relativamente grandes se aplicará de nuevo esta técnica hasta alcanzar sub problemas lo suficientemente pequeños para ser solucionados directamente.

- Naturalmente sugiere el uso de la recursión en las implementaciones de estos algoritmos.

DIVIDE Y VENCERÁS

Pasos

1. El problema puede ser descompuesto en *k subproblemas del mismo tipo, pero de menor tamaño. Es* decir, si el tamaño de la entrada es *n, hemos de conseguir dividir el problema en k subproblemas (donde* $1 \leq k \leq n$*), cada uno con una entrada de tamaño nk y* donde $0 \leq nk < n$. *(división).*

2. En segundo lugar han de resolverse independientemente todos los subproblemas, bien directamente si son elementales o bien de forma recursiva. El hecho de que el tamaño de los subproblemas sea estrictamente menor que el tamaño original del problema nos garantiza la convergencia hacia los casos elementales, también denominados casos *base.*

3. *Combinar las soluciones obtenidas en el paso anterior para construir* la solución del problema original.

Esquema general del funcionamiento de los algoritmos Divide y Vencerás

```
PROCEDURE DyV(x:TipoProblema):TipoSolucion;
   VAR i,k,:CARDINAL;
       s:TipoSolucion;
       subproblemas: ARRAY OF TipoProblema;
       subsoluciones:ARRAY OF TipoSolucion;
BEGIN
   IF EsCasobase(x) THEN
      s:=ResuelveCasoBase(x)
   ELSE
      k:=Divide(x,subproblemas);
      FOR i:=1 TO k DO
         subsoluciones[i]:=DyV(subproblemas[i])
      END;
      s:=Combina(subsoluciones)
   END;
   RETURN s
END DyV;
```

DIVIDE Y VENCERÁS

Ventajas

a) El diseño que se obtiene suele ser simple, claro, robusto y elegante, lo que da lugar a una mayor legibilidad y facilidad de depuración y mantenimiento del código obtenido.

b) Los diseños recursivos con llevan normalmente un mayor tiempo de ejecución que los iterativos, además de la complejidad espacial que puede representar el uso de la pila de recursión.

Es muy importante conseguir que los subproblemas sean independientes, es decir, que no exista solapamiento entre ellos. De lo contrario el tiempo de ejecución de estos algoritmos será exponencial. (Ej. Sucesión de Fibonacci).

DIVIDE Y VENCERÁS

En cuanto a la eficiencia hay que tener en consideración un factor importante durante el diseño del algoritmo: el número de subproblemas y su tamaño, pues esto influye de forma notable en la complejidad del algoritmo resultante.

El diseño Divide y Vencerás produce algoritmos recursivos cuyo tiempo de ejecución se puede expresar mediante una ecuación en recurrencia del tipo:

$$T(n) = \begin{cases} cn^k & \text{si } 1 \le n < b \\ aT(n/b) + cn^k & \text{si } n \ge b \end{cases}$$

donde *a, c y k son números reales, n y b son números naturales, y donde a>0, c>0, k≥0 y b>1. El valor de a representa el número de subproblemas, n/b es el tamaño* de cada uno de ellos, y la expresión *cnk representa el coste de descomponer el* problema inicial en los *a subproblemas y el de combinar las soluciones para* producir la solución del problema original, o bien el de resolver un problema elemental. La solución a esta ecuación puede alcanzar distintas complejidades.

DIVIDE Y VENCERÁS

Recordemos que el orden de complejidad de la solución a esta ecuación es:

$$T(n) \in \begin{cases} \Theta(n^k) & \text{si } a < b^k \\ \Theta(n^k \log n) & \text{si } a = b^k \\ \Theta(n^{\log_b a}) & \text{si } a > b^k \end{cases}$$

Las diferencias surgen de los distintos valores que pueden tomar *a y b, que en* definitiva determinan el número de subproblemas y su tamaño. Lo importante es observar que en todos los casos la complejidad es de orden polinómico o polilogarítmico pero nunca exponencial, frente a los algoritmos recursivos que pueden alcanzar esta complejidad en muchos casos. Esto se debe normalmente a la repetición de los cálculos que se produce al existir solapamiento en los subproblemas en los que se descompone el problema original.

DIVIDE Y VENCERÁS

Otra consideración importante a la hora de diseñar algoritmos Divide y Vencerás es el reparto de la carga entre los subproblemas, puesto que es importante que la división en subproblemas se haga de la forma más equilibrada posible.

En caso contrario nos podemos encontrar con "anomalías de funcionamiento" como le ocurre al algoritmo de ordenación Quicksort. Éste es un representante claro de los algoritmos Divide y Vencerás, y su caso peor aparece cuando existe un desequilibrio total en los subproblemas al descomponer el vector original en dos subvectores de tamaño 0 y *n–1. En este caso su* orden es O(*n2), frente a la buena complejidad, O(nlogn), que consigue cuando* descompone el vector en dos subvectores de igual tamaño.

También es interesante tener presente la dificultad y es esfuerzo requerido en cada una de estas fases va a depender del planteamiento del algoritmo concreto.

(Ej. Ordenación por Mezcla y Quicksort son dos representantes claros de esta técnica pues ambos están diseñados siguiendo el esquema presentado: dividir y combinar.

DIVIDE Y VENCERÁS

Ejemplos de esta técnica de diseño.

1.- De ordenación por Mezcla y Quicksort. Se señala la diferencia de esfuerzo que realizan en sus fases de división y combinación.

La división de Quicksort es costosa, pero una vez ordenados los dos subvectores la combinación es inmediata.

La división que realiza el método de ordenación por Mezcla consiste simplemente en considerar la mitad de los elementos, mientras que su proceso d combinación es el que lleva asociado todo el esfuerzo.

DIVIDE Y VENCERÁS

Ejemplos de esta técnica de diseño: **BÚSQUEDA BINARIA**

El problema de partida es decidir si existe un elemento dado *x en un* vector de enteros ordenado. El hecho de que esté ordenado va a permitir utilizar esta técnica, pues podemos plantear un algoritmo con la siguiente estrategia: compárese el elemento dado *x con el que ocupa la posición central del vector. En* caso de que coincida con él, hemos solucionado el problema. Pero si son distintos, pueden darse dos situaciones: que *x sea mayor que el elemento en posición central,* o que sea menor. En cualquiera de los dos casos podemos descartar una de las dos mitades del vector, puesto que si *x es mayor que el elemento en posición central,* también será mayor que todos los elementos en posiciones anteriores, y al revés. Ahora se procede de forma recursiva sobre la mitad que no hemos descartado.

La división del problema es fácil, puesto que en cada paso se divide el vector en dos mitades tomando como referencia su posición central. El problema queda reducido a uno de menor tamaño y por ello hablamos de "simplificación".

Aquí no es necesario un proceso de combinación de resultados. Su caso base se produce cuando el vector tiene sólo un elemento. En esta situación la solución del problema se basa en comparar dicho elemento con *x.*

Como el tamaño de la entrada (en este caso el número de elementos del vector a tratar) se va dividiendo en cada paso por dos, tenemos asegurada la convergencia al caso base.

```
CONST n = ...;
TYPE vector = ARRAY[1..n] OF INTEGER;
PROCEDURE BuscBin(VAR a:vector;
                  prim,ult:CARDINAL;x:INTEGER):BOOLEAN;
 VAR mitad:CARDINAL;
BEGIN
 IF (prim>=ult) THEN RETURN a[ult]=x              (* 1 *)
 ELSE                                             (* 2 *)
   mitad:=(prim+ult)DIV 2;                        (* 3 *)
   IF x=a[mitad] THEN RETURN TRUE                  (* 4 *)
   ELSIF (x<a[mitad]) THEN                        (* 5 *)
      RETURN BuscBin(a,prim,mitad-1,x)            (* 6 *)
   ELSE                                           (* 7 *)
      RETURN BuscBin(a,mitad+1,ult,x)             (* 8 *)
   END                                            (* 9 *)
 END                                              (* 10 *)
END BuscBin;
```

DIVIDE Y VENCERÁS

Ejemplos de esta técnica de diseño: **BÚSQUEDA BINARIA NO CENTRADA**

Concretamente, en el problema de la búsqueda binaria nos podemos plantear la siguiente cuestión: supongamos que en vez de dividir el vector de elementos en dos mitades del mismo tamaño, las dividimos en dos partes de tamaños 1/3 y 2/3. ¿Conseguiremos de esta forma un algoritmo mejor que el original?

```
PROCEDURE BuscBin2(Var a:vector;
          prim,ult:CARDINAL;x:INTEGER):BOOLEAN;
   VAR tercio:CARDINAL; (* posicion del elemento n/3 *)
BEGIN
   IF (prim>=ult) THEN RETURN a[ult]=x
   ELSE
      tercio:=prim+((ult-prim+1)DIV 3);
      IF x=a[tercio] THEN RETURN TRUE
      ELSIF (x<a[tercio]) THEN RETURN BuscBin2(a,prim,tercio,x)
      ELSE RETURN BuscBin2(a,tercio+1,ult,x)
      END
   END
END BuscBin2;
```

A partir del cálculo del número de operaciones elementales que se realiza en el peor caso de una invocación a esta función se puede concluir que la mejor forma de partir el vector para realizar la búsqueda binaria es por la mitad, es decir, tratando de equilibrar los subproblemas en los que realizamos la división.

DIVIDE Y VENCERÁS

Ejemplos de esta técnica de diseño: **BÚSQUEDA TERNARIA**

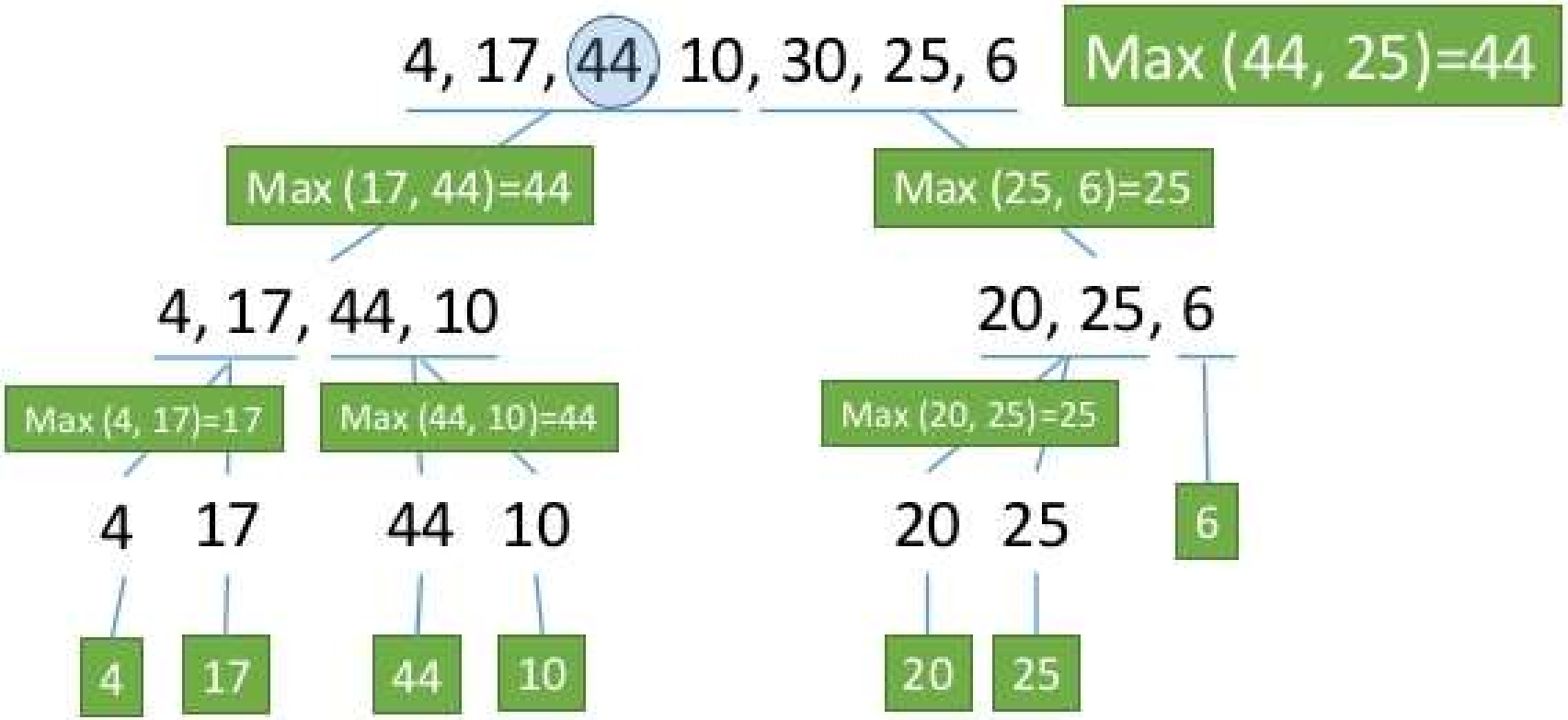
Primero compara con el elemento en posición *n/3 del vector, si éste es menor que* el elemento *x a buscar entonces compara con el elemento en posición 2n/3, y si no* coincide con *x busca recursivamente en el correspondiente subvector de tamaño 1/3* del original. ¿Conseguimos así un algoritmo mejor que el de búsqueda binaria?

DIVIDE Y VENCERÁS

```
PROCEDURE BuscBin3(VAR a:vector;prim,ult:CARDINAL;x:INTEGER):BOOLEAN;
   VAR nterc:CARDINAL; (* 1/3 del numero de elementos *)
BEGIN
   IF (prim>=ult) THEN RETURN a[ult]=x END;          (* 1 *)
   nterc:=(ult-prim+1)DIV 3;                         (* 2 *)
   IF x=a[prim+nterc] THEN RETURN TRUE          (* 3 *)
   ELSIF x<a[prim+nterc] THEN                        (* 4 *)
      RETURN BuscBin3(a,prim,prim+nterc-1,x)    (* 5 *)
   ELSIF x=a[ult-nterc] THEN RETURN TRUE             (* 6 *)
   ELSIF x<a[ult-nterc] THEN                         (* 7 *)
      RETURN BuscBin3(a,prim+nterc+1,ult-nterc-1,x) (* 8 *)
   ELSE                                              (* 9 *)
      RETURN BuscBin3(a,ult-nterc+1,ult,x)           (* 10 *)
   END                                               (* 11 *)
   END                                               (* 12 *)
END BuscBin3;
```

A partir del número de operaciones elementales se llega a que el tiempo de ejecución de la búsqueda ternaria es mayor al de la búsqueda binaria, por lo que no consigue ninguna mejora con este algoritmo.

Ejemplo: Máximo de un Array



Cada problema se divide en dos nuevos subproblemas (buscar máximo en cada mitad)
Hasta que la mitad tenga 1 elemento (caso base), el máximo

Búsqueda Binaria

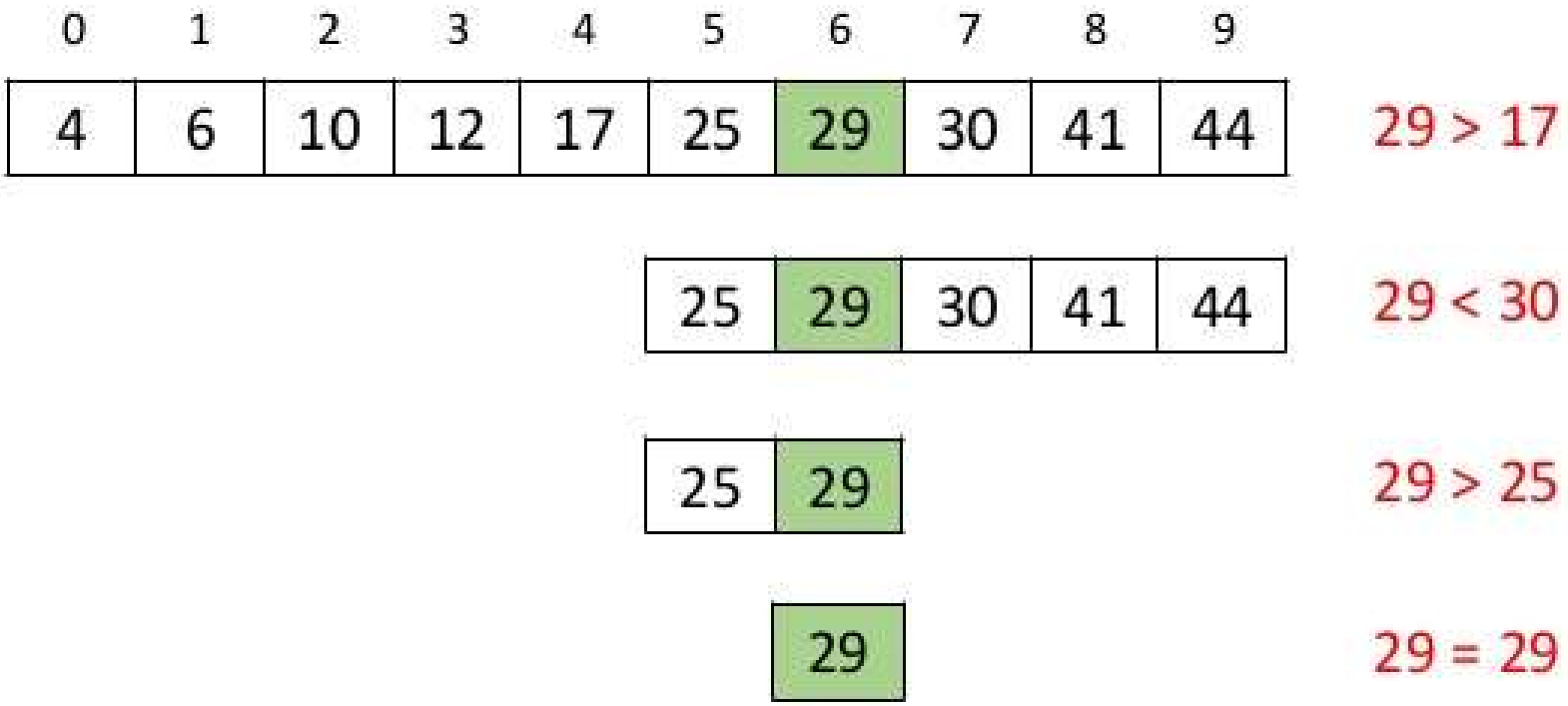
Ejemplos de la vida cotidiana

- Buscar una palabra en un diccionario
- Buscar un nombre en el directorio telefónico
- Buscar una página de un libro
- Retomar la visualización de un video

En general **optimizamos** la búsqueda cuando los elementos están ordenados

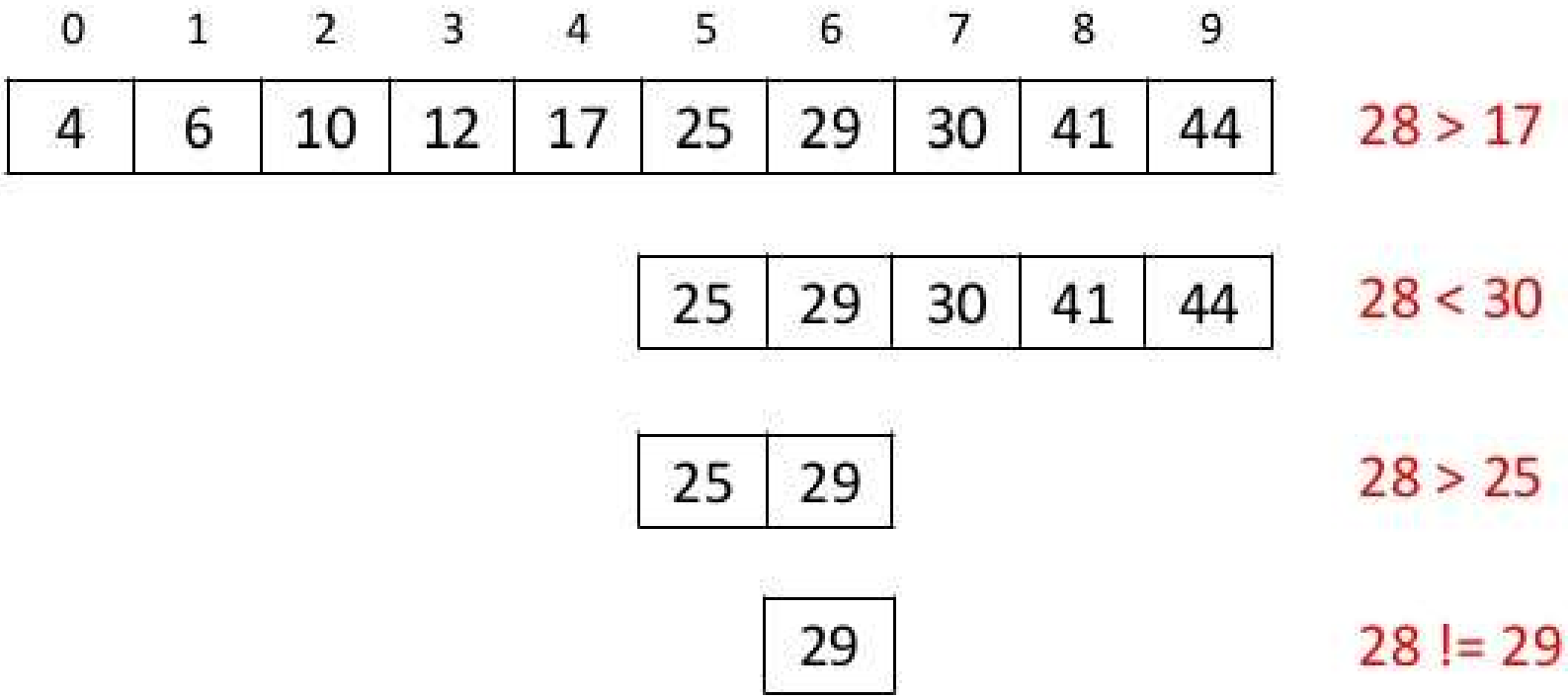
Optimización en este caso significa hacer menos comparaciones para encontrar el elemento... o determinar que no está

Búsqueda Binaria



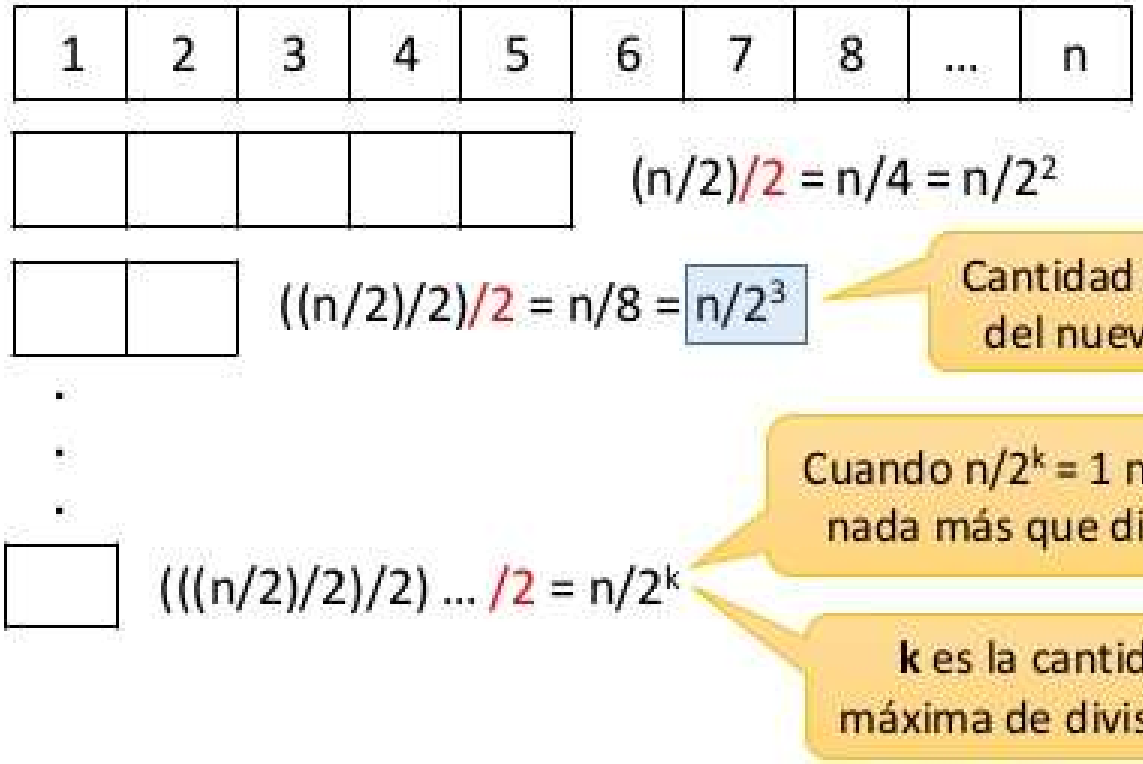
Se encontró el elemento haciendo solo 4 comparaciones!

Búsqueda Binaria



Se determinó que el elemento no está haciendo solo 4 comparaciones!

Búsqueda Binaria



$k?, \quad \frac{n}{2^k} = 1$

$$\frac{n}{2^k} = 1$$

$$n = 2^k$$

$$\log_2 n = k$$

$$\log_2 10 = \lceil 3.32 \rceil = 4$$

DIVIDE Y VENCERÁS

Ejemplos de esta técnica de diseño:

MEDIANA DE DOS VECTORES

EL ELEMENTO EN SU POSICIÓN

REPETICIÓN DE CÁLCULOS EN FIBONACCI

EL ELEMENTO MAYORITARIO

LA MODA DE UN VECTOR

EL TORNEO DE TENIS

ALGORITMOS ÁVIDOS

Es un método muy sencillo y que puede ser aplicado a numerosos problemas, especialmente los de optimización.

Dado un problema con *n entradas el método consiste en obtener un subconjunto* de éstas que satisfaga una determinada restricción definida para el problema.

Cada uno de los subconjuntos que cumplan las restricciones diremos que son soluciones *prometedoras. Una solución prometedora que maximice o minimice una función* objetivo la denominaremos solución óptima.

ALGORITMOS ÁVIDOS

Si un problema es susceptible de ser resuelto por un algoritmo ávido, debe cumplir:

- Un conjunto de *candidatos, que corresponden a las n entradas de problema.*

- Una *función de selección que en cada momento determine el candidato idóneo* para formar la solución de entre los que aún no han sido seleccionados ni rechazados.

- Una función que compruebe si un cierto subconjunto de candidatos es *prometedor. Entendemos por prometedor que sea posible seguir añadiendo* candidatos y encontrar una solución.

- Una *función objetivo que determine el valor de la solución hallada. Es la* función que queremos maximizar o minimizar.

- Una función que compruebe si un subconjunto de estas entradas es solución al problema, sea óptima o no.

ALGORITMOS ÁVIDOS

Resumen del funcionamiento de los algoritmos ávidos:

- Para resolver el problema, un algoritmo ávido tratará de encontrar un subconjunto de candidatos tales que, cumpliendo las restricciones del problema, constituya la solución óptima.

- Para ello trabajará por etapas, tomando en cada una de ellas la decisión que le parece la mejor, sin considerar las consecuencias futuras, y por tanto escogerá de entre todos los candidatos el que produce un óptimo local para esa etapa, suponiendo que será a su vez óptimo global para el problema.

- Antes de añadir un candidato a la solución que está construyendo comprobará si es prometedora al añadirlo. En caso afirmativo lo incluirá en ella y en caso contrario descartará este candidato para siempre y no volverá a considerarlo.

- Cada vez que se incluye un candidato comprobará si el conjunto obtenido es solución.

ALGORITMOS ÁVIDOS

Los algoritmos ávidos construyen la solución en etapas sucesivas, tratando siempre de tomar la decisión óptima para cada etapa.

```
PROCEDURE AlgoritmoAvido(entrada:CONJUNTO):CONJUNTO;
   VAR x:ELEMENTO; solucion:CONJUNTO; encontrada:BOOLEAN;
BEGIN
   encontrada:=FALSE; crear(solucion);
   WHILE NOT EsVacio(entrada) AND (NOT encontrada) DO
      x:=SeleccionarCandidato(entrada);
      IF EsPrometedor(x,solucion) THEN
         Incluir(x,solucion);
         IF EsSolucion(solucion) THEN
            encontrada:=TRUE
         END;
      END
   END;
   RETURN solucion;
END AlgoritmoAvido;
```

Los algoritmos ávidos son muy fáciles de implementar y producen soluciones muy eficientes. Entonces cabe preguntarse ¿por qué no utilizarlos siempre?
En primer lugar, porque no todos los problemas admiten esta estrategia de solución.

ALGORITMOS ÁVIDOS

- Encontrar la función de selección que nos garantice que el candidato escogido o rechazado en un momento determinado es el que ha de formar parte o no de la solución óptima sin posibilidad de reconsiderar dicha decisión. Por ello, una parte muy importante de este tipo de algoritmos es la demostración formal de que la función de selección escogida consigue encontrar óptimos globales para cualquier entrada del algoritmo.
- No basta con diseñar un procedimiento ávido, que seguro que será rápido y eficiente (en tiempo y en recursos), sino que hay que demostrar que siempre consigue encontrar la solución óptima del problema.
- Debido a su eficiencia, este tipo de algoritmos es muchas veces utilizado aun en los casos donde se sabe que no necesariamente encuentran la solución óptima.
- En algunas ocasiones la situación nos obliga a encontrar pronto una solución razonablemente buena, aunque no sea la óptima, puesto que si la solución óptima se consigue demasiado tarde, ya no vale para nada.

ALGORITMOS ÁVIDOS

- También hay otras circunstancias, en donde lo que interesa es conseguir cuanto antes una solución del problema y, a partir de la información suministrada por ella, conseguir la óptima más rápidamente. Es decir, la eficiencia de este tipo de algoritmos hace que se utilicen aunque no consigan resolver el problema de optimización planteado, sino que sólo den una solución "aproximada".

- El nombre de algoritmos ávidos, también conocidos como voraces (su nombre original proviene del término inglés *greedy) se debe a su comportamiento: en cada* etapa "toman lo que pueden" sin analizar consecuencias, es decir, son glotones por naturaleza.

- En este tipo de algoritmos el proceso no acaba cuando disponemos de la implementación del procedimiento que lo lleva a cabo. Lo importante es la demostración de que el algoritmo encuentra la solución óptima en todos los casos, o bien la presentación de un contraejemplo que muestra los casos en donde falla.

ALGORITMOS ÁVIDOS

Ej.: EL PROBLEMA DEL CAMBIO

Suponiendo que el sistema monetario de un país está formado por monedas de valores *v1,v2,...,vn, el problema del cambio de dinero consiste en descomponer* cualquier cantidad dada *M en monedas de ese país utilizando el menor número* posible de monedas.

En primer lugar, es fácil implementar un algoritmo ávido para resolver este problema, que es el que sigue el proceso que usualmente utilizamos en nuestra vida diaria. Sin embargo, tal algoritmo va a depender del sistema monetario utilizado y por ello vamos a plantearnos dos situaciones para las cuales deseamos conocer si el algoritmo ávido encuentra siempre la solución óptima:

a) Suponiendo que cada moneda del sistema monetario del país vale al menos el doble que la moneda de valor inferior, que existe una moneda de valor unitario, y que disponemos de un número ilimitado de monedas de cada valor.

b) Suponiendo que el sistema monetario está compuesto por monedas de valores 1, *p, p2, p3,..., pn, donde p > 1 y n > 0, y que también disponemos de un número* ilimitado de monedas de cada valor.

ALGORITMOS ÁVIDOS

```
TYPE  MONEDAS =(M500,M200,M100,M50,M25,M5,M1);(*sistema monetario*)
      VALORES = ARRAY MONEDAS OF CARDINAL; (* valores de monedas *)
      SOLUCION = ARRAY MONEDAS OF CARDINAL;

PROCEDURE Cambio(n:CARDINAL;VAR valor:VALORES;VAR cambio:SOLUCION);
(* n es la cantidad a descomponer, y el vector "valor" contiene los
valores de cada una de las monedas del sistema monetario *)
   VAR moneda:MONEDAS;
BEGIN
   FOR moneda:=FIRST(MONEDAS) TO LAST(MONEDAS) DO
      cambio[moneda]:=0
   END;
   FOR moneda:=FIRST(MONEDAS) TO LAST(MONEDAS) DO
      WHILE valor[moneda]<=n DO
         INC(cambio[moneda]);
         DEC(n,valor[moneda])
      END
   END
END Cambio;
```

Este algoritmo es de complejidad lineal respecto al número de monedas del país, y por tanto muy eficiente.

Este algoritmo es de complejidad lineal respecto al número de monedas del país, y por tanto muy eficiente.

Respecto a las dos cuestiones planteadas, comenzaremos por la primera.

Supongamos que nuestro sistema monetario esta compuesto por las siguientes monedas:

TYPE MONEDAS = (M11,M5,M1); valor:={11,5,1};

Tal sistema verifica las condiciones del enunciado pues disponemos de moneda de valor unitario, y cada una de ellas vale más del doble de la moneda inmediatamente inferior.

Consideremos la cantidad $n = 15$. El algoritmo ávido del cambio de monedas descompone tal cantidad en:

$15 = 11 + 1 + 1 + 1 + 1,$

es decir, mediante el uso de cinco monedas. Sin embargo, existe una descomposición que utiliza menos monedas (exactamente tres):

$15 = 5 + 5 + 5.$

Aunque queda comprobado que bajo estas circunstancias el diseño ávido no puede utilizarse, las razones por las que el algoritmo falla quedarán al descubierto más adelante.

ALGORITMOS ÁVIDOS

Ej.: RECORRIDOS DEL CABALLO DE AJEDREZ

Dado un tablero de ajedrez y una casilla inicial, queremos decidir si es posible que un caballo recorra todos y cada uno de los escaques sin duplicar ninguno. No es necesario en este problema que el caballo vuelva al escaque de partida. Un posible algoritmo ávido decide, en cada iteración, colocar el caballo en la casilla desde la cual domina el menor número posible de casillas aún no visitadas.

a) Implementar dicho algoritmo a partir de un tamaño de tablero *nxn y una casilla* inicial (*x0,y0).*

b) Buscar, utilizando el algoritmo realizado en el apartado anterior, todas las casillas iniciales para los que el algoritmo encuentra solución.

c) Basándose en los resultados del apartado anterior, encontrar el patrón general de las soluciones del recorrido del caballo.

Solución

d)Para implementar el algoritmo pedido comenzaremos definiendo las constantes y tipos que utilizaremos:

```
CONST TAMMAX = ... (* dimension maxima del tablero *)
TYPE tablero = ARRAY[1..TAMMAX],[1..TAMMAX] OF INT
```

Cada una de las casillas del tablero va a almacenar un número natural que indica el número de orden del movimiento del caballo en el que visita la casilla. Podrá tomar también el valor cero, indicando que la casilla no ha sido visitada aún.

Inicialmente todas las casillas tomarán este valor.

Una posible implementación del algoritmo viene dada por la función *Caballo* que se muestra a continuación, la cual, dado un tablero *t*, su dimensión *n* y una posición inicial (*x*,*y*), decide si el caballo recorre todo el tablero o no.

Algoritmo Caballo(VAR t:tablero; n:INT; x,y:INT):BOOLEAN

```
BEGIN
  InicTablero(t,n)     (* inicializa las casillas del tablero a 0 *)
  FOR i ← 1 TO n*n DO
        t[x,y]← i
        IF NOT NuevoMov(t,n,x,y) AND (i<n*n-1) THEN
              RETURN FALSE
        END
  END
RETURN TRUE    (* hemos recorrido las n*n casillas *)
END Caballo
```

```
algoritmo NuevoMov(VAR t:tablero; n:INT; VAR x,y:INT):BOOLEAN
(*Esta función es la que va a ir calculando la nueva casilla a la que salta el caballo
siguiendo la indicación del enunciado, devolviendo FALSE si no puede Moverse*)

INT accesibles,minaccesibles,i,solx,soly,nuevax,nuevay
BEGIN
  minaccesibles ← 9
  solx ← x
  soly ← y
  FOR i:=1 TO 8 DO
        IF Salto(t,n,i,x,y,nuevax,nuevay) THEN
              accesibles:=Cuenta(t,n,nuevax,nuevay)
              IF (accesibles>0) AND (accesibles<minaccesibles) THEN
                    minaccesibles ← accesibles
                    solx ← nuevax  soly ← nuevay
              END
        END
  END
X ← solx
 y ← soly
RETURN (minaccesibles<9)
END NuevoMov
```

```
Algoritmo Salto(VAR t:tablero; n,i,x,y: INT;VAR nx,ny:INT ) :BOOL
(* i indica el numero del movimiento, (x,y) es la casilla actual, y (nx,ny) es la casilla a
donde salta. *)
BEGIN
        CASE i OF
                1: nx ← x-2     ny ← y+1
                2: nx ← x-1     ny ← y+2
                3: nx ← x+1     ny ← y+2
                4: nx ← x+2     ny ← y+1
                5: nx ← x+2     ny ← y-1
                6: nx ← x+1     ny ← y-2
                7: nx ← x-1     ny ← y-2
                8: nx ← x-2     ny ← y-1
END
RETURN((1<=nx) AND (nx<=n) AND (1<=ny) AND (ny<=n) AND (t[nx,ny]=0))
END Salto
```

La función Salto calcula las coordenadas de la casilla a donde salta el caballo (tiene 8 posibilidades), y devuelve si es posible o no realizar ese movimiento ya que la casilla puede estar ocupada o bien salirse del tablero.

Dicha función intenta los movimientos en el orden que muestra la siguiente figura:

	2		3	
1				4
		X		

```
Cuenta(VAR t:tablero; n,x,y:INT):INT
INT acc,i,nx,ny
BEGIN
        acc ← 0
        FOR i←1 TO 8 DO
                IF Salto(t,n,i,x,y,nx,ny) THEN
                        INC(acc)
                 END
        END
RETURN acc
END Cuenta
```

La otra función es *Cuenta*, que devuelve el número de casillas a las que el caballo puede saltar desde una posición dada.

ALGORITMOS ÁVIDOS

Ej.: EL VIAJANTE DE COMERCIO

Se conocen las distancias entre un cierto número de ciudades. Un viajante debe, a partir de una de ellas, visitar cada ciudad exactamente una vez y regresar al punto de partida habiendo recorrido en total la menor distancia posible.

Este problema también puede ser enunciado más formalmente como sigue: dado un grafo *g conexo y ponderado y dado uno de sus vértices v0, encontrar el ciclo* Hamiltoniano de coste mínimo que comienza y termina en *v0.*

Cara a intentar solucionarlo mediante un algoritmo ávido, nos planteamos las siguientes estrategias:

a) Sea (*C,v) el camino construido hasta el momento que comienza en v0 y termina* en *v. Inicialmente C es vacío y v = v0. Si C contiene todos los vértices de g, el* algoritmo incluye el arco (*v,v0) y termina. Si no, incluye el arco (v,w) de* longitud mínima entre todos los arcos desde *v a los vértices w que no están en el* camino *C.*

b) Otro posible algoritmo ávido escogería en cada iteración el arco más corto aún no considerado que cumpliera las dos condiciones siguientes: (*i) no formar un* ciclo con los arcos ya seleccionados, excepto en la última iteración, que es donde completa el viaje; y (*ii) no es el tercer arco que incide en un mismo* vértice de entre los ya escogidos.

ALGORITMOS ÁVIDOS

Ej.: **LA MOCHILA**

Dados *n elementos e1, e2, ..., en con pesos p1, p2, ..., pn y beneficios b1, b2, ..., bn, y* dada una mochila capaz de albergar hasta un máximo de peso *M (capacidad de la* mochila), queremos encontrar las proporciones de los *n elementos x1, x2, ..., xn* (0 ≤ *xi* ≤ *1) que tenemos que introducir en la mochila de forma que la suma de los* beneficios de los elementos escogidos sea máxima.

Esto es, hay que encontrar valores (*x1, x2, ..., xn) de forma que se maximice la* cantidad $\sum_{i=1}^{n} b_i x_i$

sujeta a la restricción $\sum_{i=1}^{n} p_i x_i \leq M.$

ALGORITMOS ÁVIDOS

LA DIVISIÓN EN PÁRRAFOS

LOS ALGORITMOS DE PRIM, KRUSKAL y DIJKSTRA

EL FONTANERO DILIGENTE

LA ASIGNACIÓN DE TAREAS

LOS FICHEROS Y EL DISQUETE

EL CAMIONERO CON PRISA

LA MULTIPLICACIÓN ÓPTIMA DE MATRICES

Algoritmo de KRUSKAL:

Es un algoritmo de la teoría de grafos para encontrar un árbol recubridor mínimo en un grafo conexo y ponderado. Es decir, busca un subconjunto de aristas que, formando un árbol, incluyen todos los vértices y donde el valor total de todas las aristas del árbol es el mínimo.

Algoritmo de KRUSKAL:

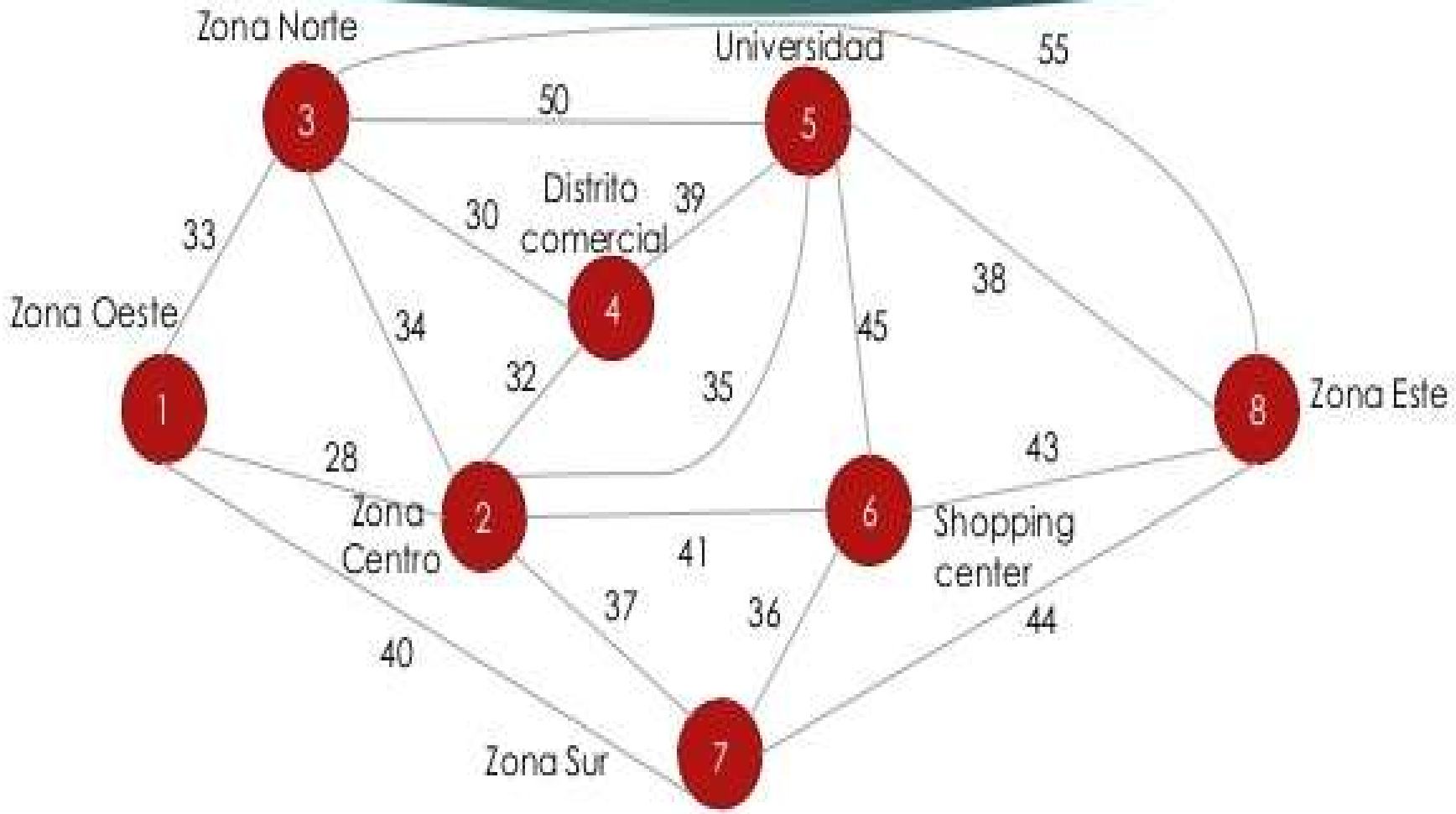
1. • Comenzar en forma arbitraria en cualquier nodo y conectarlo con el mas próximo (menos distancia o costoso).
2. • Identificar el nodo no conectado que esta mas cerca o menos costos de alguno de los nodos conectados. Deshacer los empates de forma arbitraria.
3. • Repartir este paso hasta que se hayan conectado todos los nodos.

Ejemplo: EL TRANSITO DEL DISTRITO METROPOLITANO

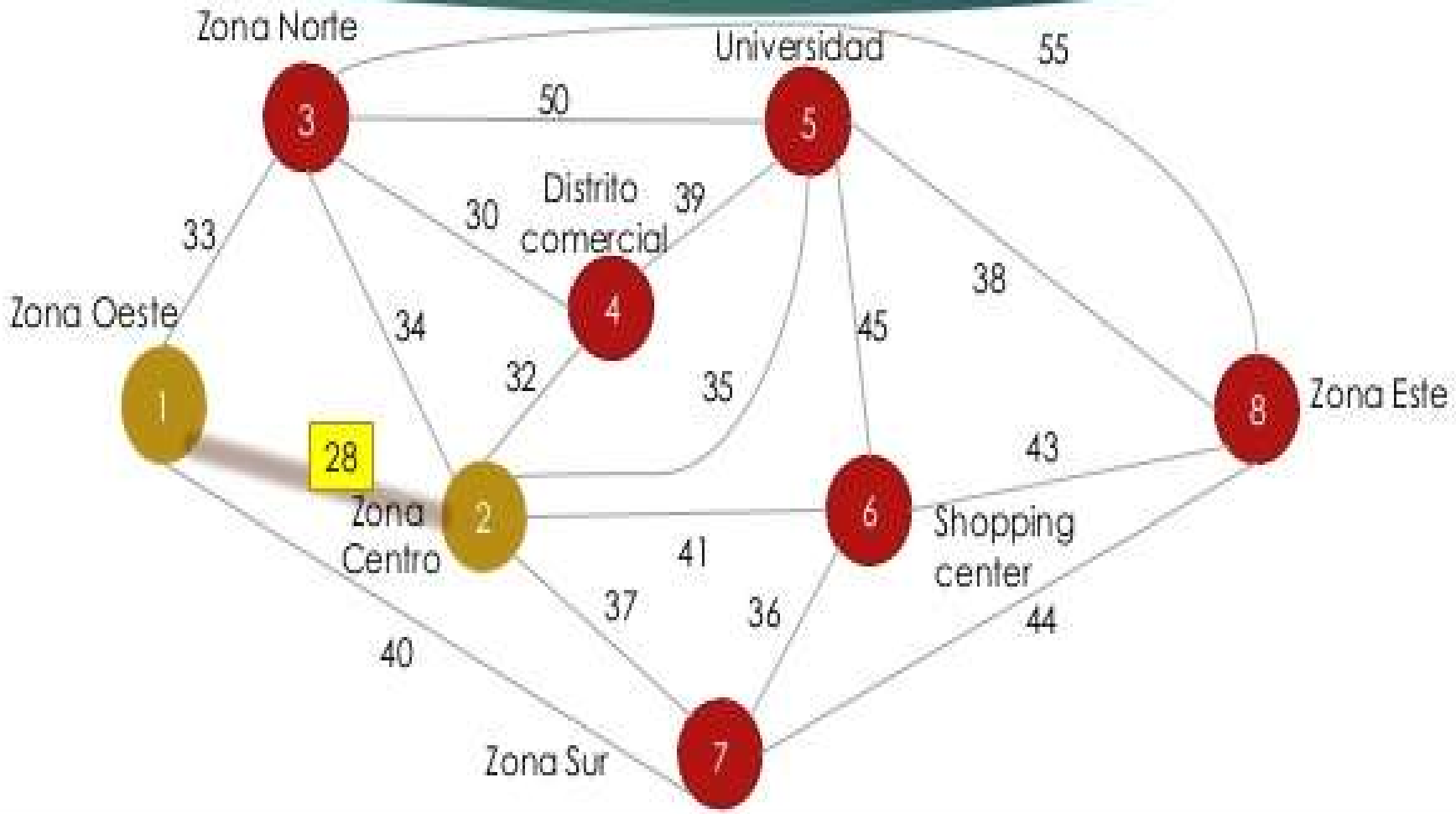
EL TRANSITO DEL DISTRITO METROPOLITANO

- ✓ Una ciudad esta planificando el desarrollo de una nueva línea en sistemas de transito.
- ✓ El sistema debe unir 8 residencias y centros comerciales.
- ✓ El distrito metropolitano de transito necesita seleccionar un conjunto de líneas que conecten todos los centros a una mínimo costo.
- ✓ La red seleccionada debe permitir:
  Factibilidad de las líneas que deben ser construidas.
  Mínimo costo posible por línea.

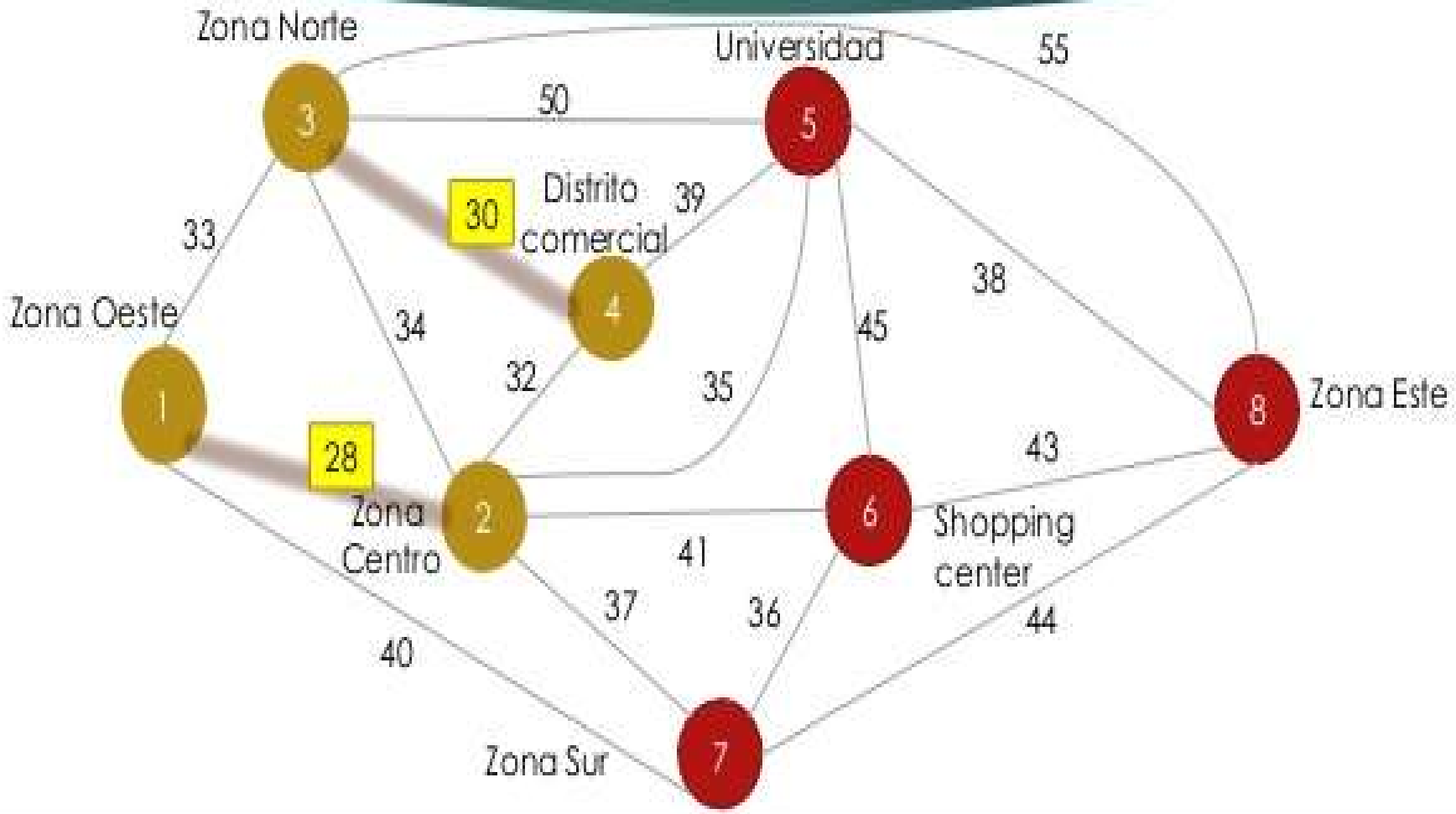
RED QUE REPRESENTA EL ARBOL EXPANDIDO METODO DE KRUSKAL



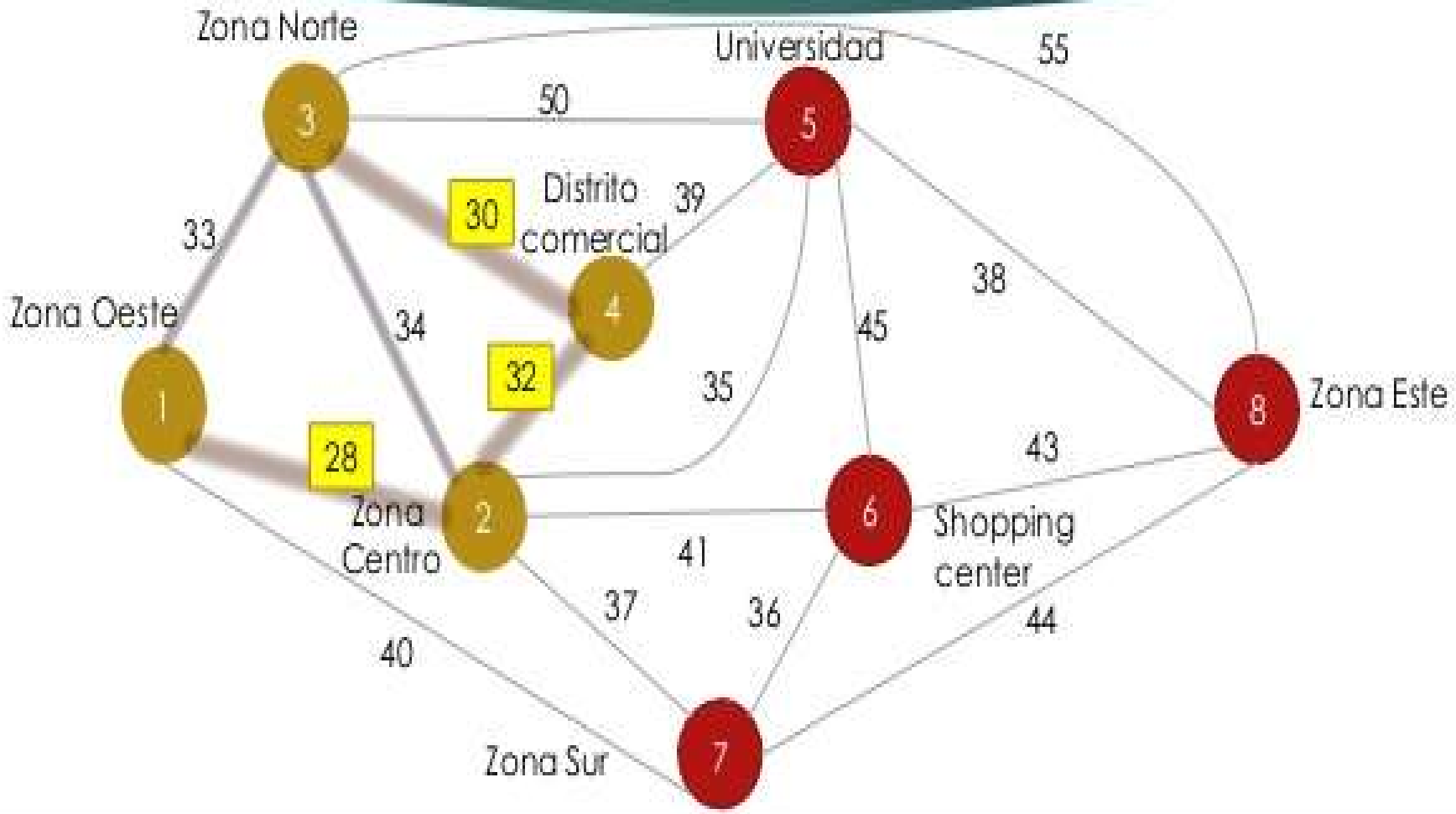
RED QUE REPRESENTA EL ARBOL EXPANDIDO METODO DE KRUSKAL



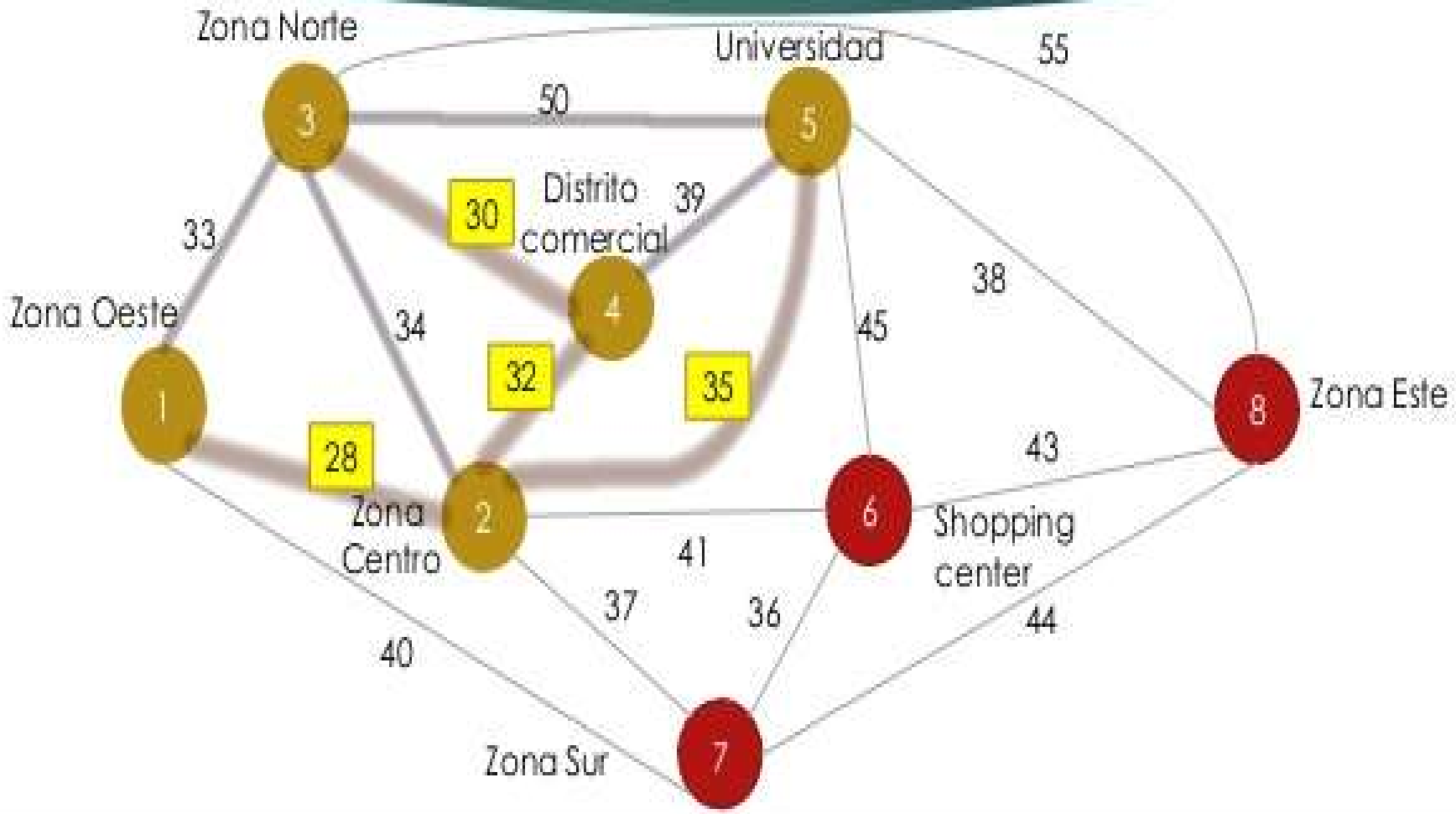
RED QUE REPRESENTA EL ARBOL EXPANDIDO METODO DE KRUSKAL



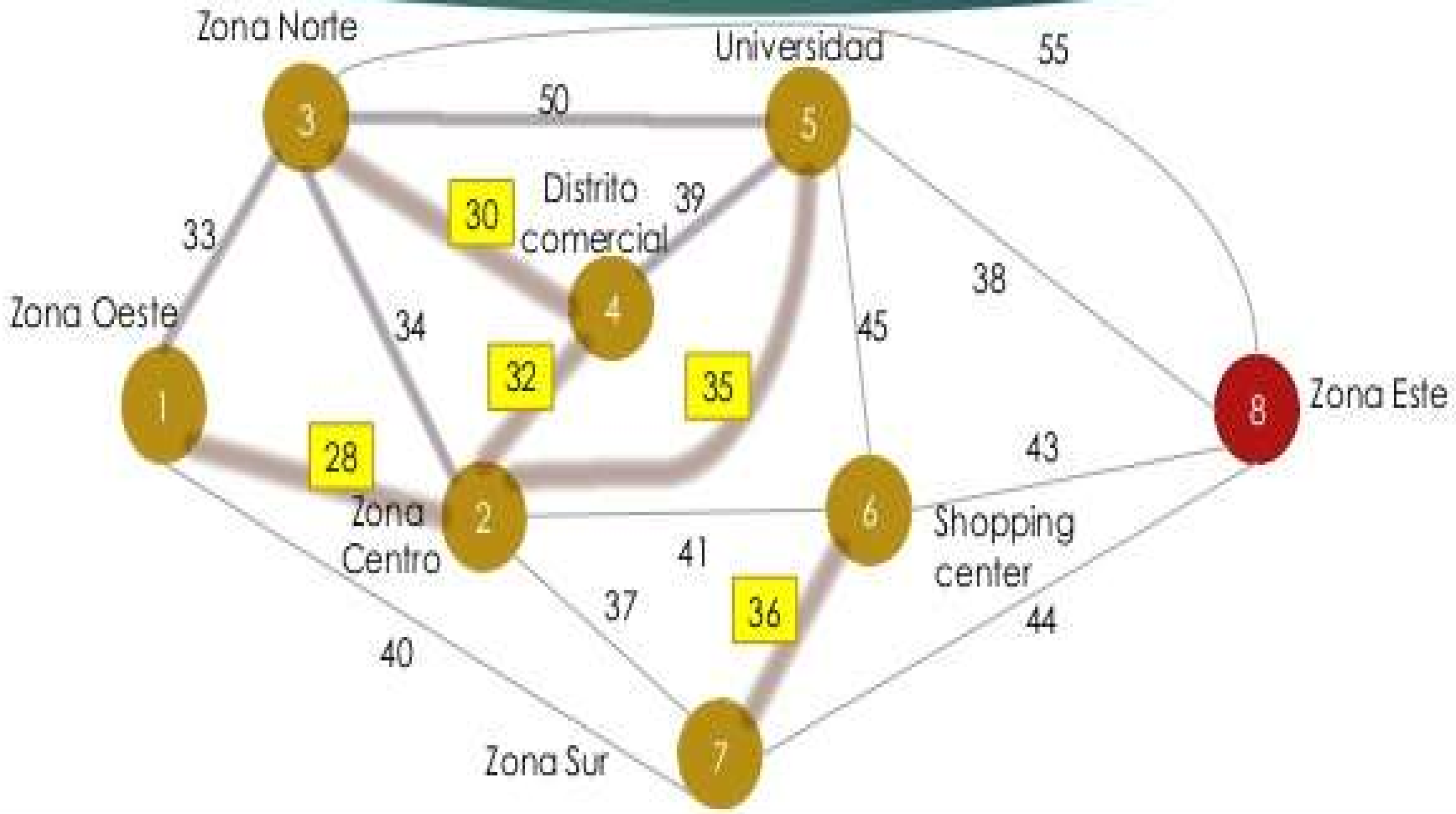
RED QUE REPRESENTA EL ARBOL EXPANDIDO METODO DE KRUSKAL



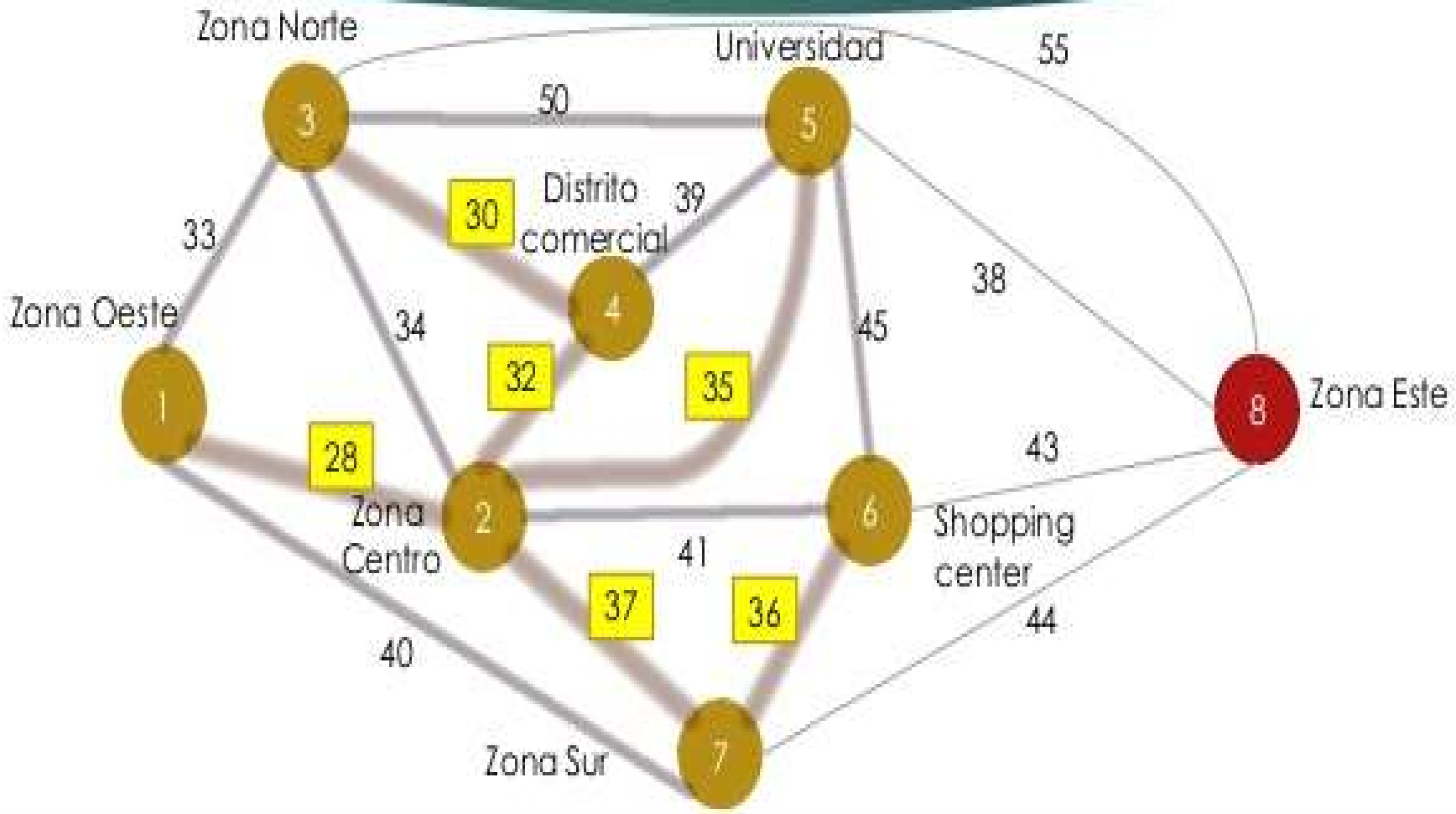
RED QUE REPRESENTA EL ARBOL EXPANDIDO METODO DE KRUSKAL



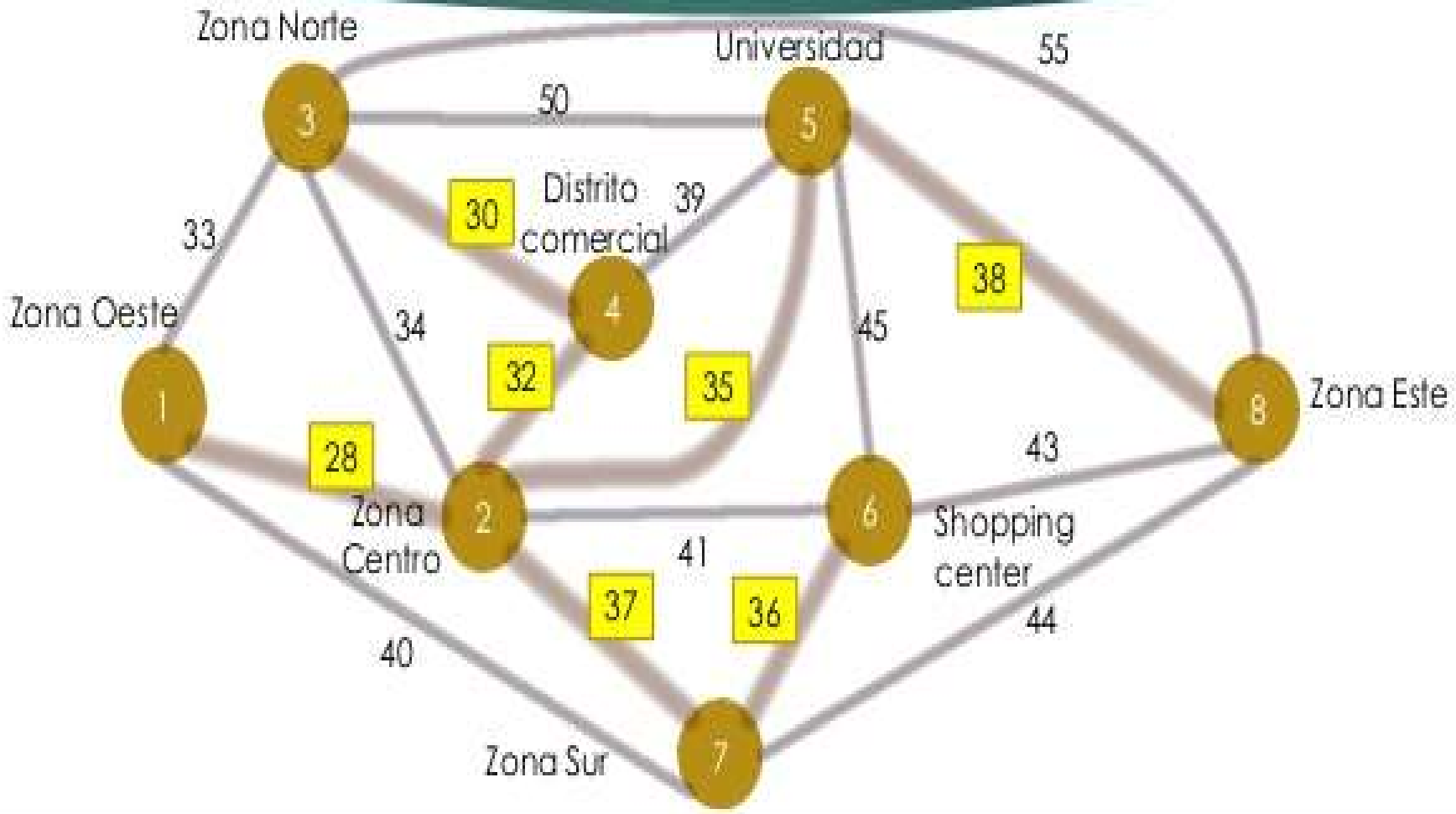
RED QUE REPRESENTA EL ARBOL EXPANDIDO METODO DE KRUSKAL



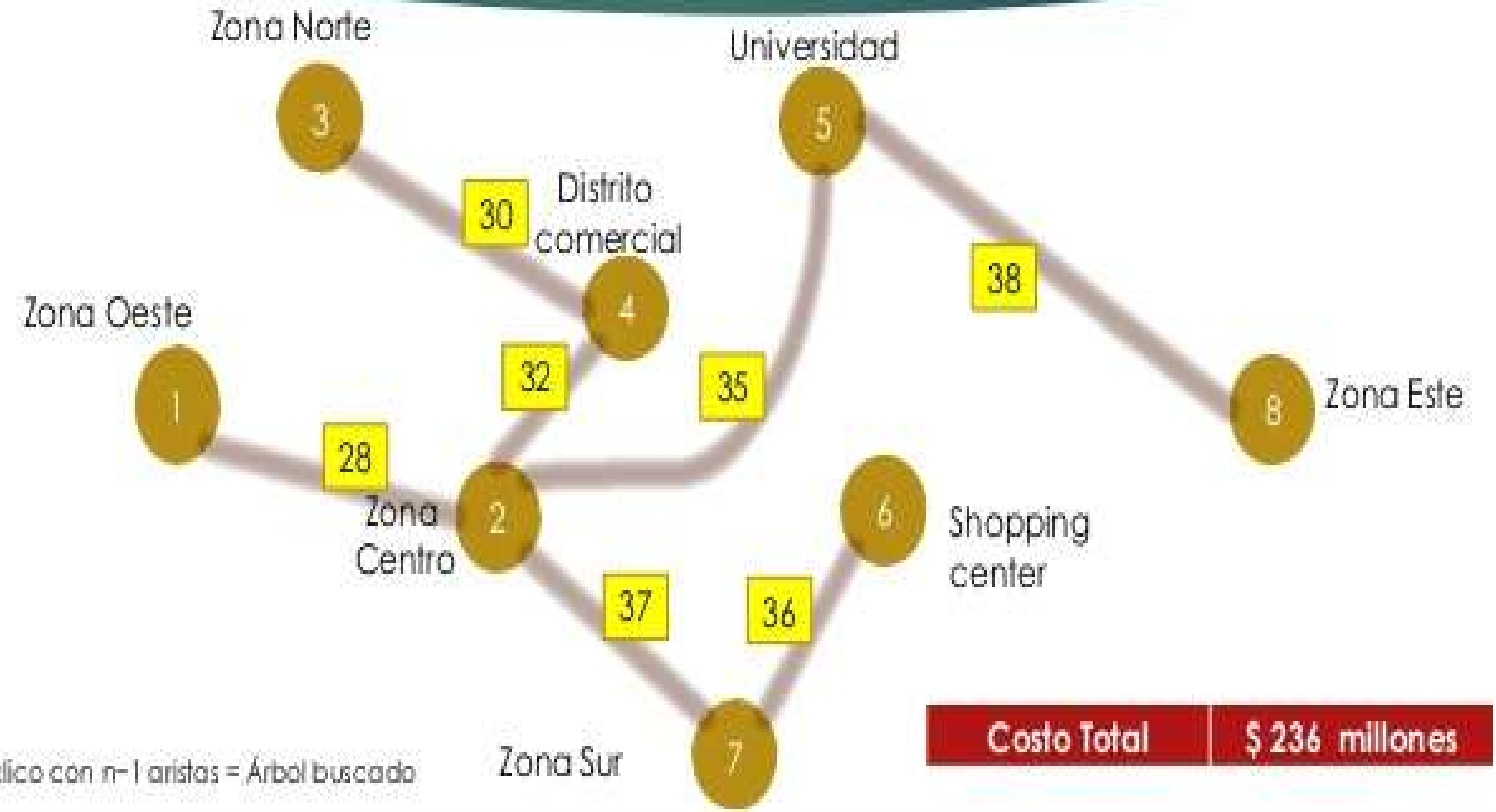
RED QUE REPRESENTA EL ARBOL EXPANDIDO METODO DE KRUSKAL



RED QUE REPRESENTA EL ARBOL EXPANDIDO METODO DE KRUSKAL



RED QUE REPRESENTA EL ARBOL EXPANDIDO METODO DE KRUSKAL



Aciclico con n-1 aristas = Árbol buscado

Costo Total	$ 236 millones

```
void Kruskal( GRAFO G, CONJUNTO T )
{
   CONJUNTO_V  U;
   VERTICE     v;

   T = ∅;
   for( cada vértice v de G )
     construye un árbol con v;
   Ordena los arcos de G en orden no decreciente;
   while( Haya más de un árbol ) {
     Sea (u,v) es el arco de menor costo tal que el árbol
     de u es diferente al árbol de v;
     Mezcla los árboles de u y de v en uno solo;
     T = T ∪ { (u,v) };
   }
}
```

Algoritmo de PRIM:

Es un algoritmo perteneciente a la teoría de los grafos para encontrar un árbol recubridor mínimo en un grafo conexo, no dirigido y cuyas aristas están etiquetadas.

El algoritmo incrementa continuamente el tamaño de un árbol, comenzando por un vértice inicial al que se le van agregando sucesivamente vértices cuya distancia a los anteriores es mínima.

Algoritmo de PRIM:

1. • Seleccionar inicialmente cualquier nodo y conectarlo con el mas próximo que contenga el arco de menor costo o distancia.
2. • Completar de red interactivamente, identificando el nodo no conectado que esta mas cerca o menos costoso de algunos de los nodos conectados.
3. • Agregar este nodo al conjunto de nodos conectado. En caso de empate este se rompe en forma arbitraria.

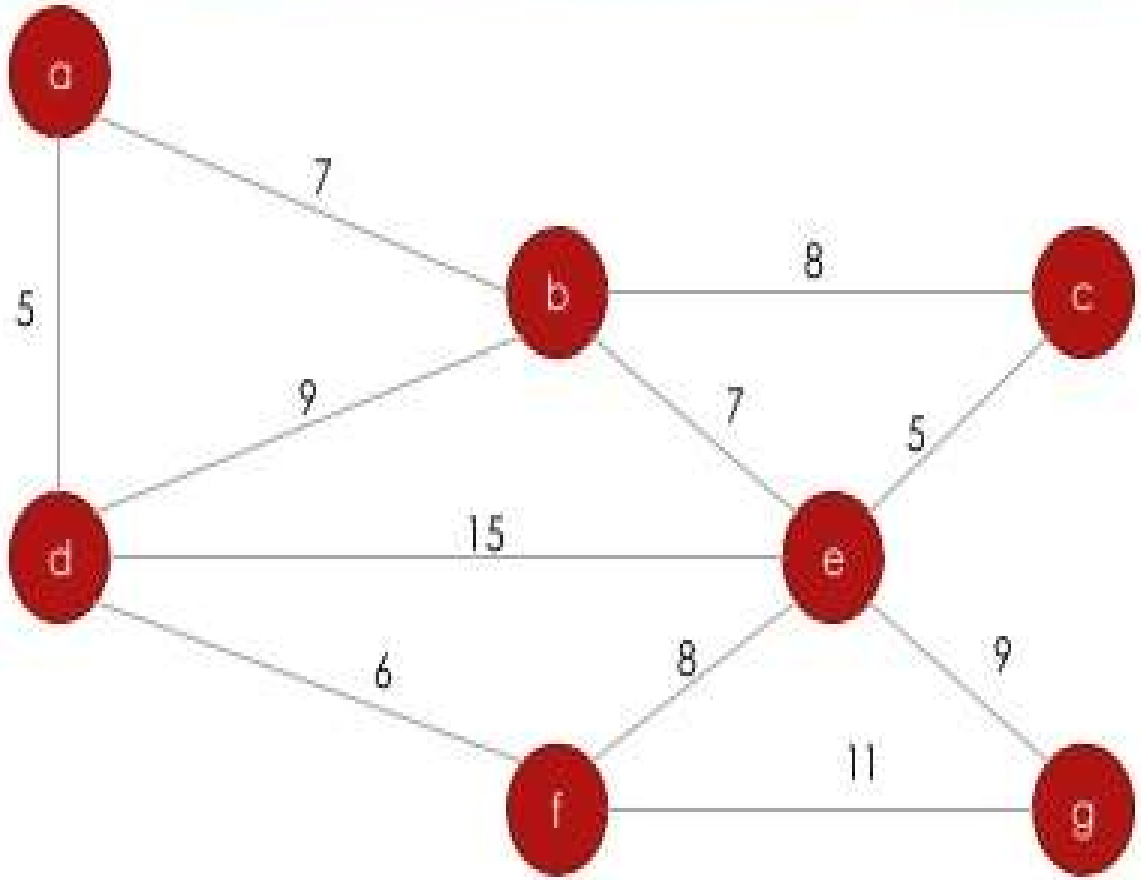
Objetivo:

Encontrar el árbol recubridor mas corto.

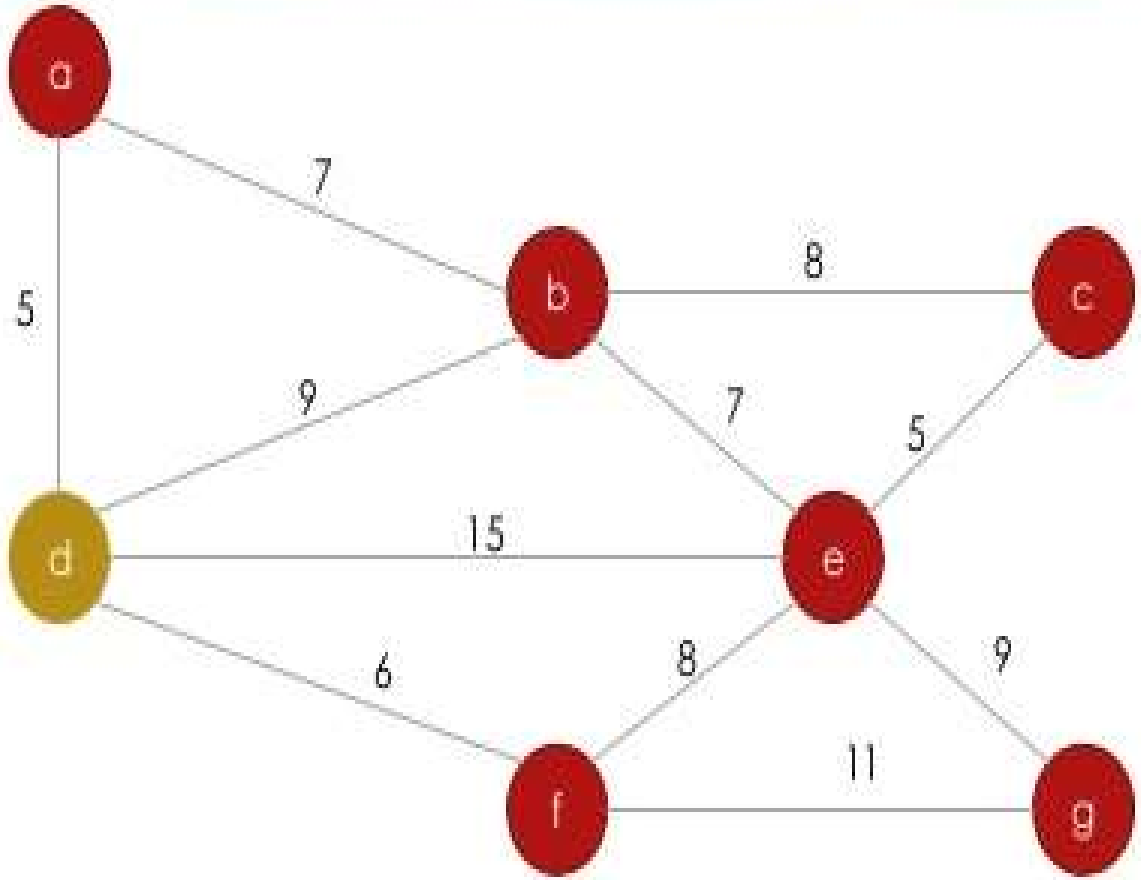
Requisitos:

Ser un grafo conexo.
Ser un grafos sin ciclos.
Tener todos los arcos etiquetados.

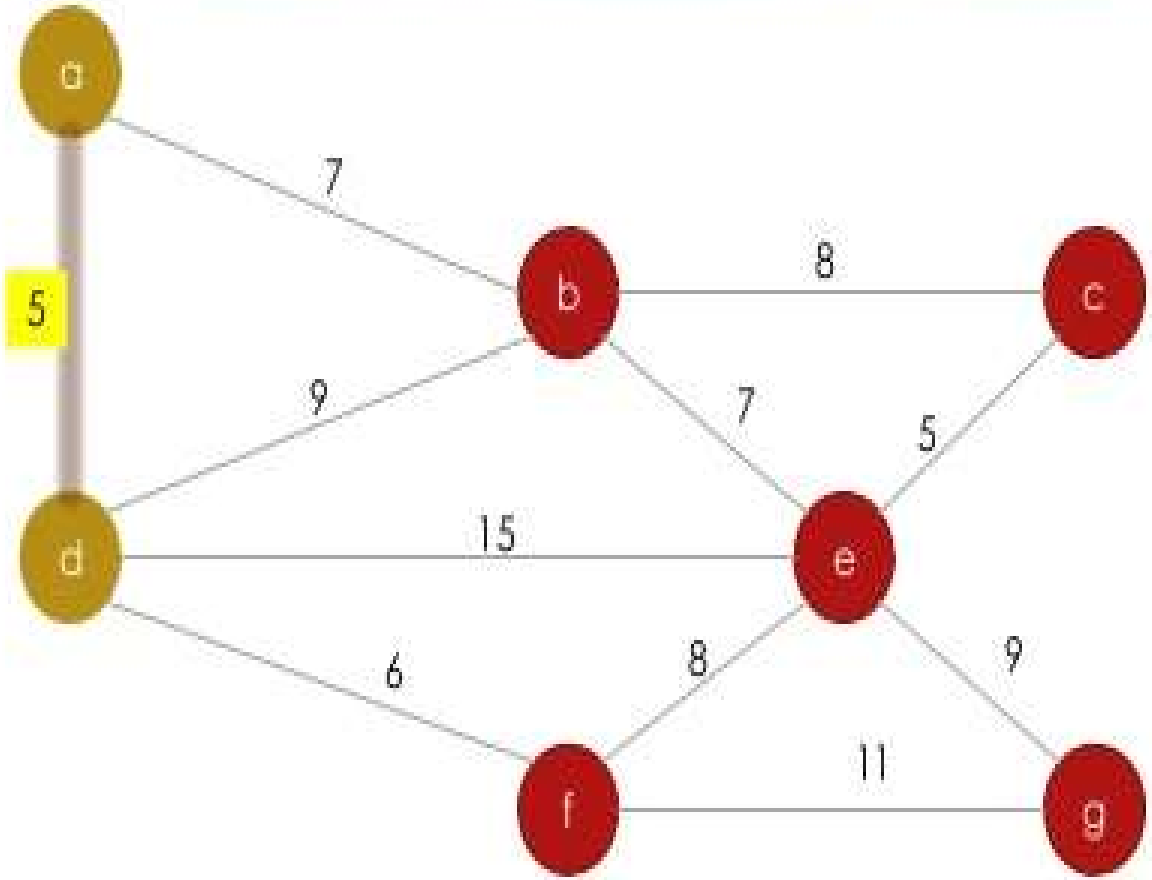
ARBOL DE MINIMA EXPANSIÓN
METODO DE PRIM



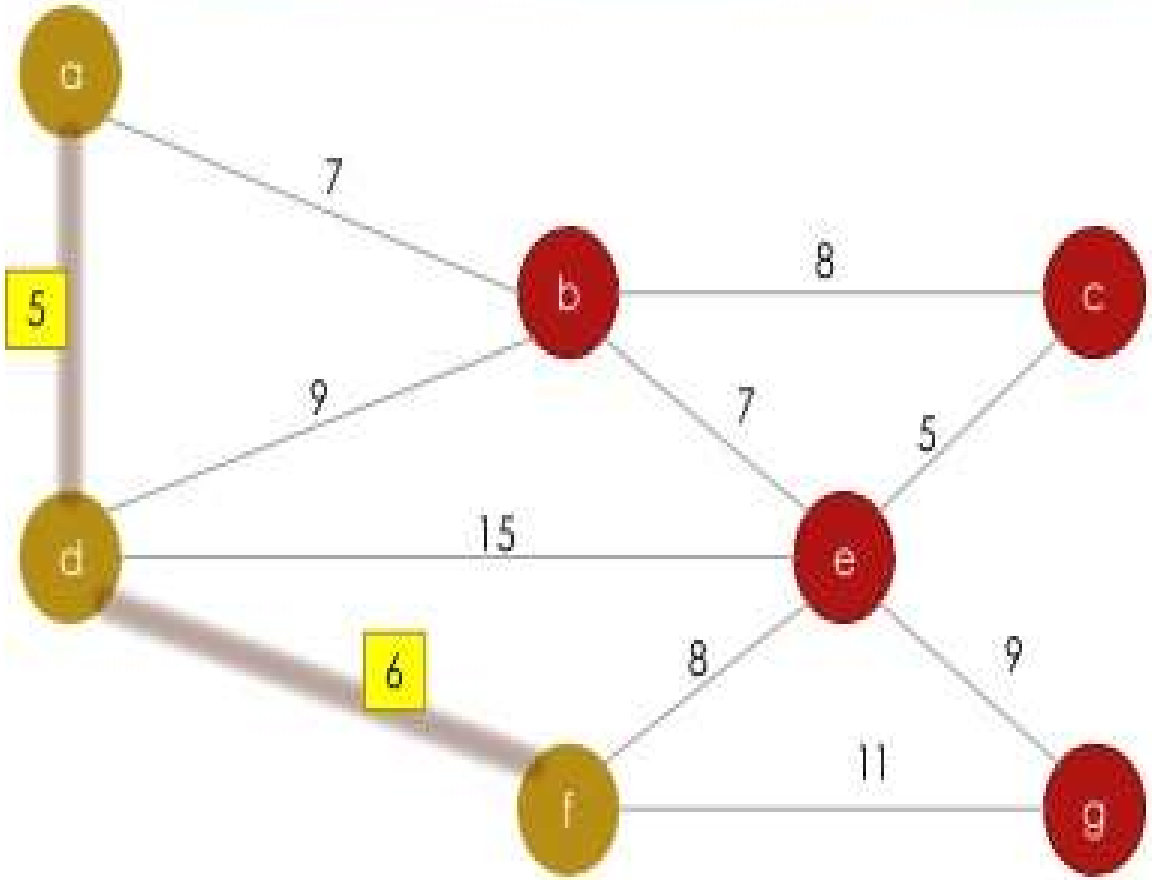
ARBOL DE MINIMA EXPANSIÓN
METODO DE PRIM



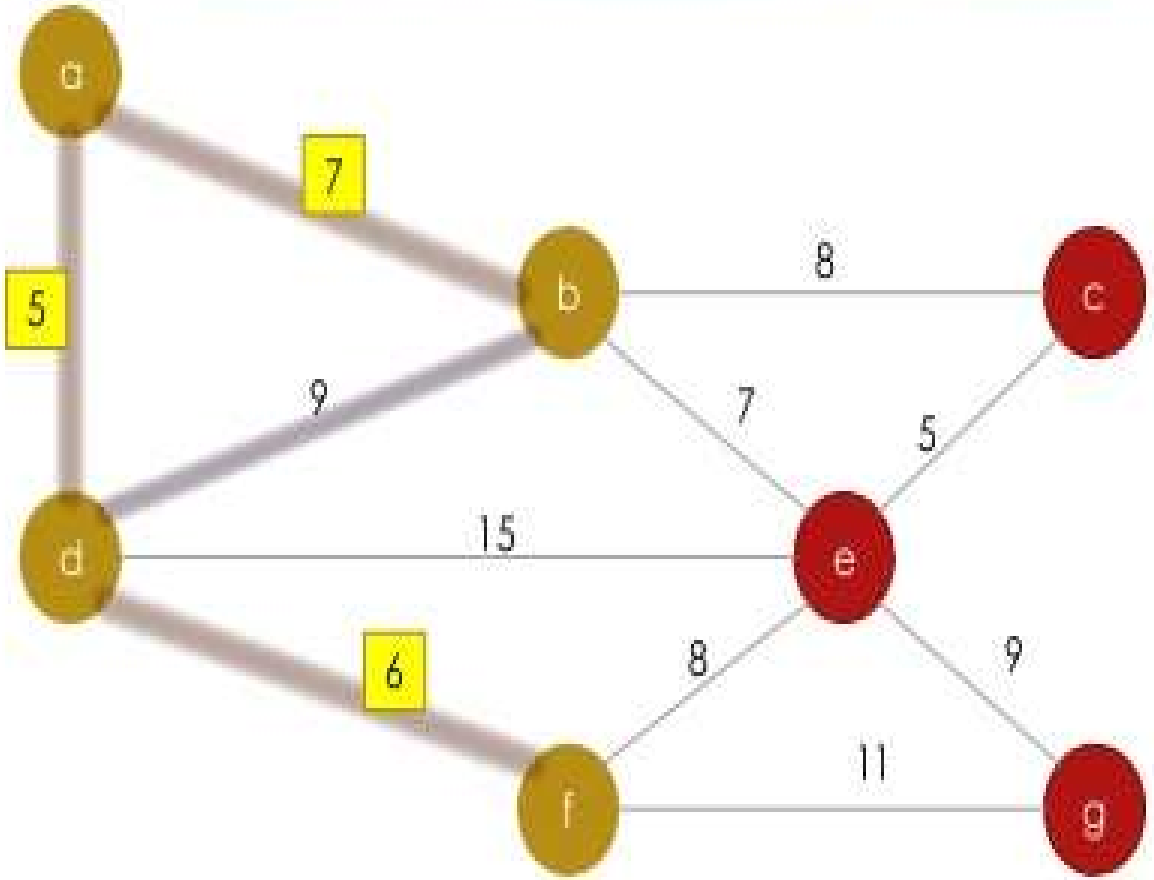
ARBOL DE MINIMA EXPANSIÓN
METODO DE PRIM



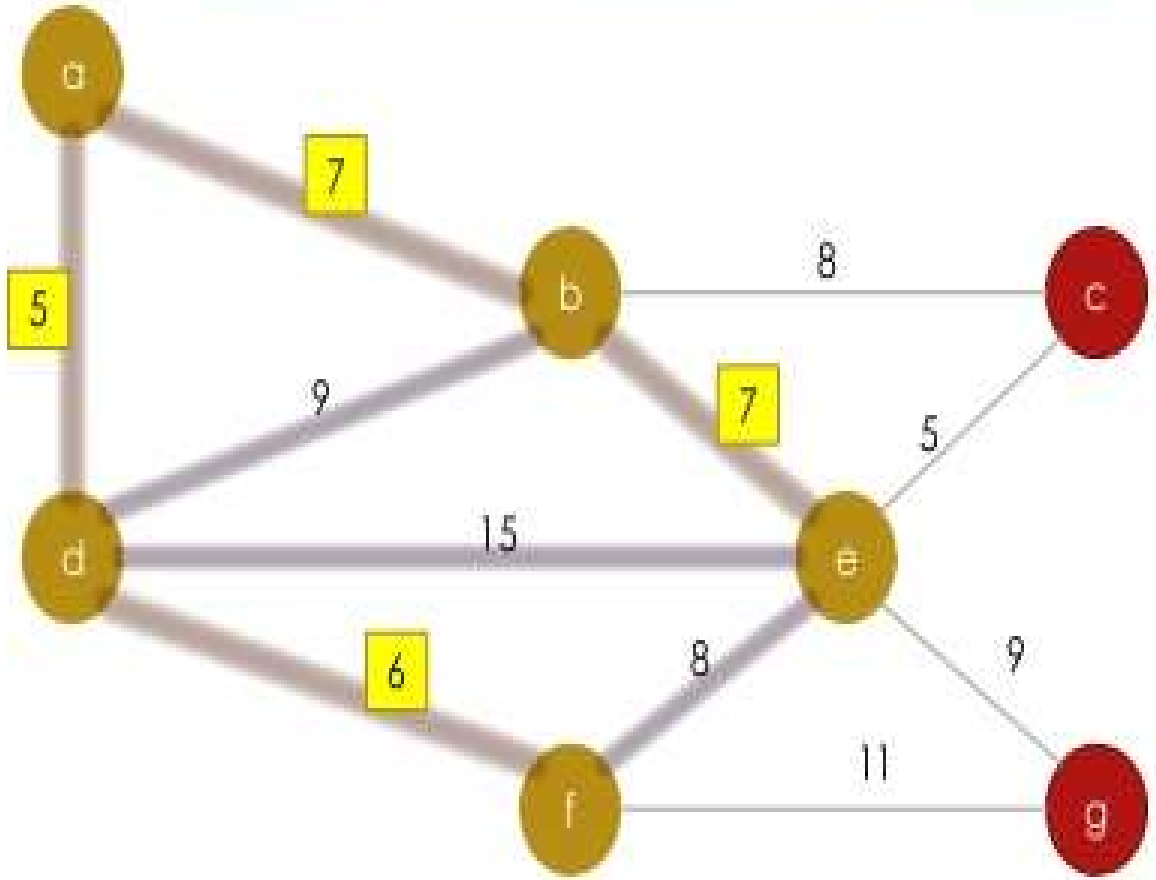
ARBOL DE MINIMA EXPANSIÓN
METODO DE PRIM



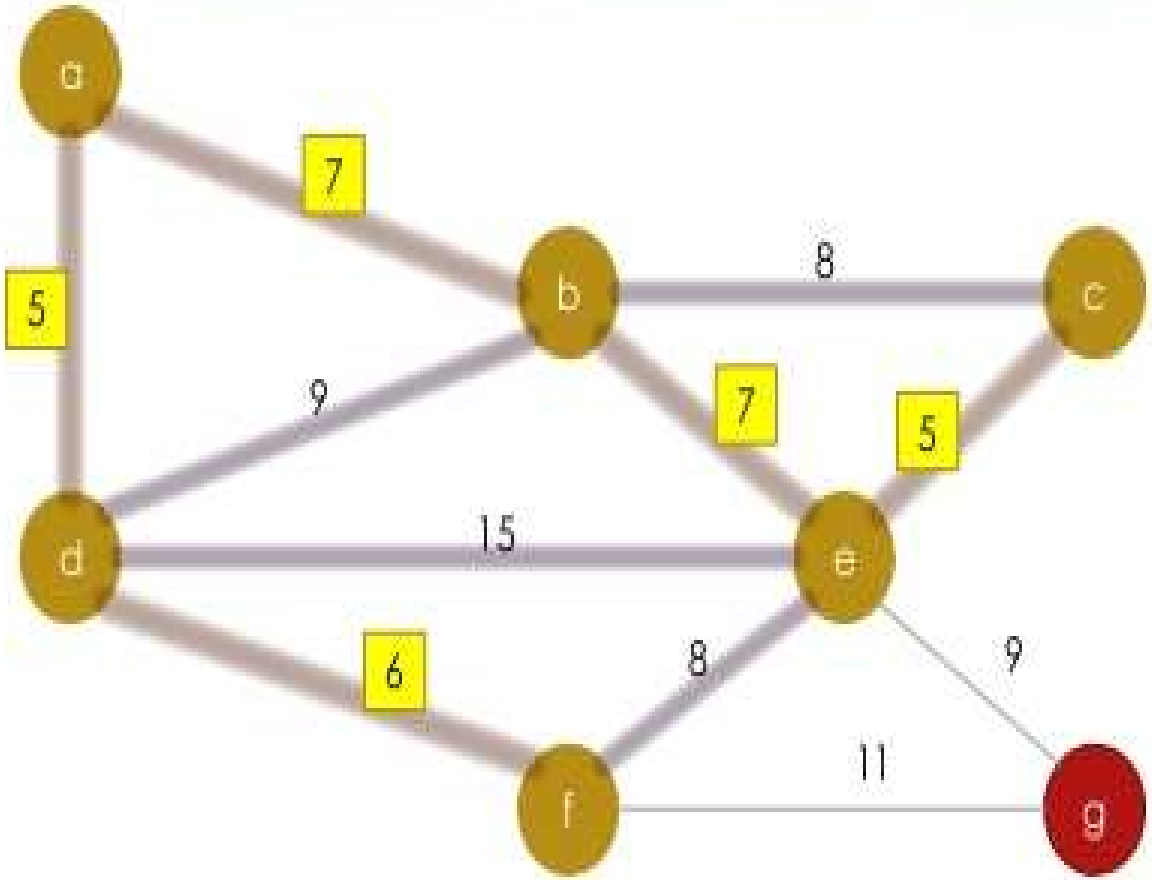
ARBOL DE MINIMA EXPANSIÓN
METODO DE PRIM



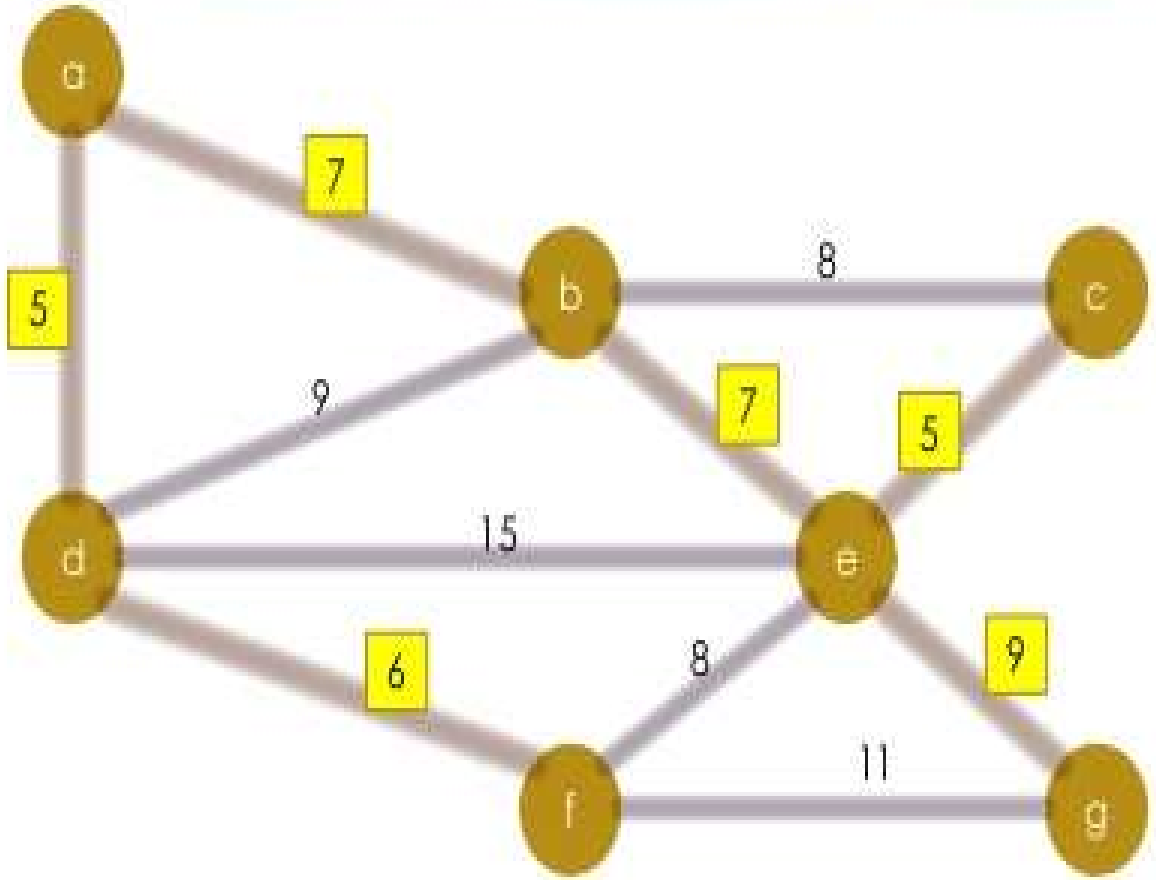
ARBOL DE MINIMA EXPANSIÓN
METODO DE PRIM



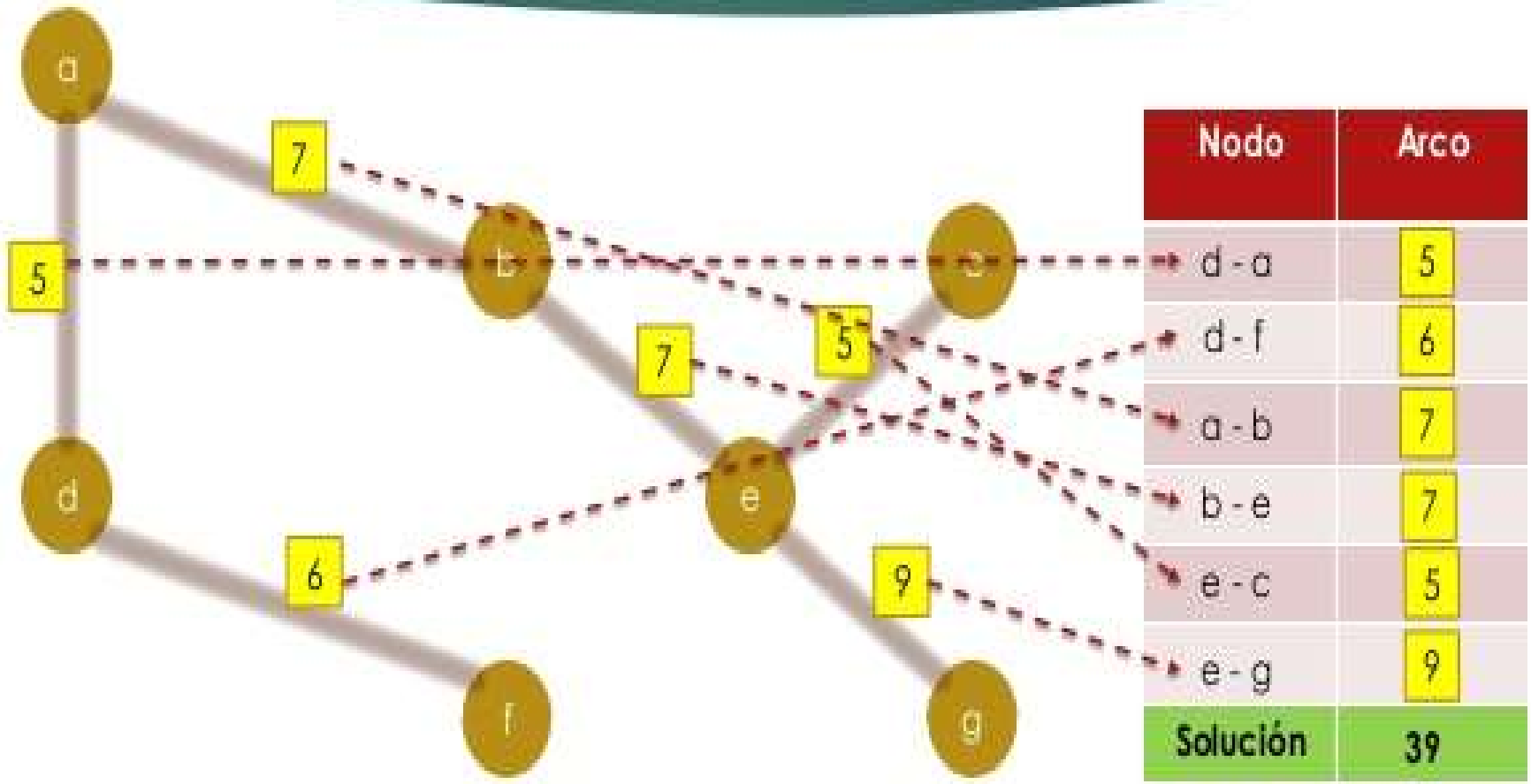
ARBOL DE MINIMA EXPANSIÓN
METODO DE PRIM



ARBOL DE MINIMA EXPANSIÓN
METODO DE PRIM



ARBOL DE MINIMA EXPANSIÓN
METODO DE PRIM



Nodo	Arco
d - a	5
d - f	6
a - b	7
b - e	7
e - c	5
e - g	9
Solución	39

```
void Prim( GRAFO G, CONJUNTO T )
{
   CONJUNTO_V  U;
   VERTICE     v;

   T = ∅;
   U = { cualquier vértice de G };
   while( U != V ) {
     Sea (u,v) es el arco de menor costo tal que
         u ∈ U  y  v ∈ V-U;
     T = T ∪ { (u,v) };
     U = U ∪ { v };
   }
}
```

En resumen …

ALGORITMO DE PRIM

- El algoritmo incrementa continuamente el tamaño de un árbol, comenzando por un vértice inicial al que se le van agregando sucesivamente vértices cuya distancia a los anteriores es mínima. Esto significa que en cada paso, las aristas a considerar son aquellas que inciden en vértices que ya pertenecen al árbol.

- El árbol recubridor mínimo está completamente construido cuando no quedan más vértices por agregar.

PROGRAMACIÓN DINÁMICA

Existe una serie de problemas cuyas soluciones pueden ser expresadas recursivamente en términos matemáticos, y posiblemente la manera más natural de resolverlos es mediante un algoritmo recursivo. Sin embargo, el tiempo de ejecución de la solución recursiva, normalmente de orden exponencial y por tanto impracticable, puede mejorarse substancialmente mediante la Programación Dinámica.

En el diseño Divide y Vencerás veíamos cómo para resolver un problema lo dividíamos en subproblemas independientes, los cuales se resolvían de manera recursiva para combinar finalmente las soluciones y así resolver el problema original. El inconveniente se presenta cuando los subproblemas obtenidos no son independientes sino que existe solapamiento entre ellos; entonces es cuando una solución recursiva no resulta eficiente por la repetición de cálculos que conlleva.

En estos casos es cuando la Programación Dinámica nos puede ofrecer una solución aceptable. La eficiencia de esta técnica consiste en resolver los subproblemas una sola vez, guardando sus soluciones en una tabla para su futura utilización.

PROGRAMACIÓN DINÁMICA

La Programación Dinámica no sólo tiene sentido aplicarla por razones de eficiencia, sino porque además presenta un método capaz de resolver de manera eficiente problemas cuya solución ha sido abordada por otras técnicas y ha fracasado.

Donde tiene mayor aplicación la Programación Dinámica es en la resolución de problemas de optimización. En este tipo de problemas se pueden presentar distintas soluciones, cada una con un valor, y lo que se desea es encontrar la solución de valor óptimo (máximo o mínimo).

Principio de óptimo enunciado por Bellman en 1957 y que dice:

"*En una secuencia de decisiones óptima toda subsecuencia ha de ser también óptima".*

Hemos de observar que aunque este principio parece evidente no siempre es aplicable y por tanto es necesario verificar que se cumple para el problema en cuestión.

PROGRAMACIÓN DINÁMICA

Para que un problema pueda ser abordado por esta técnica ha de cumplir dos condiciones:

- La solución al problema ha de ser alcanzada a través de una secuencia de decisiones, una en cada etapa.
- Dicha secuencia de decisiones ha de cumplir el principio de óptimo.

En grandes líneas, el diseño de un algoritmo de Programación Dinámica consta de los siguientes pasos:

1. Planteamiento de la solución como una sucesión de decisiones y verificación de que ésta cumple el principio de óptimo.
2. Definición recursiva de la solución.
3. Cálculo del valor de la solución óptima mediante una tabla en donde se almacenan soluciones a problemas parciales para reutilizar los cálculos.
4. Construcción de la solución óptima haciendo uso de la información contenida en la tabla anterior.

PROGRAMACIÓN DINÁMICA

Ej.: **CÁLCULO DE LOS NÚMEROS DE FIBONACCI**

Dicha sucesión podemos expresarla recursivamente en términos matemáticos de la siguiente manera:

$$Fib(n) = \begin{cases} 1 & \text{si } n = 0,1 \\ Fib(n-1) + Fib(n-2) & \text{si } n > 1 \end{cases}$$

Por tanto, la forma más natural de calcular los términos de esa sucesión es mediante un programa recursivo:

```
PROCEDURE FibRec(n:CARDINAL):CARDINAL;
BEGIN
      IF n<=1 THEN RETURN 1
      ELSE
         RETURN FibRec(n-1) + FibRec(n-2)
      END
END FibRec;
```

PROGRAMACIÓN DINÁMICA

Ej.: **CÁLCULO DE LOS NÚMEROS DE FIBONACCI**

El inconveniente es que el algoritmo resultante es poco eficiente ya que su tiempo de ejecución es de orden exponencial. Como podemos observar, la falta de eficiencia del algoritmo se debe a que se producen llamadas recursivas repetidas para calcular valores de la sucesión, que habiéndose calculado previamente, no se conserva el resultado.

Para este problema es posible diseñar un algoritmo que en tiempo lineal lo resuelva mediante la construcción de una tabla que permita ir almacenando los cálculos realizados hasta el momento para poder reutilizarlos: necesario volver a calcular cada vez

Fib(0)	*Fib*(1)	*Fib*(2)	...	*Fib*(*n*)

```
TYPE TABLA = ARRAY [0..n] OF CARDINAL
PROCEDURE FibIter(VAR T:TABLA;n:CARDINAL):CARDINAL;
   VAR i:CARDINAL;
BEGIN
   IF n<=1 THEN RETURN 1
   ELSE
      T[0]:=1;
      T[1]:=1:
      FOR i:=2 TO n DO
         T[i]:=T[i-1]+T[i-2]
      END;
      RETURN T[n]
   END
END FibIter;
```

PROGRAMACIÓN DINÁMICA

Existe aún otra mejora a este algoritmo, que aparece al fijarnos que únicamente son necesarios los dos últimos valores calculados para determinar cada término, lo que permite eliminar la tabla entera y quedarnos solamente con dos variables para almacenar los dos últimos términos:

```
PROCEDURE FibIter2(n: CARDINAL):CARDINAL;
  VAR i,suma,x,y:CARDINAL; (* x e y son los 2 ultimos terminos *)
BEGIN
  IF n<=1 THEN RETURN 1
  ELSE
    x:=1; y:=1;
    FOR i:=2 TO n DO
      suma:=x+y; y:=x; x:=suma;
    END;
    RETURN suma
  END
END FibIter2;
```

Aunque esta función sea de la misma complejidad temporal que la anterior (lineal), consigue una complejidad espacial menor, pues de ser de orden O(*n) pasa* a ser O(1) ya que hemos eliminado la tabla.

El uso de estructuras (vectores o tablas) para eliminar la repetición de los cálculos, pieza clave de los algoritmos de Programación Dinámica, hace que en nos fijemos no sólo en la complejidad temporal de los algoritmos estudiados, sino también en su complejidad espacial.

En general, los algoritmos obtenidos mediante la aplicación de esta técnica consiguen tener complejidades (espacio y tiempo) bastante razonables, pero debemos evitar que el tratar de obtener una complejidad temporal de orden polinómico conduzca a una complejidad espacial demasiado elevada

PROGRAMACIÓN DINÁMICA

CÁLCULO DE LOS COEFICIENTES BINOMIALES

LA SUBSECUENCIA COMÚN MÁXIMA

INTERESES BANCARIOS

EL VIAJE MÁS BARATO POR RÍO

TRANSFORMACIÓN DE CADENAS

LA FUNCIÓN DE ACKERMANN

EL PROBLEMA DEL CAMBIO

EL ALGORITMO DE DIJKSTRA

EL ALGORITMO DE FLOYD

EL ALGORITMO DE WARSHALL

ORDENACIONES DE OBJETOS ENTRE DOS RELACIONES

EL VIAJANTE DE COMERCIO

HORARIOS DE TRENES

LA MOCHILA

LA MULTIPLICACIÓN ÓPTIMA DE MATRICES

VUELTA ATRÁS

Dentro de las técnicas de diseño de algoritmos, el método de Vuelta Atrás (del inglés *Backtracking) es uno de los de más ámplia utilización, en el sentido de que* puede aplicarse en la resolución de un gran número de problemas, muy especialmente en aquellos de optimización.

Los algoritmos Ávidos se va construyendo la solución por etapas, siempre avanzando sobre la solución parcial previamente calculada;

La Programación Dinámica para dar una expresión recursiva de la solución si se verifica el principio de óptimo, y luego calcularla eficientemente.

Sin embargo ciertos problemas no son susceptibles de solucionarse con ninguna de estas técnicas, de manera que la única forma de resolverlos es a través de un estudio exhaustivo de un conjunto conocido a priori de posible soluciones, en las que tratamos de encontrar una o todas las soluciones y por tanto también la óptima.

VUELTA ATRÁS

El diseño Vuelta Atrás proporciona una manera sistemática de generar todas las posibles soluciones siempre que dichas soluciones sean susceptibles de resolverse en etapas.

En su forma básica la Vuelta Atrás se asemeja a un recorrido en profundidad dentro de un árbol cuya existencia sólo es implícita, y que denominaremos *árbol de expansión. Este árbol es conceptual y sólo haremos uso de su organización como* tal, en donde cada nodo de nivel *k representa una parte de la solución y está* formado por *k etapas que se suponen ya realizadas. Sus hijos son las* prolongaciones posibles al añadir una nueva etapa.

Para examinar el conjunto de posibles soluciones es suficiente recorrer este árbol construyendo soluciones parciales a medida que se avanza en el recorrido.

En este recorrido pueden suceder dos cosas. La primera es que tenga éxito si, procediendo de esta manera, se llega a una solución (una hoja del árbol). Si lo único que buscábamos era una solución al problema, el algoritmo finaliza aquí; ahora bien, si lo que buscábamos eran todas las soluciones o la mejor de entre todas ellas, el algoritmo seguirá explorando el árbol en búsqueda de soluciones alternativas.

VUELTA ATRÁS

El recorrido no tiene éxito si en alguna etapa la solución parcial construida hasta el momento no se puede completar; nos encontramos en lo que llamamos *nodos fracaso.*

En tal caso, el algoritmo vuelve atrás (y de ahí su nombre) en su recorrido eliminando los elementos que se hubieran añadido en cada etapa a partir de ese nodo.

En este retroceso, si existe uno o más caminos aún no explorados que puedan conducir a solución, el recorrido del árbol continúa por ellos.

La filosofía de estos algoritmos no sigue unas reglas fijas en la búsqueda de las soluciones. Podríamos hablar de un proceso de prueba y error en el cual se va trabajando por etapas construyendo gradualmente una solución. Para muchos problemas esta prueba en cada etapa crece de una manera exponencial, lo cual es necesario evitar.

VUELTA ATRÁS

Gran parte de la eficiencia de un algoritmo de Vuelta Atrás proviene de considerar el menor conjunto de nodos que puedan llegar a ser soluciones, aunque siempre asegurándonos de que el árbol "podado" siga conteniendo todas las soluciones.

Por otra parte debemos tener cuidado a la hora de decidir el tipo de condiciones (*restricciones) que comprobamos en cada nodo a fin* de detectar nodos fracaso. Evidentemente el análisis de estas restricciones permite ahorrar tiempo, al delimitar el tamaño del árbol a explorar.

Sin embargo esta evaluación requiere a su vez tiempo extra, de manera que aquellas restricciones que vayan a detectar pocos nodos fracaso no serán normalmente interesantes.

Como norma de actuación general, las restricciones sencillas son siempre apropiadas, mientras que las más sofisticadas que requieren más tiempo en su cálculo deberían reservarse para situaciones en las que el árbol que se genera sea muy grande.

VUELTA ATRÁS

Como se lleva a cabo la búsqueda de soluciones trabajando sobre este árbol y su recorrido:

En líneas generales, un problema puede resolverse con un algoritmo Vuelta Atrás cuando la solución puede expresarse como una *n-tupla* [*x1, x2, ..., xn] donde cada una de las componentes xi de este vector es elegida en* cada etapa de entre un conjunto finito de valores.

Cada etapa representará un nivel en el árbol de expansión.

En primer lugar debemos fijar la descomposición en etapas que vamos a realizar y definir, dependiendo del problema, la *n-tupla que representa la solución del* problema y el significado de sus componentes *xi.*

Una vez que veamos las posibles opciones de cada etapa quedará definida la estructura del árbol a recorrer.

VUELTA ATRÁS

Ej.: LAS *n REINAS*

Disponemos de un tablero de ajedrez de tamaño 8x8, y se trata de colocar en él ocho reinas de manera que no se amenacen según las normas del ajedrez, es decir, que no se encuentren dos reinas ni en la misma fila, ni en la misma columna, ni en la misma diagonal.

Numeramos las reinas del 1 al 8. Cualquier solución a este problema estará representada por una 8-tupla [*x1,x2,x3,x4,x5,x6,x7,x8] en la que cada xi representa la* columna donde la reina de la fila *i-ésima es colocada. Una posible solución al* problema es la tupla [4,6,8,2,7,1,3,5].

VUELTA ATRÁS

Para decidir en cada etapa cuáles son los valores que puede tomar cada uno de los elementos *xi hemos de tener en cuenta lo que hemos denominado restricciones a* fin de que el número de opciones en cada etapa sea el menor posible.

- *Restricciones explícitas. Formadas por reglas que restringen los valores que* pueden tomar los elementos *xi a un conjunto determinado. En nuestro problema* este conjunto es

$$S = \{1,2,3,4,5,6,7,8\}$$

- *Restricciones implícitas. Indican la relación existente entre los posibles valores* de los *xi para que éstos puedan formar parte de una n-tupla solución.*

  Dos restricciones implícitas.

  a) sabemos que dos reinas no pueden situarse en la misma columna y por tanto no puede haber dos *xi iguales (obsérvese además que la propia definición* de la tupla impide situar a dos reinas en la misma fila, con lo cual tenemos cubiertos los dos casos, el de las filas y el de las columnas).

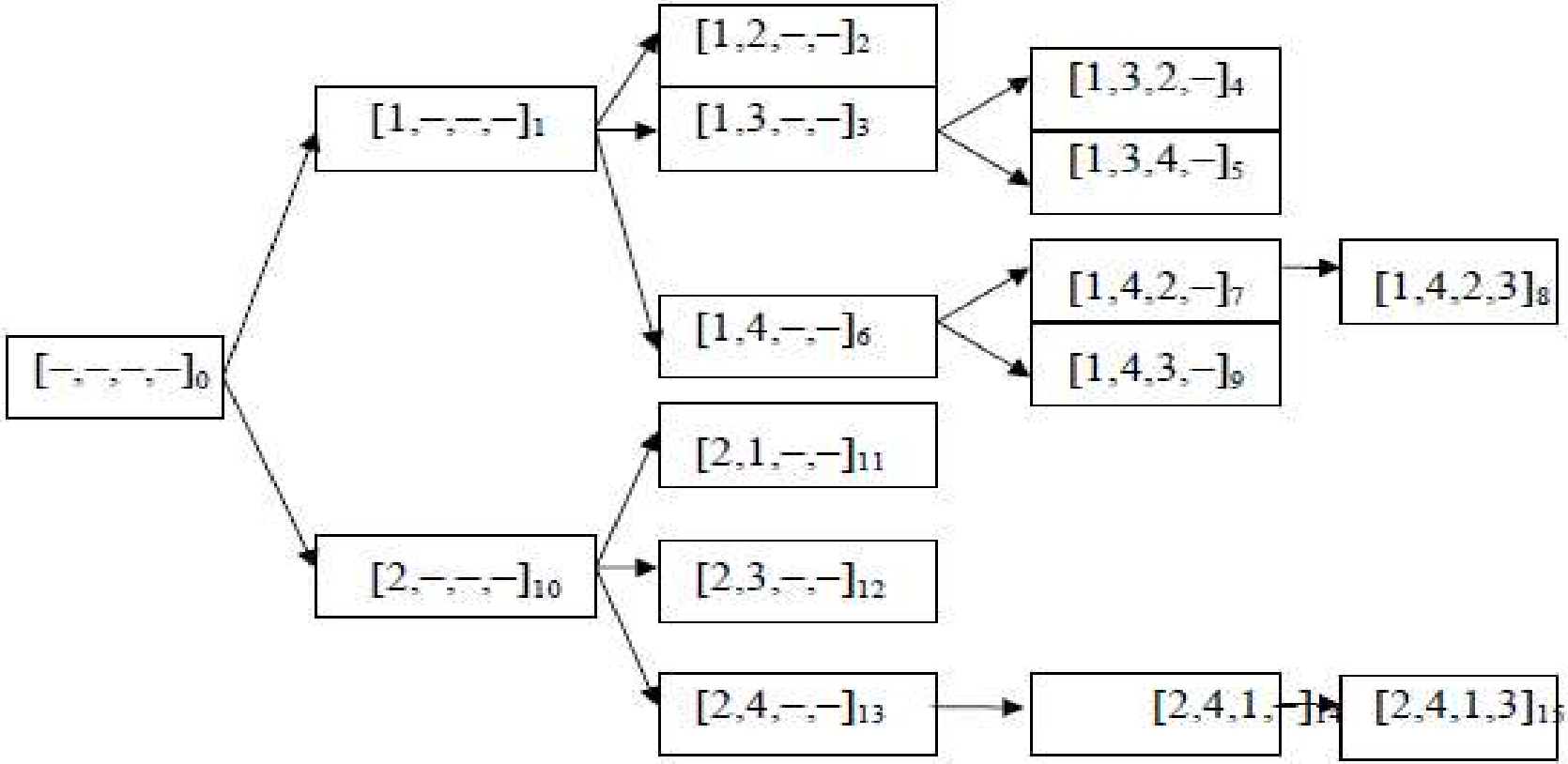
  b) sabemos que dos reinas no pueden estar en la misma diagonal. Esta condición se refleja en la segunda restricción implicita que, en forma de ecuación, puede ser expresada como :

$|x - x'| \neq |y - y'|$, *siendo (x,y) y (x',y') las coordenadas de dos reinas en el* tablero

VUELTA ATRÁS

De esta manera, y aplicando las restricciones, en cada etapa *k iremos generando* sólo las *k-tuplas con posibilidad de solución. A los prefijos de longitud k de la ntupla* solución que vamos construyendo y que verifiquen las restricciones expuestas los denominaremos *k-prometedores, pues a priori pueden llevarnos a la solución* buscada. Obsérvese que todo nodo generado es o bien fracaso o bien *k-prometedor.*

Con estas condiciones queda definida la estructura del árbol de expansión, que representamos a continuación para un tablero 4x4:



VUELTA ATRÁS

En resumen …

- Podemos decir que Vuelta Atrás es un método exhaustivo de tanteo (prueba y error) que se caracteriza por un avance progresivo en la búsqueda de una solución mediante una serie de etapas.
- En dichas etapas se presentan unas opciones cuya validez ha de examinarse con objeto de seleccionar una de ellas para proseguir con el siguiente paso.
- Este comportamiento supone la generación de un árbol y su examen y eventual poda hasta llegar a una solución o a determinar su imposibilidad.
- Este avance se puede detener cuando se alcanza una solución, o bien si se llega a una situación en que ninguna de las soluciones es válida;
  - En este caso se vuelve al paso anterior, lo que supone que deben recordarse las elecciones hechas en cada paso para poder probar otra opción aún no examinada.
- Este retroceso (vuelta atrás) puede continuar si no quedan opciones que examinar hasta llegar a la primera etapa.
- El agotamiento de todas las opciones de la primera etapa supondrá que no hay solución posible pues se habrán examinado todas las posibilidades.
- El hecho de que la solución sea encontrada a través de ir añadiendo elementos a la solución parcial, y que el diseño Vuelta Atrás consista básicamente en recorrer un árbol hace que el uso de recursión sea muy apropiado.

VUELTA ATRÁS

Se observa que están presentes tres elementos principales.

En primer lugar hay una generación de descendientes, en donde para cada nodo generamos sus descendientes con posibilidad de solución. A este paso se le denomina expansión, ramificación o bifurcación. A continuación, y para cada uno de estos descendientes, hemos de aplicar lo que denominamos prueba de fracaso (segundo elemento). Finalmente, caso de que sea aceptable este nodo, aplicaremos la prueba de solución (tercer elemento) que comprueba si el nodo que es posible solución efectivamente lo es.

Tal vez lo más difícil de ver en este esquema es donde se realiza la vuelta atrás, y para ello hemos de pensar en la propia recursión y su mecanismo de funcionamiento, que es la que permite ir recorriendo el árbol en profundidad.

```
PROCEDURE VueltaAtras(etapa);
BEGIN
   IniciarOpciones;
   REPEAT
      SeleccionarNuevaOpcion;
      IF Aceptable THEN
         AnotarOpcion;
         IF SolucionIncompleta THEN
            VueltaAtras(etapa_siguiente);
            IF NOT exito THEN
               CancelarAnotacion
            END
         ELSE (* solucion completa *)
            exito:=TRUE
         END
      END
   UNTIL (exito) OR (UltimaOpcion)
END VueltaAtras;
```

VUELTA ATRÁS

```
CONST n = ...; (* numero de reinas; n>3 *)
TYPE SOLUCION = ARRAY[1..n] OF CARDINAL;
VAR X:SOLUCION; exito:BOOLEAN;

PROCEDURE Reinas(k: CARDINAL);

 (* encuentra una manera de disponer las n reinas *)
 BEGIN
    IF k>n THEN RETURN END;
    X[k]:=0;
    REPEAT
       INC(X[k]); (* seleccion de nueva opcion *)
       IF Valido(k) THEN (* prueba de fracaso *)
          IF k<>n THEN
             Reinas(k+1) (* llamada recursiva *)
          ELSE
             exito:=TRUE
          END
       END
    UNTIL (X[k]=n) OR exito;
 END Reinas;
```

La función *Valido* es la que comprueba las restricciones implícitas, realizando la prueba de fracaso:

```
PROCEDURE Valido(k:CARDINAL):BOOLEAN;
(* comprueba si el vector solucion X construido hasta el paso k
   es k-prometedor, es decir, si la reina puede situarse en la
   columna k *)
   VAR i: CARDINAL;
BEGIN
   FOR i:=1 TO k-1 DO
      IF (X[i]=X[k]) OR (ValAbs(X[i],X[k])=ValAbs(i,k)) THEN
         RETURN FALSE
      END
   END;
   RETURN TRUE
END Valido;
```

Utilizamos la funcion *ValAbs*(x,y), que es la que devuelve $|x-y|$:

```
PROCEDURE ValAbs(x,y:CARDINAL):CARDINAL;
BEGIN
   IF x>y THEN RETURN x-y ELSE RETURN y-x END;
END ValAbs;
```

VUELTA ATRÁS

La complejidad es exponencial por la forma en la que se busca la solución mediante el recorrido en profundidad del árbol. De esta forma estos algoritmos van a ser de un orden de complejidad al menos del número de nodos del árbol que se generen y este número, si no se utilizan restricciones, es de orden de *zn* donde *z son las posibles opciones que existen en cada etapa, y n el número de* etapas que es necesario recorrer hasta construir la solución (esto es, la profundidad del árbol o la longitud de la *n-tupla solución).*

El uso de restricciones, tanto implícitas como explícitas, trata de reducir este número tanto como sea posible (en el ejemplo de las reinas se pasa de 88 nodos si no se usa ninguna restricción a poco más de 2000), pero sin embargo en muchos casos no son suficientes para conseguir algoritmos "tratables", es decir, que sus tiempos de ejecución sean de orden de complejidad razonable.

Para aquellos problemas en donde se busca una solución y no todas, es donde entra en juego la posibilidad de considerar distintas formas de ir generando los nodos del árbol. Y como la búsqueda que realiza la Vuelta Atrás es siempre en profundidad, para lograr esto sólo hemos de ir variando el orden en el que se generan los descendientes de un nodo, de manera que trate de ser lo más apropiado a nuestra estrategia. (Ej. el problema del laberinto).

Algoritmo de Backtracking

```
Subprograma saltoCaballo(i,x,y,*exito)
    inicio
    éxito←0
    k←0
    1.Hacer
        k←k+1
        nx←x+d[k-1][0]
        ny←y+d[k-1][1]
        1.1 Si (nx >= 1) y (nx <= N) y (ny >=1) y (ny <=N) y (tablero[nx][ny]=0) entonces
            tablero[nx][ny]←i
            1.2 Si (i < N*N) entonces
                saltoCaballo(i+1,nx,ny,exito)
            1.3 fin_si
            1.4 Si (!exito) entonces
                tablero[nx][ny]←0
            1.5 fin_si
            1.6 Si no entonces
                éxito←1
            1.7 fin_si
    2. Mientras (k < 8) y (!exito)
    3. Fin del ciclo del paso 1.
    fin subprograma saltoCaballo
```

VUELTA ATRÁS

- **RECORRIDOS DEL REY DE AJEDREZ**
- **LAS PAREJAS ESTABLES**
- **EL LABERINTO**
- **LA ASIGNACIÓN DE TAREAS**
- **LA MOCHILA (0,1)**
- **LOS SUBCONJUNTOS DE SUMA DADA**
- **CICLOS HAMILTONIANOS. EL VIAJANTE DE COMERCIO**
- **EL CONTINENTAL**
- **HORARIOS DE TRENES**
- **LA ASIGNACIÓN DE TAREAS EN PARALELO**
- **EL COLOREADO DE MAPAS**
- **RECONOCIMIENTO DE GRAFOS**
- **SUBCONJUNTOS DE IGUAL SUMA**
- **LAS MÚLTIPLES MOCHILAS**

RAMIFICACIÓN Y PODA

Es una variante del diseño Vuelta Atrás estudiado.

Su nombre en castellano proviene del término inglés *Branch and Bound, se aplica normalmente para resolver problemas de* optimización.

Ramificación y Poda, al igual que el diseño Vuelta Atrás, realiza una enumeración parcial del espacio de soluciones basándose en la generación de un árbol de expansión.

Una característica que le hace diferente al de Vuelta Atrás es la posibilidad de generar nodos siguiendo distintas estrategias.

Recordemos que el diseño Vuelta Atrás realiza la generación de descendientes de una manera sistemática y de la misma forma para todos los problemas, haciendo un recorrido en profundidad del árbol que representa el espacio de soluciones.

El diseño Ramificación y Poda en su versión más sencilla puede seguir un recorrido en anchura (estrategia LIFO) o en profundidad (estrategia FIFO), o utilizando el cálculo de funciones de coste para seleccionar el nodo que en principio parece más prometedor (estrategia de mínimo coste o LC).

RAMIFICACIÓN Y PODA

Ramificación y Poda utiliza cotas para podar aquellas ramas del árbol que no conducen a la solución óptima.

Para ello calcula en cada nodo una cota del posible valor de aquellas soluciones alcanzables desde ése. Si la cota muestra que cualquiera de estas soluciones tiene que ser necesariamente peor que la mejor solución hallada hasta el momento no necesitamos seguir explorando por esa rama del árbol, lo que permite realizar el proceso de poda.

Definimos *nodo vivo del árbol a un nodo con posibilidades de ser ramificado, es* decir, que no ha sido podado. Para determinar en cada momento que nodo va a ser expandido y dependiendo de la estrategia de búsqueda seleccionada, necesitaremos almacenar todos los nodos vivos en alguna estructura que podamos recorrer.

Emplearemos una pila para almacenar los nodos que se han generado pero no han sido examinados en una búsqueda en profundidad (LIFO). Las búsquedas en amplitud utilizan una cola (FIFO) para almacenar los nodos vivos de tal manera que van explorando nodos en el mismo orden en que son creados.

RAMIFICACIÓN Y PODA

Básicamente, en un algoritmo de Ramificación y Poda se realizan tres etapas.

a) *Selección*, se encarga de extraer un nodo de entre el conjunto de los nodos vivos. La forma de escogerlo va a depender directamente de la estrategia de búsqueda que decidamos para el algoritmo.
b) *Ramificación*, se construyen los posibles nodos hijos del nodo seleccionado en el paso anterior.
c) *Poda*, en la que se eliminan algunos de los nodos creados en la etapa anterior. Esto contribuye a disminuir en lo posible el espacio de búsqueda y así atenuar la complejidad de estos algoritmos basados en la exploración de un árbol de posibilidades.

Aquellos nodos no podados pasan a formar parte del conjunto de nodos vivos, y se comienza de nuevo por el proceso de selección.

El algoritmo finaliza cuando encuentra la solución, o bien cuando se agota el conjunto de nodos vivos.

RAMIFICACIÓN Y PODA

Para cada nodo del árbol dispondremos de una función de coste que nos estime el valor óptimo de la solución si continuáramos por ese camino.

De esta manera, si la cota que se obtiene para un nodo, que por su propia construcción deberá ser mejor que la solución real (o a lo sumo, igual que ella), es peor que una solución ya obtenida por otra rama, podemos podar esa rama.

Evidentemente no podremos realizar ninguna poda hasta que hayamos encontrado alguna solución.

Las funciones de coste han de ser crecientes respecto a la profundidad del árbol, es decir, si h es una función de coste entonces $h(n) \leq h(n')$ para todo n' nodo descendiente de n.

Lo que le da valor a esta técnica es la posibilidad de disponer de distintas estrategias de exploración del árbol y de acotar la búsqueda de la solución (eficiencia).

La dificultad está en encontrar una buena función de coste para el problema, buena en el sentido de que garantice la poda y que su cálculo no sea muy costoso.

Si es demasiado simple pocas ramas puedan ser excluidas.

RAMIFICACIÓN Y PODA

Inicialmente (antes de proceder a la poda) se tiene que disponer del coste de la mejor solución encontrada hasta el momento que permite excluir de futuras expansiones cualquier solución parcial con un coste mayor.

Como muchas veces no se desea esperar a encontrar la primera solución para empezar a podar, un buen recurso para los problemas de optimización es tomar como mejor solución inicial la obtenida con un algoritmo ávido, que como vimos no encuentra siempre la solución óptima, pero sí una cercana a la óptima.

Ventajas: La posibilidad de ejecutarlos en paralelo. (Debido a que disponen de un conjunto de nodos vivos sobre el que se efectúan las tres etapas del algoritmo antes mencionadas, nada impide tener más de un proceso trabajando sobre este conjunto, extrayendo nodos, expandiéndolos y realizando la poda).

RAMIFICACIÓN Y PODA

CONSIDERACIONES DE IMPLEMENTACIÓN

Estructura general de implementación, basada en tres módulos principales:

1. Módulo que contiene el esquema de funcionamiento general de este tipo de algoritmos.

2. Modulo que maneja la estructura de datos en donde se almacenan los nodos que se van generando, y que puede tratarse de una pila, una cola o un montículo (según se siga una estrategia LIFO, FIFO o LC).

3. Módulo que describe e implementa las estructuras de datos que conforman los nodos.

RAMIFICACIÓN Y PODA

Ej. De Aplicación de los algoritmos de ramificación y poda

EL PUZZLE (*n*2–1)

EL VIAJANTE DE COMERCIO

EL LABERINTO

LA COLOCACIÓN ÓPTIMA DE RECTÁNGULOS

LA MOCHILA (0,1)

LA ASIGNACIÓN DE TAREAS

LAS *n* REINAS

EL FONTANERO CON PENALIZACIONES

Bibliografía

- Análisis y diseño de Algoritmos: Un enfoque practico. Jose Ignacio Pelaez. Editorial UMA. 2003.