



Algoritmos y Estructuras de Datos II

Clase 15

Carreras:

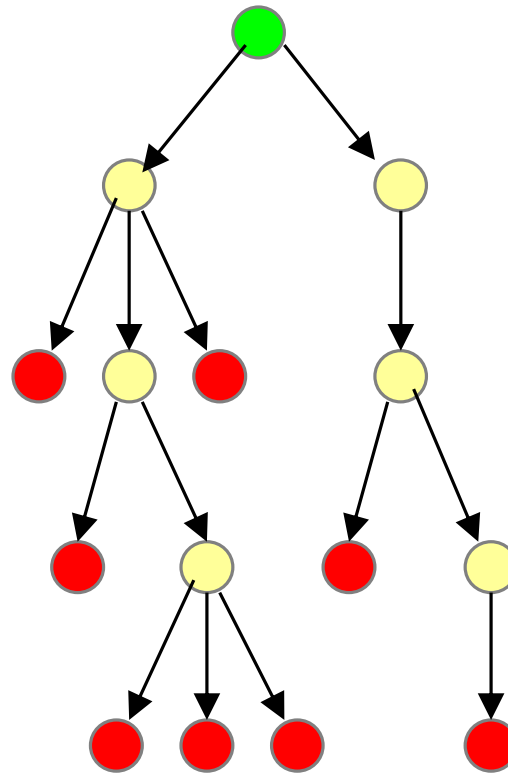
Licenciatura en Informática

Ingeniería en Informática

2024

Unidad III

Técnicas de diseño de algoritmos



Vuelta Atrás (1) (en inglés: ***Backtracking***) 2

Vuelta Atrás

El método de *Vuelta Atrás* es uno de los de más amplia utilización, en el sentido de que puede aplicarse en la resolución de un gran número de problemas, muy especialmente en aquellos de ***optimización***.

Ciertos problemas no encuentran solución con ninguna de las técnicas vistas hasta ahora, de manera que la única forma de resolverlos es a través de un estudio exhaustivo de un conjunto conocido a priori de posibles soluciones, entre las que se trata de encontrar una o todas las soluciones y por tanto también la solución óptima.

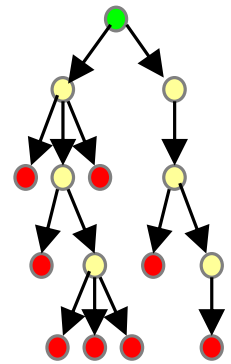
Para llevar a cabo este estudio exhaustivo, el diseño Vuelta Atrás proporciona *una manera sistemática de generar todas las posibles soluciones* siempre que dichas soluciones puedan resolverse en etapas.

Se puede aplicar vuelta atrás para la solución de numerosos Problemas:

- La mochila.
- Las n reinas en un tablero de ajedrez.
- Encontrar la Salida de un laberinto.
- Encontrar subconjuntos de suma fija en un conjunto de números.
- Descomponer un conjunto de números en 2 subconjuntos disjuntos de igual suma.
- Correspondencia.
- Los matrimonios estables.
- Rompecabezas.
- Generar las permutaciones de n números.
- Asignación de tareas con costo mínimo.
- Agente viajero.
- Coloreado en un mapa.
- Satisfactibilidad.
- Juego del Sudoku.

- 5

Vuelta Atrás



En este recorrido pueden suceder dos cosas:

- Que **tenga éxito** si se llega a una solución (una hoja del árbol).
 - Si lo único que se busca es una solución al problema, el algoritmo finaliza aquí.
 - Si se buscan todas las soluciones o la mejor de entre todas ellas, el algoritmo seguirá explorando el árbol en búsqueda de soluciones alternativas.
- Que el recorrido **no tenga éxito**. Ocurre si en alguna etapa la solución parcial construida hasta el momento no se puede completar; se ha llegado a lo que se llama un **nodo no prometedor**.
 - En tal caso, el algoritmo vuelve atrás (y de allí su nombre) en su recorrido eliminando los elementos que se hubieran agregado 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.

Vuelta Atrás

Estos algoritmos no utilizan una estrategia en la búsqueda de las soluciones.

Se plantea como un *proceso de prueba y error* en el cual se va trabajando por etapas construyendo gradualmente una solución.

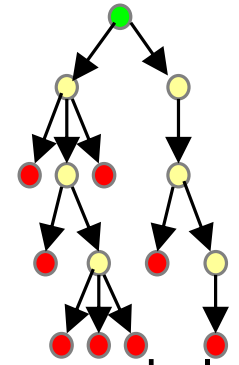
Para muchos problemas esta prueba en cada etapa crece de una manera *exponencial*.

Gran parte de la *eficiencia* (siempre relativa) de un algoritmo de Vuelta Atrás proviene de considerar el menor conjunto de nodos que puedan llegar a ser soluciones: *árbol “acotado”*.

La idea es entonces: *delimitar el tamaño del árbol* a explorar con *restricciones* a fin de detectar nodos fracaso.

- las restricciones sencillas son siempre apropiadas,
- las más sofisticadas que requieren más tiempo en su cálculo.

Vuelta Atrás



Un problema puede resolverse con un algoritmo Vuelta Atrás cuando la **solución** puede expresarse como un **vector \mathbf{X}** de **n** componentes:

$$\mathbf{X} = (x_1, x_2, \dots, x_n)$$

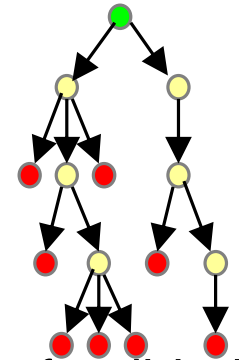
Cada componente **x_i** es elegida en las sucesivas etapas de entre un conjunto finito de valores.

Cada etapa representará un nivel en el árbol de expansión.

En primer lugar se debe fijar la descomposición en etapas que se van a realizar y definir, dependiendo del problema, el vector que representa la solución del problema y el significado de sus componentes **x_i** .

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

Vuelta Atrás



- La técnica de backtracking es un recorrido en profundidad (preorden) del árbol de expansión.
- En cada momento el algoritmo se encontrará en un cierto nivel k
- En el nivel k se tiene una solución parcial (x_1, \dots, x_{k-1})
- Se recorren los posibles valores para el elemento de la solución. Por cada valor x_k , se comprueba:
 - Si, (x_1, \dots, x_k) es prometedor entonces se genera la solución parcial (x_1, \dots, x_k) y se avanza al nivel $k+1$
 - Sino se prueban otros valores de x_k
- Si ya no existen mas valores para x_k , se retrocede (se vuelve atrás) al nivel anterior $k-1$
- El algoritmo continua hasta que la solución parcial sea una solución completa del algoritmo, o hasta que no queden mas posibilidades

Esquema de un algoritmo de Vuelta Atrás

Los algoritmos de vuelta atrás se pueden usar aun cuando las soluciones buscadas no tengan todas necesariamente la misma longitud, siguiendo el siguiente esquema:

Algoritmo **VueltaAtras**(X,k): **vector x entero**→**vector**

Entrada: X(1..k) : vector k prometedor

Salida: X(1..k+1) : vector k+1 prometedor

Si X “es una solución” entonces

 Escribir X

Sino

 Para cada vector (k+1)-prometedor w tal que $w(1..k)=X(1..k)$ hacer

VueltaAtras (w(1..k+1))

Fin

Problema de la mochila múltiple

Datos:

- Se tienen n tipos de objetos y una mochila para llevarlos.
- Del objeto de tipo i se pueden elegir tantas unidades como quiera.
- Cada objeto de tipo i tiene un peso: p_i y un beneficio asociado: b_i .
- La mochila puede cargar un peso máximo dado: M .

Objetivo: *llenar la mochila de tal manera que se maximice el beneficio de los objetos transportados, respetando la limitación de la capacidad impuesta.*

Solución : vector $X = (x_1, x_2, \dots, x_n)$

Donde x_i = cantidad de objetos de tipo i que van en la mochila

$$\text{Maximizar la cantidad: } \sum_{i=1}^n b_i x_i \quad \text{Restricción: } \sum_{i=1}^n p_i x_i \leq M$$

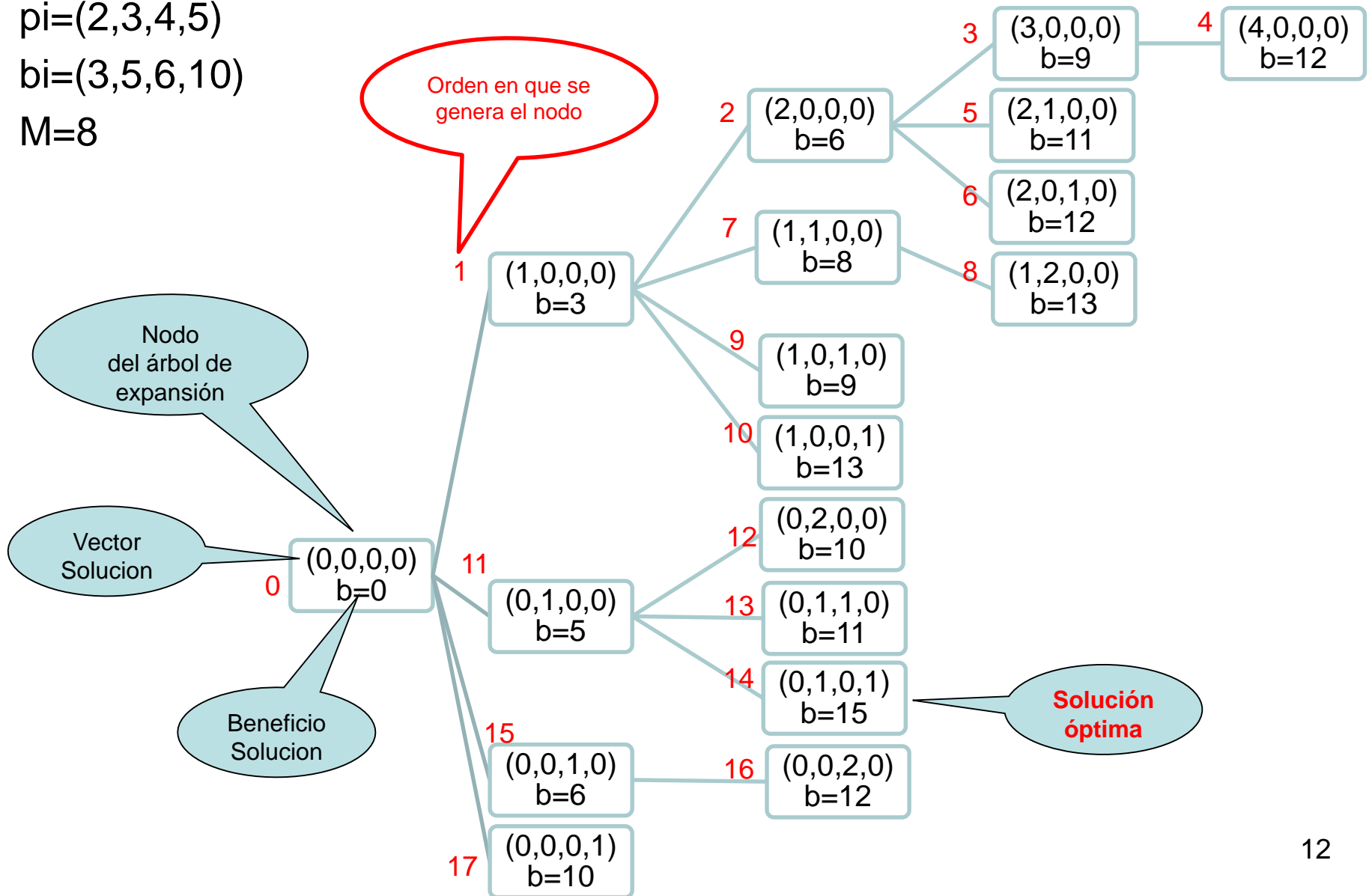
Ejemplo del Problema de la mochila múltiple:

$n=4$ tipos de objetos con:

$p_i=(2,3,4,5)$

$b_i=(3,5,6,10)$

$M=8$



Problema de la mochila

Un esquema del algoritmo de la mochila es el siguiente:

Función **mochila** (i, M): entero x peso \rightarrow beneficio

// globales: n, b y p

// entrada: elementos i a n y con peso máximo M.

// salida: bmax el beneficio de la mejor carga.

bmax \leftarrow 0

Para k=i hasta n hacer

 si $p(k) \leq M$ entonces

 bmax \leftarrow max (bmax, b(k)+**mochila**(k, M-p(k)))

retorna bmax

Fin

Examina posibilidades
dentro del nivel k

Baja un nivel en el árbol

Para obtener el valor de máximo beneficio se invoca a: **mochila(1,M)**.

Laberinto



Datos: Un laberinto cuadrado se puede representar con una matriz bidimensional $n \times n$:

Cada posición contiene un entero no negativo que indica si la celda:

- Es transitable (0)
- No es transitable (∞).

0	0	0	∞	0	0	0	0	0	0
0	0	0	0	∞	0	0	0	0	0
0	0	∞	0	0	0	∞	∞	∞	0
∞	0	∞	∞	∞	0	0	0	0	0
0	0	0	∞	0	∞	∞	0	0	∞
0	∞	0	∞	0	0	0	0	0	0
0	0	∞	0	0	0	∞	∞	∞	0
0	∞	0	∞	0	0	∞	∞	0	0
∞	0	0	∞	0	0	0	0	0	0
0	0	0	0	0	∞	0	0	0	0

Las posiciones $(1,1)$ y (n,n) corresponden a la entrada y salida del laberinto y siempre serán transitables.

Los movimientos posibles son desde una celda a la celda contigua por fila o por columna.



Laberinto

Dada una matriz con un laberinto, el problema consiste en ***diseñar un algoritmo que encuentre un camino, si es que existe, para ir de la entrada a la salida.***

Solución:

En este problema se avanzará por el laberinto en cada etapa, y cada nodo representará el camino recorrido hasta el momento.

Por la forma en la que trabaja el esquema general de Vuelta Atrás se puede usar una variable global (una matriz) para representar el laberinto e ir anotando los movimientos que ya se realizaron, indicando en cada posición el orden en el que ésta ha sido visitada.

Al producirse la vuelta atrás se liberaran las posiciones ocupadas por el nodo del que se vuelve (marcándolas de nuevo con 0).

Laberinto

Algoritmo

Entrada:

k: etapa

fil: índice de fila

col: índice de columna

Salida:





éxito e BOOLEAN

0	0	0	∞	0	0	0	0	0	0
0	0	0	0	∞	0	0	0	0	0
0	0	∞	0	0	0	∞	∞	∞	0
∞	0	∞	∞	∞	0	0	0	0	0
0	0	0	∞	0	∞	∞	0	0	∞
0	∞	0	∞	0	0	0	0	0	0
0	0	∞	0	0	0	∞	∞	∞	0
0	∞	0	∞	0	0	∞	∞	0	0
∞	0	0	∞	0	0	0	0	0	0
0	0	0	0	0	∞	0	0	0	0

Globales:

La matriz con el laberinto: Lab(1..n,1..n) e matriz nxn

Los vectores: *mfil* y *mcol* que contienen los 4 posibles movimientos:

$mfil(1) \leftarrow 1;$	$mcol(1) \leftarrow 0;$	// sur	
$mfil(2) \leftarrow 0;$	$mcol(2) \leftarrow 1;$	// este	
$mfil(3) \leftarrow 0;$	$mcol(3) \leftarrow -1;$	// oeste	
$mfil(4) \leftarrow -1;$	$mcol(4) \leftarrow 0;$	// norte	

Función **Laberinto (k,fil,col) entero x entero x entero → bool**

orden \leftarrow 0

éxito \leftarrow FALSE

Repetir

orden \leftarrow orden + 1

fil \leftarrow fil + mfil(orden)

col \leftarrow col + mcol(orden)

Si $(1 \leq \text{fil})$ AND $(\text{fil} \leq n)$ AND $(1 \leq \text{col})$ AND $(\text{col} \leq n)$ AND $(\text{lab}(\text{fil}, \text{col}) = 0)$

lab(fil,col) \leftarrow k;

Si $(\text{fil} = n)$ AND $(\text{col} = n)$ entonces

éxito \leftarrow TRUE

sino

éxito \leftarrow **Laberinto**(k+1,fil,col)

Si NOT éxito entonces

lab(fil,col) \leftarrow 0

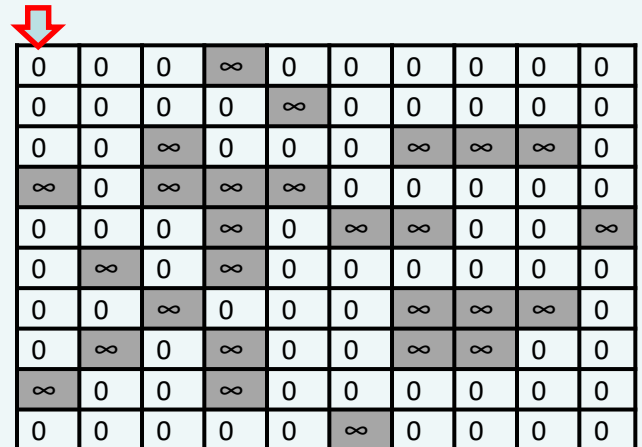
fil \leftarrow fil - mfil(orden)

col \leftarrow col - mcol(orden)

Hasta (éxito) OR (orden=4)

Retorna éxito

Fin



0	0	0	∞	0	0	0	0	0	0
0	0	0	0	∞	0	0	0	0	0
0	0	∞	0	0	0	∞	∞	∞	0
∞	0	∞	∞	∞	0	0	0	0	0
0	0	0	∞	0	∞	∞	0	0	∞
0	∞	0	∞	0	0	0	0	0	0
0	0	∞	0	0	0	∞	∞	∞	0
0	∞	0	∞	0	0	∞	∞	0	0
∞	0	0	∞	0	0	0	0	0	0
0	0	0	0	0	∞	0	0	0	0

Se invoca con **Laberinto**(1,0,1)

0	0	0	∞	0	0	0	0	0	0
0	0	0	0	∞	0	0	0	0	0
0	0	∞	0	0	0	∞	∞	∞	0
∞	0	∞	∞	∞	0	0	0	0	0
0	0	0	∞	0	∞	∞	0	0	∞
0	∞	0	∞	0	0	0	0	0	0
0	0	∞	0	0	0	∞	∞	∞	0
0	∞	0	∞	0	0	∞	∞	0	0
∞	0	0	∞	0	0	0	0	0	0
0	0	0	0	0	∞	0	0	0	0

1	0	0	∞	0	0	0	0	0	0
2	5	6	7	∞	0	0	0	0	0
3	4	∞	8	9	10	∞	∞	∞	0
∞	0	∞	∞	∞	11	12	13	0	0
0	0	0	∞	0	∞	∞	14	0	∞
0	∞	0	∞	0	0	0	15	16	17
0	0	∞	0	0	0	∞	∞	∞	18
0	∞	0	∞	0	0	∞	∞	0	19
∞	0	0	∞	0	0	0	0	0	20
0	0	0	0	0	∞	0	0	0	21



Subconjuntos

Problema: dado V un conjunto de n enteros no negativos y M un número entero positivo, el problema consiste en **diseñar un algoritmo para encontrar todos los posibles subconjuntos de V cuya suma sea exactamente M .**

Datos: se puede suponer que:

- los datos están almacenados en un vector V de enteros no negativos:
$$V = (v1, v2, ..., vn)$$
- el vector se encuentra **ordenado** de forma creciente. Sea v_i al valor del i -ésimo elemento: $v_i \leq v_{i+1}, i=1, n$

Solución : la solución al problema puede ser expresada como un vector.

$$X = (x1, x2, ..., xn)$$

- $x_i = 1$: si el entero v_i forma parte de la solución.
- $x_i = 0$: si el entero v_i **no** forma parte de la solución.

Subconjuntos

El algoritmo trabaja por etapas y en cada etapa decide si el k -ésimo entero del vector interviene o no en la solución.

- 1) En una etapa k cualquiera se puede considerar que una **condición para que pueda encontrarse solución** es que se cumpla que:

$$\sum_{i=1}^k v_i x_i + \sum_{i=k+1}^n v_i \geq M \quad (\text{Sigue})$$

En la etapa k la suma de todos los elementos que se han considerado hasta esa etapa más el valor de todos los que faltan por considerar tiene que ser al menos igual al valor M dado, porque si no es así, ni sumando todos se llega a alcanzar este valor, significa que por este camino no hay solución.

$$\sum_{i=1}^k v_i x_i + \sum_{i=k+1}^n v_i < M \quad (\text{No sigue})$$

Subconjuntos

- 2) Si al valor conseguido hasta la etapa k se le suma el siguiente elemento ($k+1$, que es el menor) y ya se supera el valor de M , esto significa que no será posible alcanzar la solución por este camino.

$$\sum_{i=1}^k v_i x_i + v_{k+1} > M \quad (\text{No sigue})$$

Para poder seguir buscando soluciones debe cumplirse entonces que:

$$\sum_{i=1}^k v_i x_i + v_{k+1} \leq M \quad (\text{Sigue})$$

Estas **dos restricciones** van a permitir reducir la búsqueda al evitar caminos que no conducen a solución.

Subconjuntos

El procedimiento que encuentra todos los posibles subconjuntos es un algoritmo exponencial llamado **Subconjuntos(s,k,r)**

En el algoritmo, se denomina con s y r a las siguientes sumas:

$$s = \sum_{i=1}^{k-1} v_i x_i \qquad r = \sum_{i=k}^n v_i$$

Se dan los valores iniciales:

$$s = 0$$

r es la suma de todos los elementos del conjunto: $r = \sum_{i=1}^n v_i$

$$x_i = -1, i = 1 \dots n$$

Y luego el algoritmo debe invocarse como: **Subconjuntos(s,1,r)**

Variables globales:

M: entero ≥ 0 // valor de la suma
n: entero ≥ 0 // numero de elementos
V(1..n): vector // el conjunto de los datos ordenados
X(1..n): vector // el vector solución

Algoritmo Subconjuntos(s,k,r): ent ≥ 0 x ent ≥ 0 x ent ≥ 0

X(k) \leftarrow 1

Si $s+V(k)=M$ entonces

Escribir("La Solucion es:", X(1..k))

Se encontró una
Solución en la
que está V(k)

sino

Si $k=n$ entonces

Escribir ("No hay solucion")

Sino

Si $s+V(k)+V(k+1) \leq M$ entonces

Subconjuntos(s+V(k),k+1,r-V(k))

Busca una
Solución en la
que está V(k)

Si $(s+r-V(k) \geq M)$ AND $(s+V(k+1) \leq M)$ entonces

X(k) \leftarrow 0

Subconjuntos(s,k+1,r-V(k))

Busca una Solución en
la que NO está V(k)

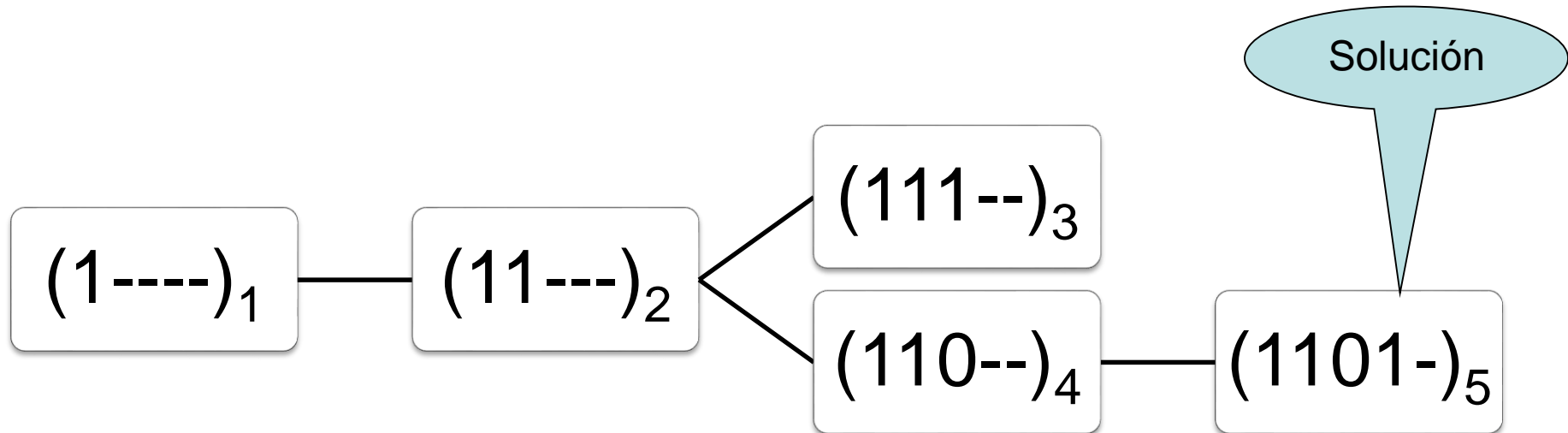
Fin

Subconjuntos

Ejemplo: conjunto $V = (2,3,5,10,20)$ y $M = 15$

Cuyas soluciones son: $X=(1,1,0,1,0)$ y $X=(0,0,1,1,0)$.

El árbol que va construyendo el algoritmo es:



Ejemplo:

$V = (2,3,5,10,20)$ y $M = 15$

