Backtrack

- *Backtracking* (o búsqueda atrás) es una técnica de programación para hacer búsqueda sistemática a través de todas las configuraciones posibles dentro de un espacio de búsqueda.
- Para lograr esto, los algoritmos de tipo backtracking construyen posibles soluciones candidatas de manera sistemática. En general, dado una solución candidata s:
- 1. Verifican si s es solución. Si lo es, hacen algo con ella (depende del problema).
- 2. Construyen todas las posibles extensiones de s, e invocan recursivamente al algoritmo con todas ellas.
- A veces los algoritmos de tipo backtracking se usan para encontrar una solución, pero otras veces interesa que las revisen todas (por ejemplo, para encontrar la mas corta).

Backtrack

- **Suposiciones sobre el espacio de soluciones**
- • Supondremos que una solución se puede modelar como un vector $a = (a_1, a_2, \ldots, a_n)$, donde cada elemento a_i está tomado de un conjunto ordenado finito Si.
- • Representamos a una solución candidata como un vector $a = (a_1, \ldots, a_k)$.
- • Las soluciones candidatas se extenderán agregando un elemento al final.

Backtrack

Algoritmo Genérico de *Backtracking*

El siguiente es un algoritmo genérico de backtracking:

```
Bt(A, k)
1 si Solucion?(A, k)
2  entonces procesarSolucion(A, k)
3  otro para cada c 2 Sucesores(A, k)
4         do A[k] = c
5         Bt(A, k + 1)
6                  si terminar?
7                  entonces devuelve
```

Backtrack

donde

- Solucion?(·) es una función que retorna verdadero si su argumento es una solución.
- procesarSolucion(·), depende del problema y que maneja una solución.
- Sucesores(·) es una función que dado un candidato, genera todos los candidatos que son extensiones de éste.
- terminar? es una variable global booleana inicialmente es falsa, pero que puede

ser hecha verdadera por procesarSolucion, en caso que sólo interesa encontrar una solución

Backtracking

Un ejemplo práctico

Supongamos que queremos un algoritmo para encontrar todos los subconjuntos de *n* elementos de un conjunto de *m* elementos.

Supongamos, además, que los conjuntos están implementados en un arreglo y que la variable *s*[] contiene al conjunto.

¿Qué es un candidato?

Como un candidato es una solución parcial, y representa a un conjunto que tiene a lo más n elementos.

El candidato se representa por un vector binario ($a_1, \ldots, a_k$) donde $a_i = 1$ si el i-ésimo elemento de s está en el subconjunto representado.

¿Cuáles son las formas de extender un candidato ($a_1, \ldots, a_k$)?

Agregando un 0 o un 1 al final de ´este.

¿Cuándo un candidato es una solución? Si $\sum_{i=0}^{k} a_k = n$ y $k = m$

Backtracking

Concepto

En su forma básica, la idea de backtracking se asemeja a un recorrido en profundidad dentro de un grafo dirigido. El grafo en cuestión suele ser un árbol, o por lo menos no contiene ciclos.

Sea cual sea su estructura, existe sólo implícitamente. El objetivo del recorrido es encontrar soluciones para algún problema. Esto se consigue construyendo soluciones parciales a medida que progresa el recorrido; estas soluciones parciales limitan las regiones en las que se puede encontrar una solución completa.

El recorrido tiene éxito si, procediendo de esta forma, se puede definir por completo una solución. En este caso el algoritmo puede bien detenerse o bien seguir buscando soluciones alternativas.

Por otra parte, el recorrido no tiene éxito si en alguna etapa la solución parcial construida hasta el momento no se puede completar. En tal caso, el recorrido vuelve atrás exactamente igual que en un recorrido en profundidad, eliminando sobre la marcha los elementos que se hubieran añadido en cada fase. Cuando vuelve a un nodo que tiene uno o más vecinos sin explorar, prosigue el recorrido de una solución.

Backtracking

Algoritmo de Backtracking

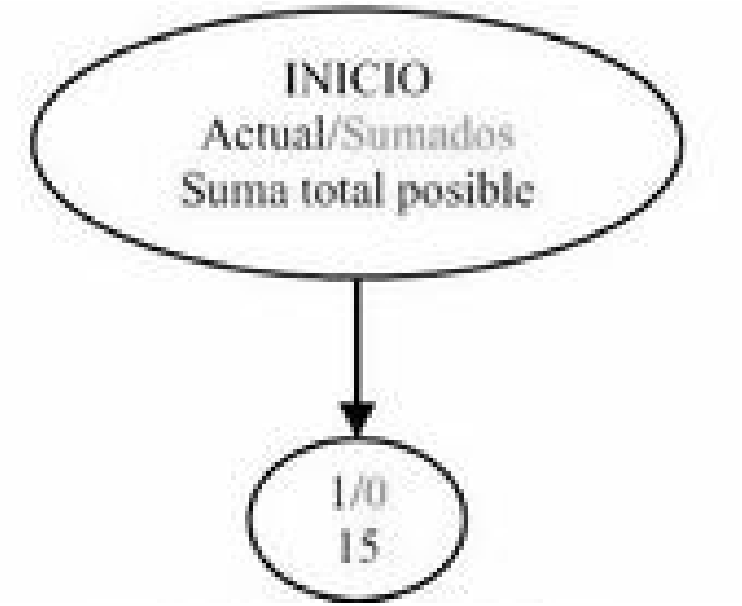
```
proc Backtracking (? X[1 . . . i ]: TSolución, ?  ok: B)
     variables L: ListaComponentes
    inicio
 si EsSolución (X)  entonces ok    CIERTO
    en otro caso
          ok    FALSO
           L=Candidatos (X)
           mientras ¬ok ^ ¬Vacía (L) hacer
          X[i + 1]    Cabeza (L); L    Resto (L)
          Backtracking (X, ok)
          finmientras
          finsi
 fin
```

Backtracking

- Podemos visualizar el funcionamiento de una técnica de backtracking como la exploración en profundidad de un grafo.
- Cada vértice del grafo es un posible estado de la solución del problema. Cada arco del grafo representa la transición entre dos estados de la solución (i.e., la toma de una decisión).
- Típicamente el tamaño de este grafo será inmenso, por lo que no existirá de manera explícita. En cada momento sólo tenemos en una estructura los nodos que van desde el estado inicial al estado actual. Si cada secuencia de decisiones distinta da lugar a un estado diferente, el grafo es un árbol (el árbol de estados).

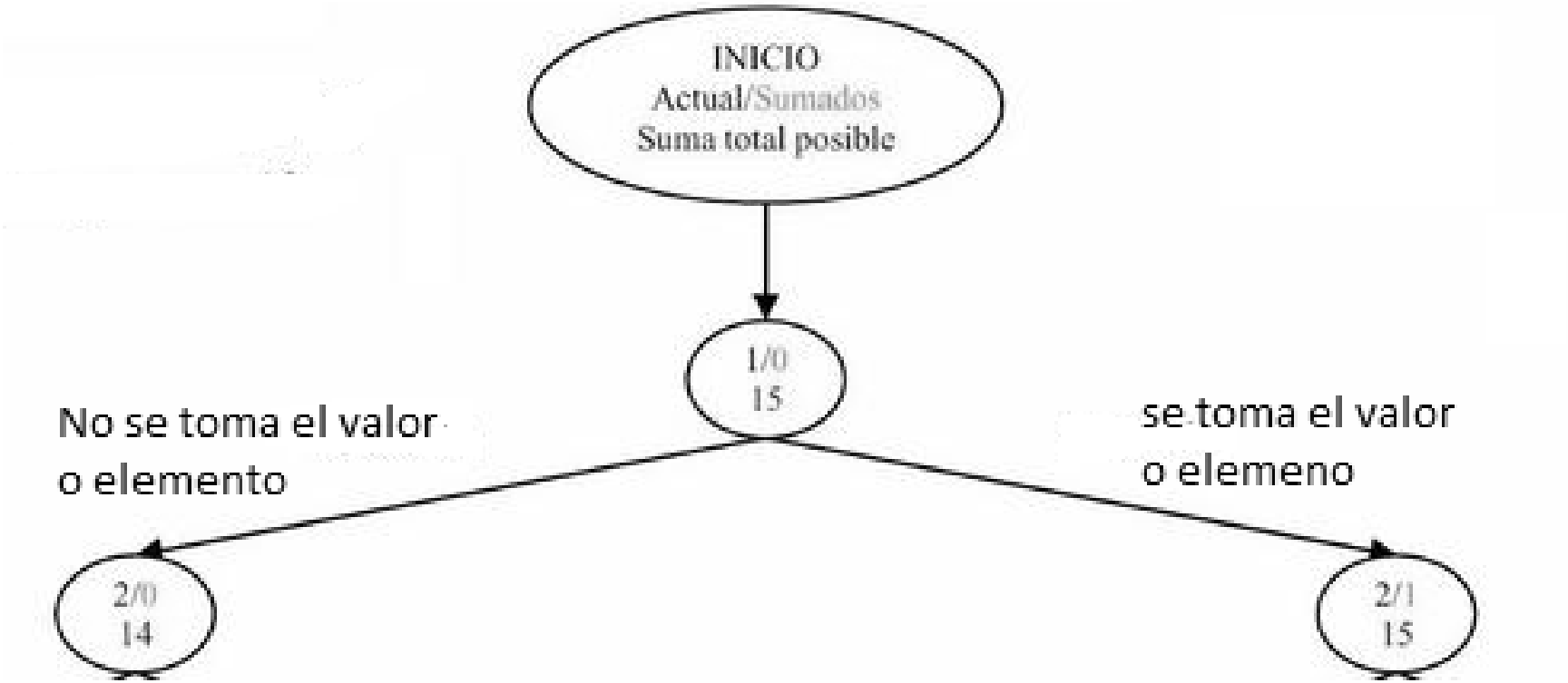
Backtracking

- Sea el conjunto de numeros: {1, 2, 3, 4, 5}
- Se desea encontrar el subconjunto cuya suma sea par



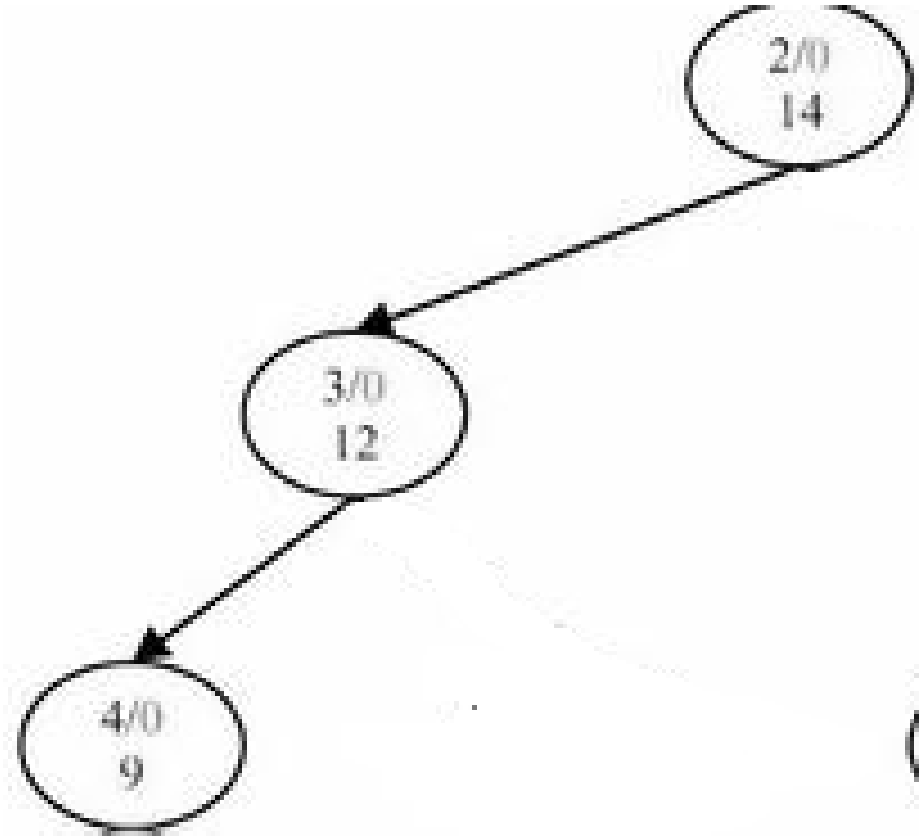
Backtracking

- Se tienen dos posibilidades:



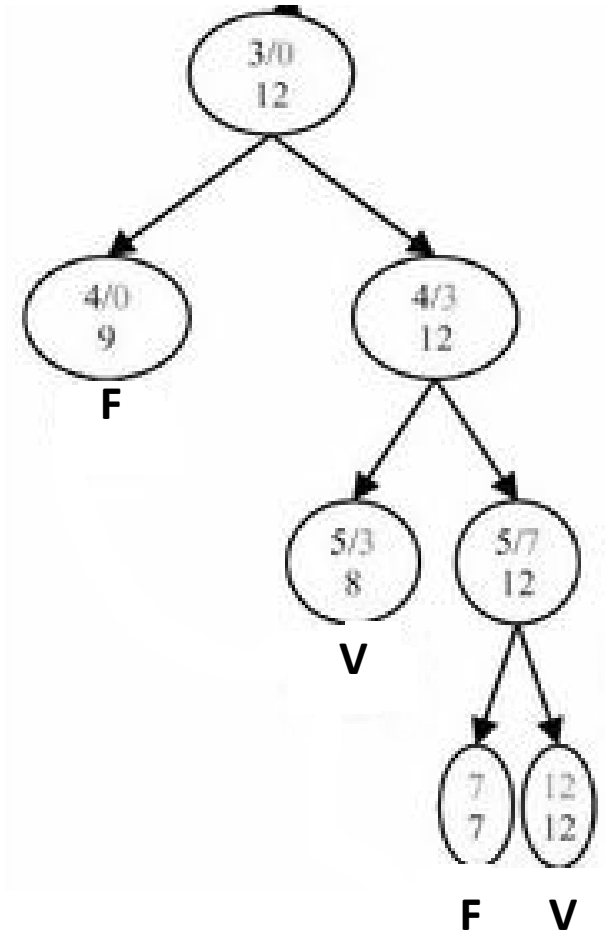
Backtrack

Veamos siempre por el subárbol izquierdo:



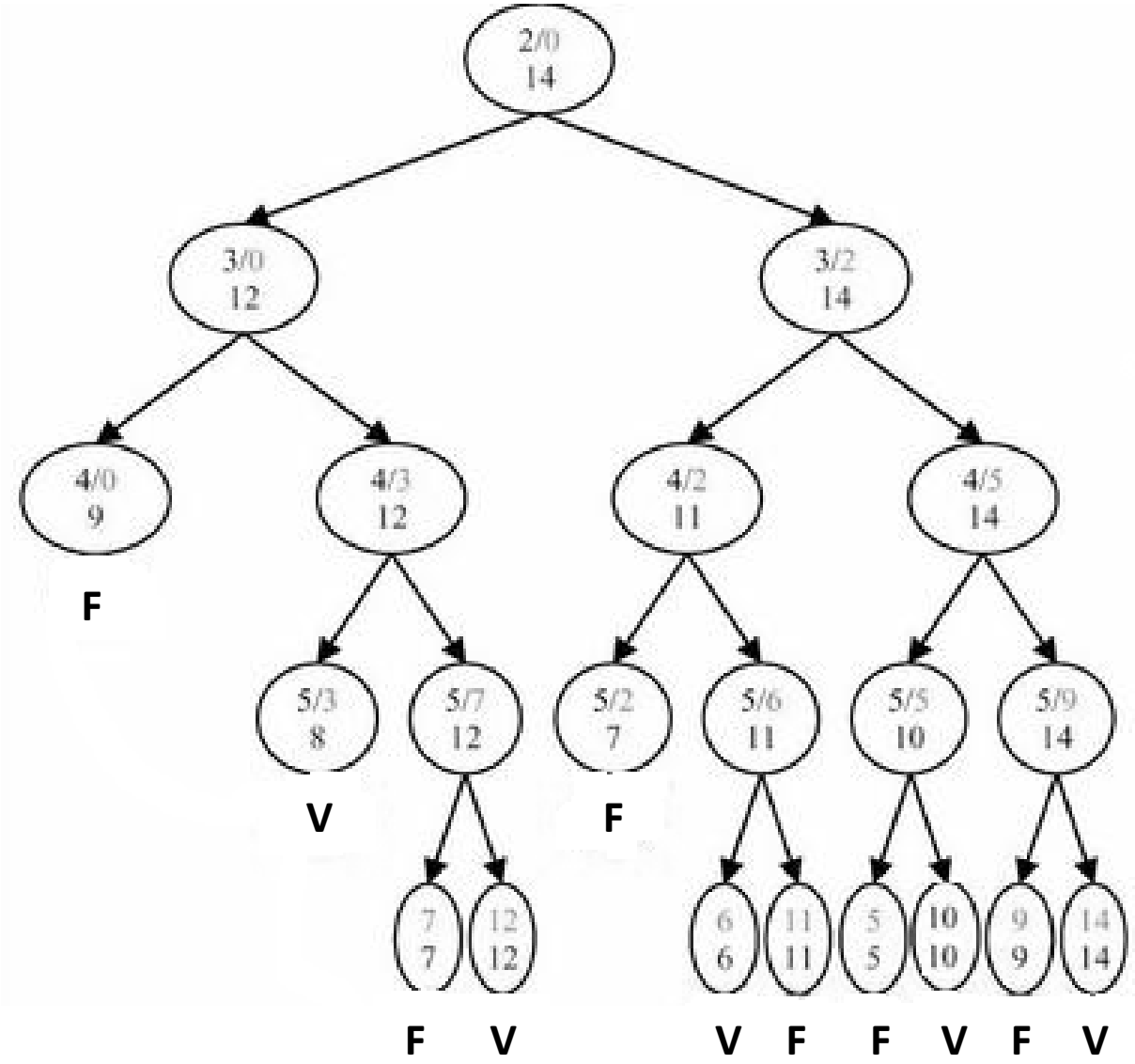
Backtrack

- Considerando los posibles caminos a partir del tercer nivel (decision negativa del 2º):



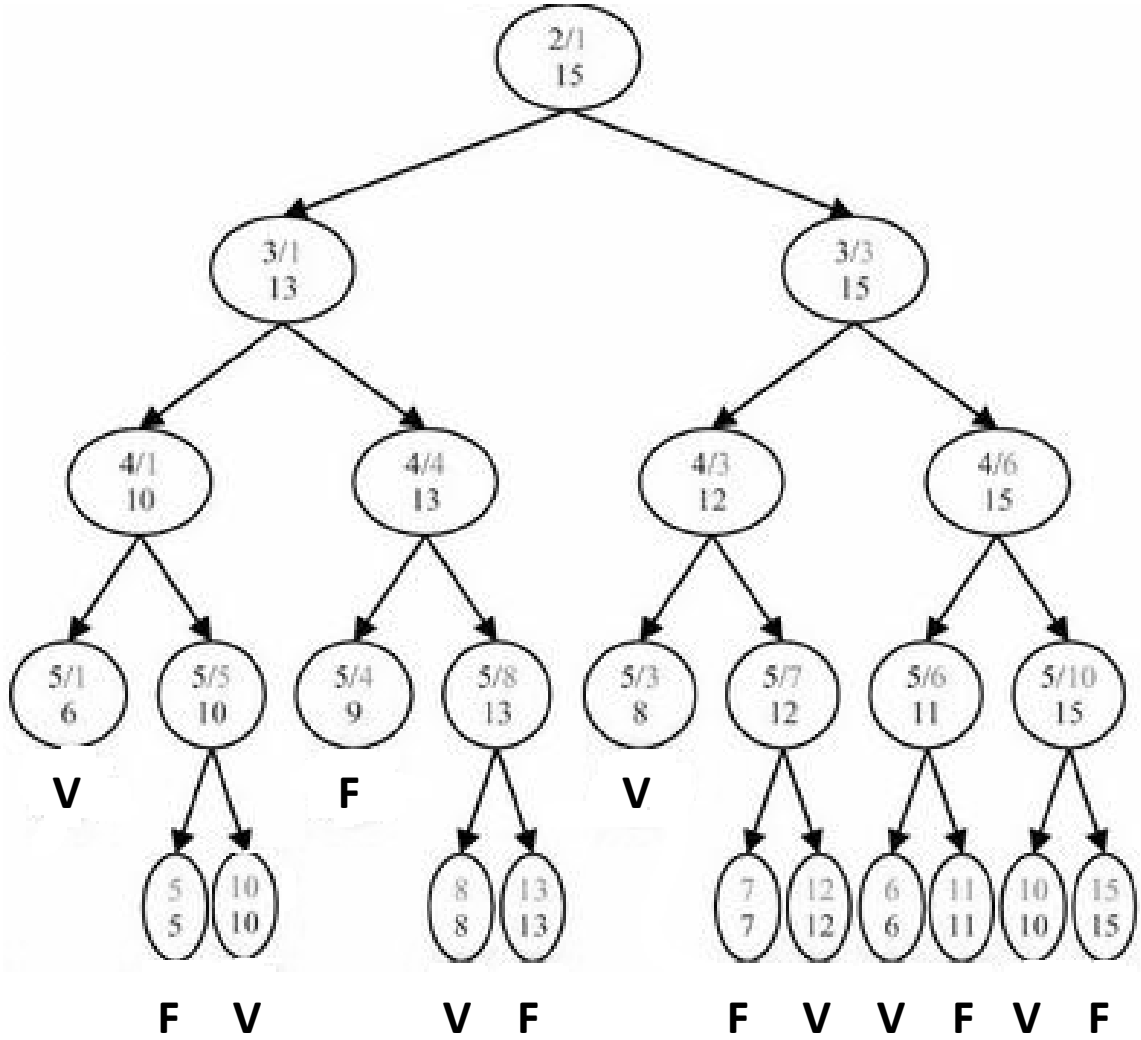
Backtrack

- Ahora todas las decisiones a partir del segundo nivel (izq.):



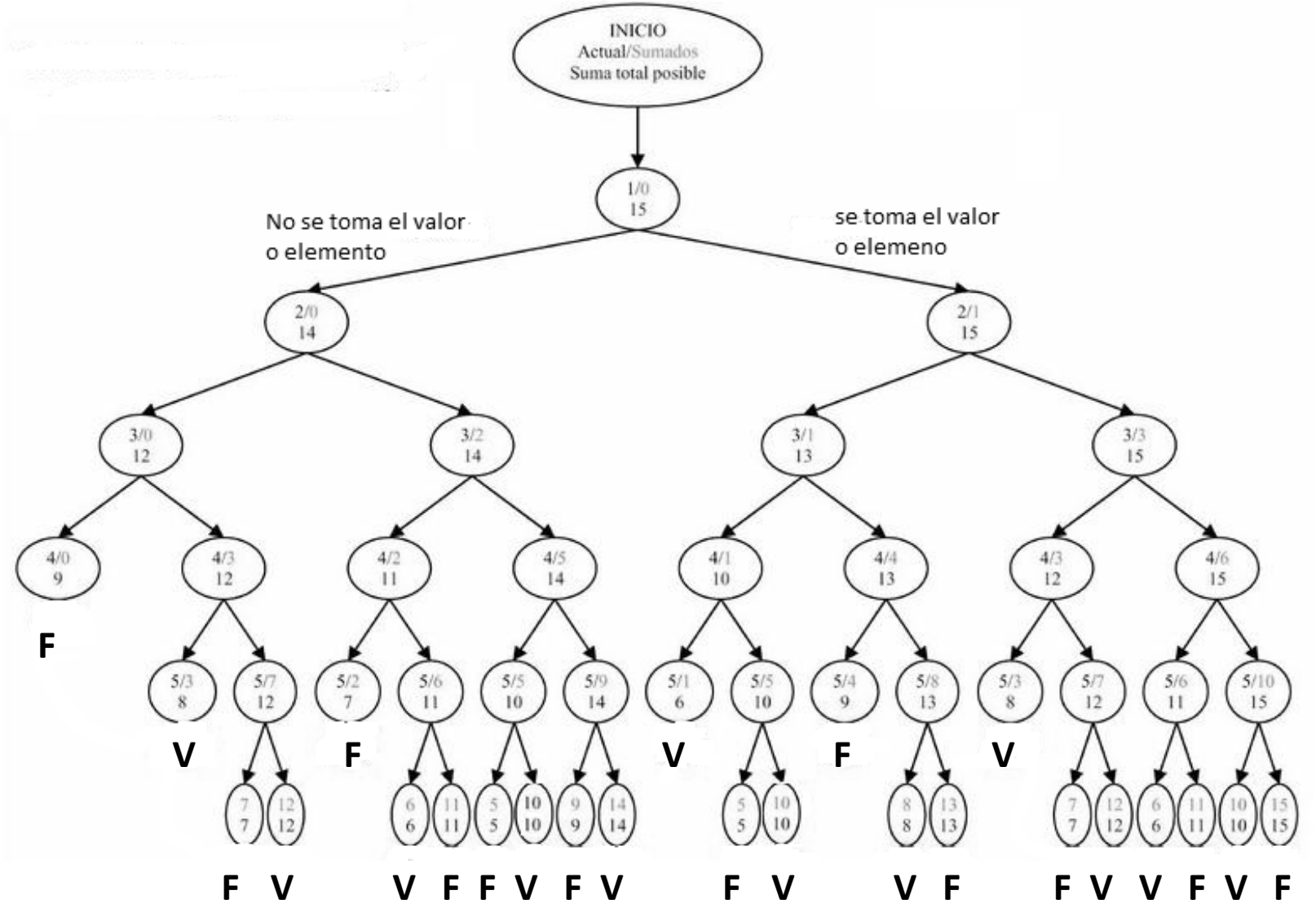
Backtrack

- Las posibilidades a partir del segundo nivel (der.):



Backtrack

- El árbol completo es el siguiente:



Backtrack

- Backtrack está muy relacionado con la búsqueda combinatoria.
- Generalmente el orden de los elementos de la solución no importa.
- Estos problemas consisten en un conjunto de variables a la que a cada una se le debe asignar un valor sujeto a las restricciones del problema.
- La técnica va creando todas las posibles combinaciones de elementos para obtener una solución.
- Su principal virtud es que en la mayoría de las implementaciones se puede evitar combinaciones, estableciendo funciones de acotación (o poda).

Backtrack

Implementación

- La idea es encontrar la mejor combinación (o alguna) posible en un momento determinado, por ello este tipo de algoritmo es una búsqueda en profundidad. Durante la búsqueda, si se encuentra una alternativa incorrecta, la búsqueda retrocede hasta el paso anterior y toma la siguiente alternativa. Cuando se han terminado las posibilidades, se vuelve a la elección anterior y se toma la siguiente opción.
- Si no hay más alternativas la búsqueda falla. De esta manera, se crea un árbol implícito, en el que cada nodo es un esta do de la solución (solución parcial o total).
- Normalmente, se suele implementar este tipo de algoritmos de forma recursiva. Así, en cada llamada al procedimiento se toma una variable y se le asignan todos los valores posibles, llamando a su vez al procedimiento para cada uno de los nuevos estados.

Backtrack

Ejemplos de aplicación de backtracking

El problema del vendedor viajero (Ciclo Hamiltoniano)

- El problema del vendedor viajero consiste en que tenemos un grafo no dirigido en el cual los arcos tienen costos asociados, y queremos encontrar un camino cerrado que recorra a todos los nodos del grafo, con costo mínimo.
- Para resolver este problema suponemos que el grafo está representado por una matriz de costos.
- En este caso los subproblemas S son caminos que parten de a y llegan a b a través de un sucesión de nodos T. (b es el mismo a lo largo de todo el algoritmo).
- Inicialmente A contiene solamente el camino (a, vacío, b).

Backtrack

- Elegimos un subproblema S cualquiera de A (y lo borramos de A) y añadimos ramas (c, a) del grafo (las c's son las adyacentes de a) . Estos caminos extendidos son los hijos. Ahora cada c juega el rol de a.
- Examinamos c/ hijo:
- Test:
- 1)Si G-T forma un camino hamiltoniano STOP (solución hallada)
- 2)Si G-T tiene un nodo de grado uno (excepto a y b) o si G-T-{a, b} es disconexo entonces DROP este subproblema.
- 3) Si 1) y 2) fallan agregar subproblema en A.

Backtrack

El Problema de la mochila

Nuevamente se trata de encontrar la mejor solución, la solución óptima, de entre todas las soluciones.

Partiendo del esquema que genera todas las soluciones expuesto anteriormente se puede obtener la mejor solución (la solución óptima, seleccionada entre todas las soluciones) si se modifica la instrucción almacenar_solucion por esta otra:
si f(solucion) > f(optimo) **entonces** optimo <- solucion
siendo *f(s)* función positiva, *optimo* es la mejor solucion encontrada hasta el momento, y *solucion* es una solución que se está probando.

Backtrack

- El problema de la mochila consiste en llenar una mochila con una serie de objetos que tienen una serie de pesos con un valor asociado. Es decir, se dispone de *n tipos* de objetos y que no hay un número limitado de cada tipo de objeto.
- Cada tipo *i* de objeto tiene un peso w_i positivo y un valor v_i positivo asociados. La mochila tiene una capacidad de peso igual a *W*. Se trata de llenar la mochila de tal manera que se **maximice** el **valor** de los objetos incluidos pero respetando al mismo tiempo la restricción de capacidad.
- Notar que no es obligatorio que una solución óptima llegue al límite de capacidad de la mochila.

Backtrack

- **Ejemplo:** se supondrá:
  n = 4
  W = 8
  w() = 2, 3, 4, 5
  v() = 3, 5, 6, 10
  Es decir, hay 4 tipos de objetos y la mochila tiene una capacidad de 8. Los pesos varían entre 2 y 5, y los valores relacionados varían entre 3 y 10.
  Una solución no óptima de valor 12 se obtiene introduciendo cuatro objetos de peso 2, o 2 de peso 4.
  Otra solución no óptima de valor 13 se obtiene introduciendo 2 objetos de peso 3 y 1 objeto de peso 2.
  ¿Cuál es la solución óptima?.
- A continuación se muestra una solución al problema, variante del esquema para obtener todas las soluciones.

Backtrack

```
void mochila(int i, int r, int solucion, int *optimo)
{
  int k;

  for (k = i; k < n; k++) {
    if (peso[k] <= r) {
      mochila(k, r - peso[k], solucion + valor[k], optimo);
      if (solucion + valor[k] > *optimo) *optimo = solucion+valor[k];
    }
  }
}
```

Dicho procedimiento puede ser ejecutado de esta manera, siendo n, W, peso y valor variables globales para simplificar el programa:

```
n = 4,      W = 8,    peso[] = {2,3,4,5}, valor[] = {3,5,6,10},          optimo = 0;
    ...
    mochila(0, W, 0, &optimo);
```

Observar que la solución óptima se obtiene independientemente de la forma en que se ordenen los objetos.