




Backtracking

Algoritmos y estructuras de datos II

José Antonio Hernández López¹

¹Departamento de Informática y Sistemas
Universidad de Murcia

9 de abril de 2025

- 1 Introducción a backtracking
- 2 Suma del subconjunto
- 3 Metodología, tiempos y comparación con PD
- 4 Mochila 0/1 
- 5 Asignación de trabajos 
- 6 Conclusiones 

Definición

Backtracking es un método que realiza una búsqueda exhaustiva y sistemática del espacio de soluciones del problema.

- Problemas de decisión (encontrar una solución)
- Problemas de enumeración (encontrar todas las soluciones)
- Problemas de optimización (encontrar la mejor solución)

Son muy ineficientes con órdenes exponenciales o factoriales.



- El algoritmo de backtracking va a buscar un solución $s = (x_1, \dots, x_n)$ que tiene que satisfacer unas **restricciones** y, quizá, optimizar una **función objetivo**.

- El algoritmo de backtracking va a buscar un solución $s = (x_1, \dots, x_n)$ que tiene que satisfacer unas **restricciones** y, quizá, optimizar una **función objetivo**.
- Cada valor x_i representa una decisión tomada.

- El algoritmo de backtracking va a buscar una solución $s = (x_1, \dots, x_n)$ que tiene que satisfacer unas **restricciones** y, quizá, optimizar una **función objetivo**.
- Cada valor x_i representa una decisión tomada.
- En cada paso, el algoritmo se encuentra en un cierto nivel k con una solución parcial (x_1, \dots, x_k) .

- El algoritmo de backtracking va a buscar un solución $s = (x_1, \dots, x_n)$ que tiene que satisfacer unas **restricciones** y, quizá, optimizar una **función objetivo**.
- Cada valor x_i representa una decisión tomada.
- En cada paso, el algoritmo se encuentra en un cierto nivel k con una solución parcial (x_1, \dots, x_k) .
 - 1 Si la solución parcial es solución final, terminamos.
 - 2 Si se puede añadir un elemento a la solución x_{k+1} se añade y se pasa al nivel $k + 1$.
 - 3 si no, se prueban otros valores para x_k .
 - 4 si no queda ningún valor más, se retrocede a $k - 1$.

Ejemplo: el problema de la rana

Descripción del problema

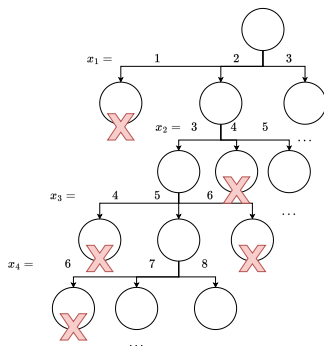
- La rana empieza en la posición 0 y quiere llegar a la posición n .
- Hay nenúfares en varias posiciones. Hay uno en la posición 0 y otra en la posición n .
- La rana puede saltar, como máximo, d unidades en un solo salto.
- **Objetivo:** Encontrar el camino que la rana debería seguir que minimize el número de saltos. Se asume que existe una solución.

Ejemplo: el problema de la rana 🐸



Forma de la solución: es una tupla o secuencia de posiciones donde la rana debe saltar.

Ejemplo: el problema de la rana 🐸



- Se recorren todas las soluciones.
- El conjunto de todas las soluciones se pueden representar como un árbol implícito.
- El árbol se recorre siguiendo un orden primero en profundidad.

Backtracking

El primer paso para aplicar backtracking a un problema es seleccionar la forma de la tupla solución (x_1, \dots, x_n) y la forma del árbol:

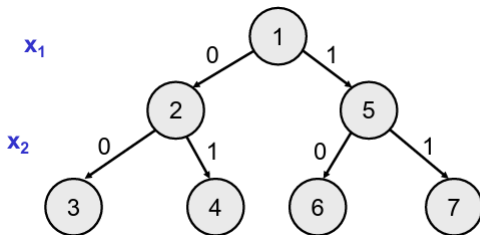
- Árboles binarios. (x_1, \dots, x_n) con $x_i = 0, 1$.
- Árboles k -arios. (x_1, \dots, x_n) con $x_i = 1, \dots, k$.
- Árboles permutacionales. (x_1, \dots, x_n) con $x_i = 1, \dots, n$ y $x_i \neq x_j$ para todo $i \neq j$.
- Árboles combinatorios. (x_1, \dots, x_m) con $m \leq n$, $x_i \in \{1, \dots, n\}$ y $x_i < x_{i+1}$.

Nota importante

El árbol es implícito. Backtracking no mantiene un árbol en memoria. Conceptualmente, el conjunto de todas las soluciones se puede ver como un árbol. El algoritmo de backtracking recorre ese árbol primero en profundidad.

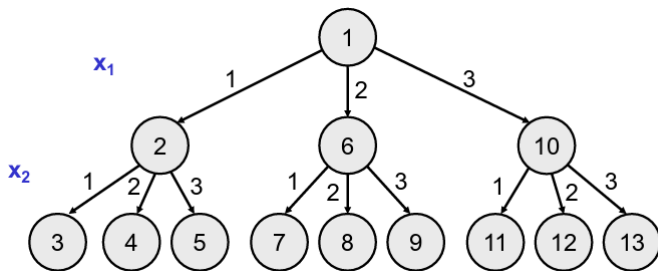
Backtracking

Árboles binarios. (x_1, \dots, x_n) con $x_i = 0, 1$.



Ejemplos de problemas. Cuando hay que escoger elementos: mochila 0/1, suma del subconjunto, etc.

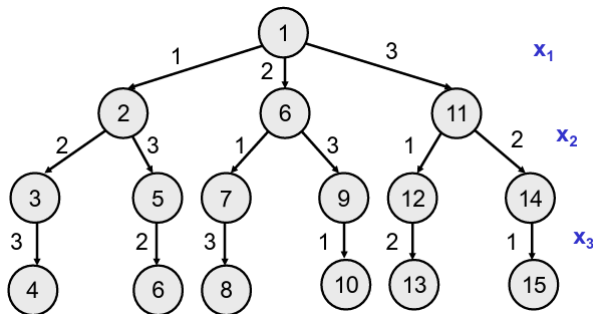
Árboles k -arios. (x_1, \dots, x_n) con $x_i \in \{1, \dots, k\}$.



Ejemplos de problemas. Tenemos como máximo k opciones para cada x_i : cambio de monedas.

Backtracking

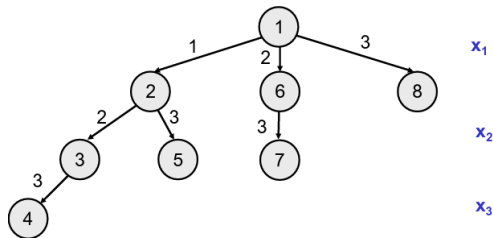
Árboles permutacionales. (x_1, \dots, x_n) con $x_i = 1, \dots, n$ y $x_i \neq x_j$ para todo $i \neq j$.



Ejemplos de problemas. Los x_i no se pueden repetir. Asignar n trabajos a n personas.

Backtracking

Árboles combinatorios. (x_1, \dots, x_m) con $m \leq n$, $x_i \in \{1, \dots, n\}$ y $x_i < x_{i+1}$.



Ejemplos de problemas. Los mismos que con árboles binarios. Ya que toda solución binaria se puede expresar como una solución combinatoria (y viceversa). Cada nodo puede ser una solución.

$$(0, 1, 0, 1, 0, 0, 1) \rightleftharpoons (2, 4, 7)$$

El árbol de la rana 🐸 pertenece a este grupo.

Esquema general (problema de optimización):

```
1 def backtracking_optimizacion_minimizar():
2     nivel = 1
3     s = s_0
4     voa = float("inf") # valor óptimo actual
5     soa = None # solución óptima actual
6
7     while nivel != 0:
8         # genero un hermano
9         generar(nivel, s)
10
11         # si he encontrado una solución que mejora a voa
12         # lo guardo
13         if solucion(nivel, s) and valor(s) < voa:
14             voa = valor(s)
15             soa = s
16
17         # si podríamos llegar a la solución final, seguimos
18         if criterio(nivel, s):
19             nivel += 1
20         else:
21             # en caso contrario, retrocedo hasta que encuentre un nodo con más hermanos
22             # por explorar
23             while nivel > 0 and (not masHermanos(nivel, s)):
24                 retroceder(nivel, s)
```


- s almacena la solución parcial.
- s_0 almacena el valor de iniciación.
- nivel: indica el nivel en el que está el algoritmo.
- generar(nivel, s): genera el siguiente hermano para el nivel actual.
- solucion(nivel, s): comprueba si la tupla actual es solución.
- criterio(nivel, s): comprueba si a partir de la solución actual es posible llegar a la solución final.
- masHermanos(nivel, s): devuelve True si hay más hermanos del nodo actual.
- retroceder(nivel, s): retrocede un nivel del árbol actualizando s si fuese necesario.

Ejemplo: el problema de la rana

```
1 # nenufares es una lista de 0s y 1s, d es un entero que representa el alcance
2 def rana_backtracking(nenufares, d):
3     # la solución es una lista inicializada a [0]
4     s = [0]
5     nivel = 1
6     voa = float("inf")
7     soa = None
8     n = len(nenufares)
9
10    while (nivel!=0):
11        # genero el siguiente hermano (la siguiente posición alcanzable)
12        generar(nivel, s, d, nenufares)
13
14        # solucion y valor
15        # si he llegado al final (solución) y valor(s) < voa
16        if s[-1] == n - 1 and len(s) < voa:
17            voa = len(s)
18            soa = s.copy()
19
20        # criterio: si todavía puedo mejorar el camino y no he llegado al final
21        if nenufares[s[-1]] == 1 and len(s) < voa and s[-1] < n - 1:
22            nivel += 1
23        else:
24            # en caso contrario retrocedo hasta que encuentre un nodo con más hermanos
25            # retroceder aquí es quitar un nivel y borrar el último elemento de la lista
26            while (nivel > 0) and (not masHermanos(nivel, s, d)):
27                nivel -= 1
28                s.pop()
29    return soa
```

Ejemplo: el problema de la rana

```
1 def generar(nivel, s, d, nenufares):
2     if len(s) == nivel:
3         # generar el primer hermano
4         x = s[-1]
5         for j in range(1, d+1):
6             if x + j < len(nenufares):
7                 candidato = x + j
8                 break
9         s.append(candidato)
10    else:
11        # coger el siguiente hermano
12        x = s[-2]
13        y = s[-1]
14        for j in range(1, d+1):
15            if x + j < len(nenufares) and x + j > y:
16                candidato = x + j
17                break
18        s[-1] = candidato
19
20 # hay más hermanos si la posición i no es la última posición y hay más posiciones alejadas
   que no superen d
21 def masHermanos(nivel, s, d):
22     x = s[-2]
23     y = s[-1]
24     return y != len(nenufares) - 1 and y - x < d
```

Suma del subconjunto

Descripción del problema

Dado un conjunto de números A , encontrar un subconjunto (si existe) que sume M . Por ejemplo, para

$$A = \{13, 11, 7\}, M = 20$$

El subconjunto $\{13, 7\}$ suma 20.

$$A = \{13, 11, 7\}, M = 8$$

No existe ningún subconjunto.

Suma del subconjunto

Supongamos los elementos de $A = \{a_1, \dots, a_n\}$ numerados de 1 a n y sea M el número a sumar.

Suma del subconjunto

Supongamos los elementos de $A = \{a_1, \dots, a_n\}$ numerados de 1 a n y sea M el número a sumar.

- Primer paso para aplicar backtracking: tenemos que definir la tupla solución s que contenga las decisiones (y, como consecuencia, tendremos el tipo de árbol).

Suma del subconjunto

Supongamos los elementos de $A = \{a_1, \dots, a_n\}$ numerados de 1 a n y sea M el número a sumar.

- Primer paso para aplicar backtracking: tenemos que definir la tupla solución s que contenga las decisiones (y, como consecuencia, tendremos el tipo de árbol).
- $s = (x_1, \dots, x_n)$ tal que $x_i = 1$ si el elemento a_i se coge y 0 en caso contrario. Esto formará un árbol binario.

Suma del subconjunto

Supongamos los elementos de $A = \{a_1, \dots, a_n\}$ numerados de 1 a n y sea M el número a sumar.

- Primer paso para aplicar backtracking: tenemos que definir la tupla solución s que contenga las decisiones (y, como consecuencia, tendremos el tipo de árbol).
- $s = (x_1, \dots, x_n)$ tal que $x_i = 1$ si el elemento a_i se coge y 0 en caso contrario. Esto formará un árbol binario.
- Inicializamos todo a -1: $s_0 = (-1, \dots, -1)$. En este contexto -1 quiere decir que el elemento a_i no ha sido investigado.

Suma del subconjunto

Estamos ante un problema de decisión (donde nos basta con una solución):

```
1 def backtracking_decision():
2     nivel = 1
3     s = s_0
4
5     while nivel != 0:
6         # genero un hermano
7         generar(nivel, s)
8
9         # si he encontrado la solución termino
10        if solucion(nivel, s):
11            return s
12
13        # si podemos llegar a la solución final en niveles inferiores
14        if criterio(nivel, s):
15            nivel += 1
16        else:
17            # en caso contrario, retrocedo hasta que encuentre un nodo con más hermanos
18            # por explorar
19            while nivel > 0 and (not masHermanos(nivel, s)):
20                retroceder(nivel, s)
21
22    print("No hay solución")
```

Suma del subconjunto

```
1 def subconjunto_backtracking(A, M):
2     nivel = 1
3     s = [-1]*len(A)
4
5     while nivel != 0:
6         # genero un hermano
7         generar(nivel, s)
8
9         # si he encontrado la solución termino
10        if solucion(nivel, s, A, M):
11            return s
12
13        # si renta seguir explorando
14        if criterio(nivel, s, A, M):
15            nivel += 1
16        else:
17            # en caso contrario, retrocedo hasta que encuentre un nodo con más hermanos
18            # por explorar
19            while nivel > 0 and (not masHermanos(nivel, s)):
20                s[nivel-1] = -1
21                nivel -= 1
22
23    print("No hay solución")
```

Suma del subconjunto

```
1 def generar(nivel, s):
2     s[nivel-1] += 1
3
4 def solucion(nivel, s, A, M):
5     suma = 0
6     for i in range(0, nivel):
7         suma += A[i]*s[i]
8     return suma == M and len(A) == nivel
9
10 def criterio(nivel, s, A, M):
11     suma = 0
12     for i in range(0, nivel):
13         suma += A[i]*s[i]
14     return suma <= M and nivel < len(A)
15
16 def masHermanos(nivel, s):
17     return s[nivel-1] == 0
```

Suma del subconjunto

Este algoritmo podemos optimizarlo añadiendo una variable auxiliar que va almacenando la suma actual. Así nos ahorramos hacer dos bucles de tamaño `nivel` en cada nodo.

```
1 def subconjunto_backtracking_optimizado(A, M):
2     nivel = 1
3     s = [-1]*len(A)
4     suma = 0
5     while nivel != 0:
6         # genero un hermano
7         generar(nivel, s)
8         if s[nivel-1] == 1:
9             suma += A[nivel - 1]
10
11         # si he encontrado la solución termino
12         if solucion_optimizada(nivel, A, M, suma):
13             return s
14
15         # si renta seguir explorando
16         if criterio_optimizado(nivel, A, M, suma):
17             nivel += 1
18         else:
19             # en caso contrario, retrocedo hasta que encuentre un nodo con más hermanos
20             # por explorar
21             while nivel > 0 and (not masHermanos(nivel, s)):
22                 suma -= s[nivel-1]*A[nivel - 1]
23                 s[nivel-1] = -1
24                 nivel -= 1
25     print("No hay solución")
```

Suma del subconjunto

```
1 def solucion_optimizada(nivel, A, M, suma):  
2     return suma == M and len(A) == nivel  
3  
4 def criterio_optimizado(nivel, A, M, suma):  
5     return suma <= M and nivel < len(A)
```

Suma del subconjunto

Si quisiéramos devolver todos los subconjuntos que sumen M , estaríamos ante un problema de enumeración (tenemos que devolver todas las soluciones):

```
1 def backtracking_enumeracion():
2     nivel = 1
3     s = s_0
4     S = []
5
6     while nivel != 0:
7         # genero un hermano
8         generar(nivel, s)
9
10        # si he encontrado la añadido a mi lista de soluciones
11        if solucion(nivel, s):
12            S.append(s)
13
14        # si renta seguir explorando
15        if criterio(nivel, s):
16            nivel += 1
17        else:
18            # en caso contrario, retrocedo hasta que encuentre un nodo con más hermanos
19            # por explorar
20            while nivel > 0 and (not masHermanos(nivel, s)):
21                retroceder(nivel, s)
22
23    return S
```

Suma del subconjunto

Si quisiéramos devolver todos los subconjuntos que sumen M , estaríamos ante un problema de enumeración (tenemos que devolver todas las soluciones):

```
1 def subconjunto_backtracking_enumeracion(A, M):
2     nivel = 1
3     s = [-1]*len(A)
4     suma = 0
5     S = []
6     while nivel != 0:
7         # genero un hermano
8         generar(nivel, s)
9         if s[nivel-1] == 1:
10             suma += A[nivel - 1]
11
12         # si he encontrado la solución termino
13         if solucion_optimizada(nivel, A, M, suma):
14             S.append(s.copy())
15
16         # si renta seguir explorando
17         if criterio_optimizado(nivel, A, M, suma):
18             nivel += 1
19         else:
20             # en caso contrario, retrocedo hasta que encuentre un nodo con más hermanos
21             # por explorar
22             while nivel > 0 and (not masHermanos(nivel, s)):
23                 suma -= s[nivel-1]*A[nivel - 1]
24                 s[nivel-1] = -1
25                 nivel -= 1
26     return S
```

Pasos para aplicar backtracking

Aplicar backtracking es bastante sistemático:

- 1 Representación de la tupla $s = (x_1, \dots, x_n)$ y forma del árbol.
- 2 En función del tipo de problema (decisión, optimización o enumeración), escoger el esquema adecuado.
- 3 Implementar funciones genéricas del esquema.
- 4 Optimizar el algoritmo (buena función criterio - podar y agilizar trabajo por nodo).

Tiempos de ejecución

El tiempo suele calcularse como

$$\# \text{ número de nodos} \times \text{tiempo/nodo}$$

Tiempos de ejecución

El tiempo suele calcularse como

$$\# \text{ número de nodos} \times \text{tiempo/nodo}$$

Dada una entrada (x_1, \dots, x_n) de n niveles, los árboles completos tienen el siguiente número de nodos:

- En árboles binarios, $\# \text{ número de nodos} = \Theta(2^n)$.
- En árboles k -arios, $\# \text{ número de nodos} = \Theta(k^n)$.
- En árboles permutacionales, $\# \text{ número de nodos} = \Theta(n!)$
- En árboles combinatorios, $\# \text{ número de nodos} = \Theta(2^n)$

Tiempos de ejecución

El tiempo suele calcularse como

$$\# \text{ número de nodos} \times \text{tiempo/nodo}$$

Dada una entrada (x_1, \dots, x_n) de n niveles, los árboles completos tienen el siguiente número de nodos:

- En árboles binarios, $\# \text{ número de nodos} = \Theta(2^n)$.
- En árboles k -arios, $\# \text{ número de nodos} = \Theta(k^n)$.
- En árboles permutacionales, $\# \text{ número de nodos} = \Theta(n!)$
- En árboles combinatorios, $\# \text{ número de nodos} = \Theta(2^n)$

Hay dos formas de optimizar estos algoritmos:

- Que el tiempo por nodo sea lo mínimo posible.
- Que no se recorra todo el árbol (podar). De manera que solemos pasar de $\Theta(\text{exponencial o factorial})$ a $O(\text{exponencial o factorial})$.

número de nodos \times tiempo/nodo

Por ejemplo, en el problema del subconjunto, hemos hecho dos optimizaciones:

- Llevar una variable auxiliar con la suma actual para evitar recorrer, en cada nodo, dos arrays de tamaño su nivel. Esto hace que el algoritmo baje de $\Theta(n2^n)$ a $\Theta(2^n)$.
- Podar cuando la suma de la variable auxiliar es mayor que M . Así obtenemos $O(2^n)$.

Backtracking vs programación dinámica

Programación dinámica se puede ver como una *super poda del backtracking*. Por ejemplo, veamos el recorrido turístico por Manhattan. La recursión ingénua es:

```
1 def manhattan_recursivo(i, j):
2     if i == j == 0:
3         m = 0
4     else:
5         if i > 0:
6             e_arriba = down[i - 1][j]
7             m1 = manhattan_recursivo(i - 1, j) + e_arriba
8         else:
9             m1 = float("-inf")
10
11        if j > 0:
12            e_izquierda = right[i][j - 1]
13            m2 = manhattan_recursivo(i, j - 1) + e_izquierda
14        else:
15            m2 = float("-inf")
16
17        m = max(m1, m2)
18    return m
```

Esto es una especie de backtracking enmascarado implementado de forma recursiva. Esto tiene orden de $O(4^n)$.

Backtracking vs programación dinámica

Al añadir una caché, esto pasa a un orden de $\Theta(n^2)$ ya que, como se repiten tantos cálculos, *podamos* de manera super agresiva.

```
1 def manhattan_memo(i, j, memo):
2     if (i, j) in memo:
3         return memo[(i,j)]
4     if i == j == 0:
5         m = 0
6     else:
7
8         if i > 0:
9             e_arriba = down[i - 1][j]
10            m1 = manhattan_memo(i - 1, j, memo) + e_arriba
11        else:
12            m1 = float("-inf")
13
14        if j > 0:
15            e_izquierda = right[i][j - 1]
16            m2 = manhattan_memo(i, j - 1, memo) + e_izquierda
17        else:
18            m2 = float("-inf")
19
20        m = max(m1, m2)
21        memo[(i,j)] = m
22        return m
```

Descripción del problema

- Tenemos $O = \{1, \dots, n\}$, n objetos con pesos $p_i > 0$ beneficios $b_i > 0$.
- En la mochila tenemos que meter objetos, con un peso máximo de M . **No** se pueden fraccionar los objetos. Un objeto o se mete o no se mete.
- **Objetivo:** Llenar la mochila maximizando el beneficio sin superar la capacidad máxima.

Representación de la solución

Representación de la solución Vamos a representar la solución con $s = (x_1, \dots, x_n)$ tal que $x_i = 1$ significa que se coge el objeto i y $x_i = 0$ si no se coge. Esto da lugar a un árbol binario.

Tipo de problema

Representación de la solución Vamos a representar la solución con $s = (x_1, \dots, x_n)$ tal que $x_i = 1$ significa que se coge el objeto i y $x_i = 0$ si no se coge. Esto da lugar a un árbol binario.

Tipo de problema Es un problema de optimización (maximización). Así que escogemos el esquema backtracking de optimización.

Implementación de las funciones genéricas

Representación de la solución Vamos a representar la solución con $s = (x_1, \dots, x_n)$ tal que $x_i = 1$ significa que se coge el objeto i y $x_i = 0$ si no se coge. Esto da lugar a un árbol binario.

Tipo de problema Es un problema de optimización (maximización). Así que escogemos el esquema backtracking de optimización.

Implementación de las funciones genéricas

- $s_0 = (-1, \dots, -1)$
- `generar(nivel, s)`: genera el siguiente hermano $\sim ++s[\text{nivel}]$.
- `solución(nivel, s)`: ¿hemos analizado todos los objetos? (es decir, ¿estamos en el último nivel?) y ¿nos hemos pasado M ?
- `criterio(nivel, s)`: ¿nos hemos pasado M ? ¿quedan objetos por analizar?
- `masHermanos(nivel, s)`: ¿hemos analizado ambas posibilidades del objeto?
- `retroceder(nivel, s)`: pasamos al objeto anterior.

Esquema general (problema de optimización):

```
1 def backtracking_optimizacion_maximizar():
2     nivel = 1
3     s = s_0
4     voa = -float("inf")
5     soa = None
6
7     while nivel != 0:
8         # genero un hermano
9         generar(nivel, s)
10
11         # si he encontrado una solución que mejora a voa
12         # lo guardo
13         if solucion(nivel, s) and valor(s) > voa:
14             voa = valor(s)
15             soa = s
16
17         # si podríamos llegar a la solución final, seguimos
18         if criterio(nivel, s):
19             nivel += 1
20         else:
21             # en caso contrario, retrocedo hasta que encuentre un nodo con más hermanos
22             # por explorar
23             while nivel > 0 and (not masHermanos(nivel, s)):
24                 retroceder(nivel, s)
25
26     return soa
```

Inicialización y variables auxiliares

```
1 s=[-1]*len(B)
2 beneficio_actual = 0
3 peso_actual = 0
```

Generar

```
1 s[nivel - 1] += 1
2 peso_actual += P[nivel - 1]*s[nivel - 1]
3 beneficio_actual += B[nivel - 1]*s[nivel - 1]
```

Solución

```
1 peso_actual <= M and len(B) == nivel
```

Criterio

```
1 peso_actual <= M and nivel < len(B)
```

Más hermanos

```
1 s[nivel-1] <= 0
```

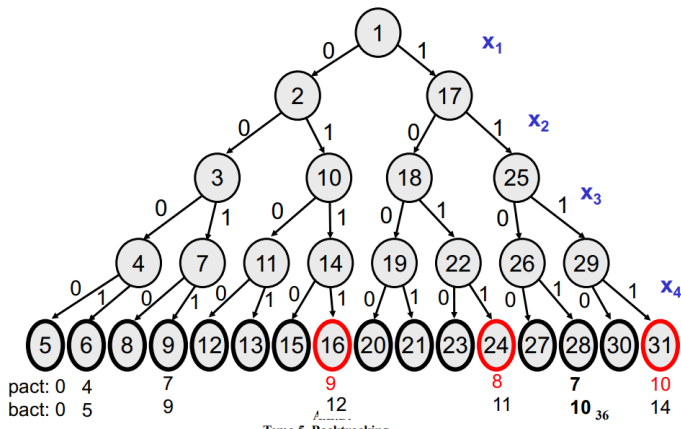
Retroceder

```
1 beneficio_actual -= s[nivel-1]*B[nivel-1]
2 peso_actual -= s[nivel-1]*P[nivel-1]
3 s[nivel - 1] = -1
4 nivel -= 1
```

```
1 def mochila(B, P, M):
2     nivel = 1
3     s = [-1]*len(B)
4     voa = -float("inf")
5     soa = None
6     peso_actual = 0
7     beneficio_actual = 0
8
9     while nivel != 0:
10        # generar
11        s[nivel - 1] += 1
12        peso_actual += P[nivel - 1]*s[nivel - 1]
13        beneficio_actual += B[nivel - 1]*s[nivel - 1]
14        # fin generar
15
16        # solucion
17        if peso_actual <= M and len(B) == nivel and beneficio_actual > voa:
18            voa = beneficio_actual
19            soa = s.copy()
20
21        # criterio
22        if peso_actual <= M and nivel < len(B):
23            nivel += 1
24        else:
25            # masHermanos y retroceder
26            while nivel > 0 and (not s[nivel-1] <= 0):
27                beneficio_actual -= s[nivel-1]*B[nivel-1]
28                peso_actual -= s[nivel-1]*P[nivel-1]
29                s[nivel - 1] = -1
30                nivel -= 1
31
32    return soa, voa
```

En muchos casos en los que la función criterio no poda nada y se explora todo el árbol...

$$B = (2, 3, 4, 5), P = (1, 2, 3, 4), M = 7$$



Tenemos que mejorar la función criterio, la idea principal es:

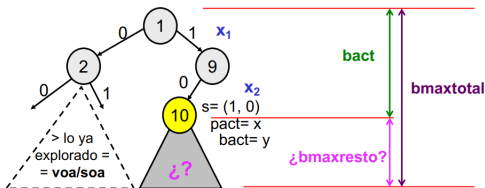
Cota superior fiable

Antes de meternos a explorar una rama, estimar el máximo beneficio que se *podría* llegar a obtener (cota superior fiable del beneficio).

Tenemos que mejorar la función criterio, la idea principal es:

Cota superior fiable

Antes de meternos a explorar una rama, estimar el máximo beneficio que se *podría* llegar a obtener (cota superior fiable del beneficio).



Si $voa \geq bmaxtotal = bact + bmaxresto$, esto implica que no sería inteligente explorar esa rama porque nunca superaríamos a la mejor solución que tenemos hasta ahora.

Buena cota superior fiable

Una buena cota superior fiable es aquella cota superior fiable que es ajustada y se puede calcular de manera rápida.

Ideas para $\text{bmaxresto}(k, B, P)$:

- Metemos todos los objetos restantes $k + 1, \dots, n$ que quedan ignorando el límite de capacidad.
- Usamos el algoritmo voraz de la mochila no 0/1 considerando $k + 1, \dots, n$ objetos, peso máximo $M - \text{peso_actual}$.

Lema

Supongamos los pesos P y los beneficios B . Sea s_1 la solución óptima al problema de la mochila 0/1 y s_2 la solución óptima al problema de la mochila no 0/1, entonces:

$$b(s_2) \geq b(s_1)$$

Demostración. Como s_1 es la solución óptima al problema de la mochila 0/1, es una solución válida al problema de la mochila no 0/1. Por tanto:

$$b(s_2) \geq b(s_1).$$

B enteros

Si los beneficios son enteros entonces:

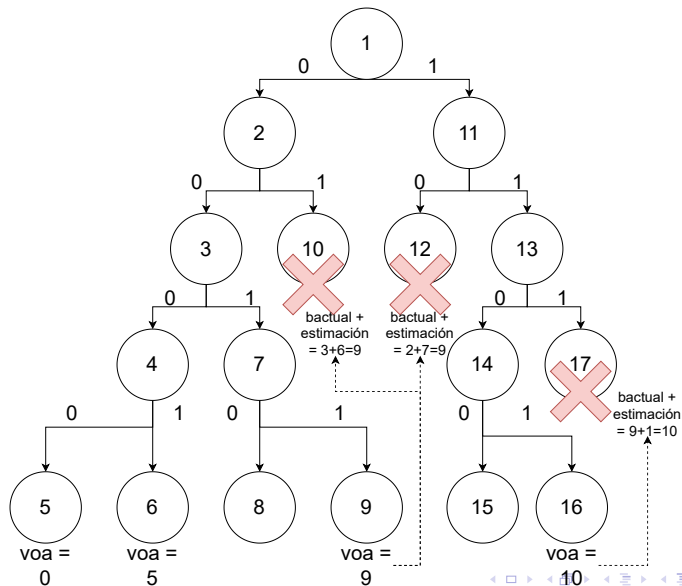
$$\lfloor b(s_2) \rfloor \geq b(s_1)$$

Esto quiere decir que $\text{bmaxresto}(k, B, P) = \text{int}(\text{voraz}(k, B, P))$ ¹ es una cota superior de la mochila 0/1 (con beneficios enteros) de los objetos y peso restantes. Así pues $\text{bact} + \text{bmaxresto}$ es una cota superior fiable del problema global.

Criterio

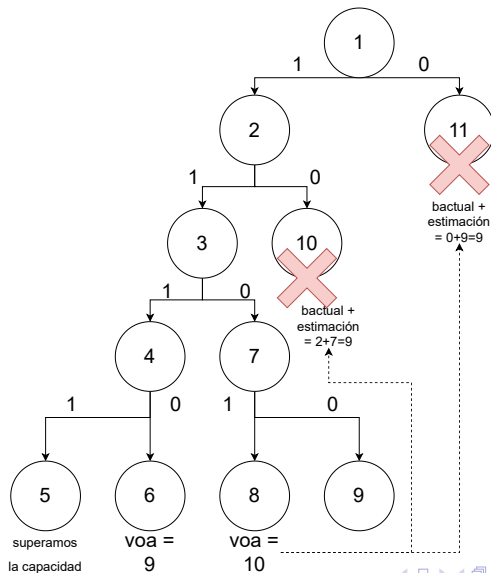
```
1 beneficio_estimado = int(mochila_voraz(set([i for i in range(nivel, len(B))]),
2 B, P, M - peso_actual))
3 peso_actual <= M and nivel < len(B) and (beneficio_estimado + beneficio_actual > voa)
```

¹`int(.)` devuelve el entero por debajo de .



Otra optimización más fina es ordenar y explorar el árbol de manera que se encuentre en las primeras hojas y que el criterio voraz pade más. Esto se puede conseguir haciendo lo siguiente:

- 1 Explorando primero la opción de meter un objeto. Es decir, primero explorando el hijo 1 y luego el 0.
- 2 Observando que la solución de la mochila 0/1 se suele *parecer* al de la mochila no 0/1 \Rightarrow ordenando los objetos usando b_i/p_i y empezando a explorar el primer objeto según ese criterio. Además, si ordenamos de esta manera el algoritmo voraz es $O(n)$.



Como ahora la función criterio utiliza la mochila voraz, el orden del algoritmo es $O(n2^n)$ que, a priori, es peor que el algoritmo sin optimizar $O(2^n)$. Sin embargo, en la práctica, como mochila voraz suele podar muy bien $t(n) \sim na^n$ con $a < 2$.

Ahora vamos a usar un árbol combinatorio. Hay que modificar generar, solución, criterio, más hermanos y retroceder.

Inicialización y variables auxiliares

```
1 s=[0]*len(B)
2 beneficio_actual = 0
3 peso_actual = 0
```

Generar

```
1 if s[nivel-1] == 0: # primer hermano?
2     if nivel == 1: # primer nivel?
3         s[nivel - 1] = 1
4         peso_actual += P[0]
5         beneficio_actual += B[0]
6     else:
7         # se genera el siguiente al padre
8         # y se actualiza el peso y beneficio actual
9         s[nivel - 1] = s[nivel - 2] + 1
10        peso_actual += P[s[nivel - 1] - 1]
11        beneficio_actual += B[s[nivel - 1] - 1]
12 else:
13     # generar el siguiente hermano
14     # se genera el siguiente al que había
15     # se actualizan las var aux restando y sumando
16     peso_actual -= P[s[nivel - 1] - 1]
17     beneficio_actual -= B[s[nivel - 1] - 1]
18     s[nivel - 1] += 1
19     peso_actual += P[s[nivel - 1] - 1]
20     beneficio_actual += B[s[nivel - 1] - 1]
```

Solución

```
1 peso_actual <= M # ya que todos los nodos son solución
```

Criterio

```
1 beneficio_estimado = int(mochila_voraz(set([i for i in range(s[nivel - 1], len(B))]), B, P,  
    M - peso_actual))  
2 peso_actual <= M and s[nivel-1] < len(B) and (beneficio_estimado + beneficio_actual > voa)
```

Más hermanos

```
1 s[nivel-1] < len(B)
```

Retroceder

```
1 beneficio_actual -= B[s[nivel - 1] - 1]  
2 peso_actual -= P[s[nivel - 1] - 1]  
3 s[nivel - 1] = 0  
4 nivel -= 1
```

Descripción del problema

- Tenemos n personas y n trabajos.
- Cada persona i realiza un trabajo j con un rendimiento $B[i][j]$.
- **Objetivo:** Asignar una tarea a cada trabajador que maximice el rendimiento.

		Tareas		
		1	2	3
Personas	B	4	9	1
	1	7	2	3
	2	6	3	5

Ejemplo 1. (P1, T1),
(P2, T3), (P3, T2)

$$B_{\text{TOTAL}} = 4 + 3 + 3 = 10$$

Ejemplo 2. (P1, T2),
(P2, T1), (P3, T3)

$$B_{\text{TOTAL}} = 9 + 7 + 5 = 21$$

Representación de la solución

Representación de la solución Vamos a representar la solución con $s = (x_1, \dots, x_n)$ tal que $x_i \in \{1 \dots, n\}$ indica el trabajo asignado a la persona i . Con la restricción de que $x_i \neq x_j$ para todo $i \neq j$. Esto forma un árbol permutacional. Se inicializa a $(0, \dots, 0)$.

Tipo de problema

Representación de la solución Vamos a representar la solución con $s = (x_1, \dots, x_n)$ tal que $x_i \in \{1 \dots, n\}$ indica el trabajo asignado a la persona i . Con la restricción de que $x_i \neq x_j$ para todo $i \neq j$. Esto forma un árbol permutacional. Se inicializa a $(0, \dots, 0)$.

Tipo de problema Es un problema de optimización (maximización). Así que escogemos el esquema backtracking de optimización.

Implementación de las funciones genéricas

Representación de la solución Vamos a representar la solución con $s = (x_1, \dots, x_n)$ tal que $x_i \in \{1 \dots, n\}$ indica el trabajo asignado a la persona i . Con la restricción de que $x_i \neq x_j$ para todo $i \neq j$. Esto forma un árbol permutacional. Se inicializa a $(0, \dots, 0)$.

Tipo de problema Es un problema de optimización (maximización). Así que escogemos el esquema backtracking de optimización.

Implementación de las funciones genéricas

- $s_0 = (0, \dots, 0)$
- `generar(nivel, s)`: genera el siguiente hermano $\sim ++s[\text{nivel}]$.
- `solución(nivel, s)`: ¿hemos analizado asignado trabajo a todos los trabajadores? ¿cumplimos la restricción del árbol permutacional?
- `criterio(nivel, s)`: ¿vamos cumpliendo las restricciones del árbol permutacional?
- `masHermanos(nivel, s)`: ¿hemos considerado todas las posibles asignaciones para el trabajador?
- `retroceder(nivel, s)`: pasamos al trabajador anterior dejando todo como estaba.

Esquema general (problema de optimización):

```
1 def backtracking_optimizacion_maximizar():
2     nivel = 1
3     s = s_0
4     voa = -float("inf")
5     soa = None
6
7     while nivel != 0:
8         # genero un hermano
9         generar(nivel, s)
10
11         # si he encontrado una solución que mejora a voa
12         # lo guardo
13         if solucion(nivel, s) and valor(s) > voa:
14             voa = valor(s)
15             soa = s
16
17         # si podríamos llegar a la solución final, seguimos
18         if criterio(nivel, s):
19             nivel += 1
20         else:
21             # en caso contrario, retrocedo hasta que encuentre un nodo con más hermanos
22             # por explorar
23             while nivel > 0 and (not masHermanos(nivel, s)):
24                 retroceder(nivel, s)
25     return soa
```


Inicialización y variables auxiliares

```
1 s=[0]*len(B)
2 beneficio_actual = 0
```

Generar

```
1 s[nivel - 1] += 1
2 if s[nivel - 1] == 1:
3     beneficio_actual += B[nivel - 1][s[nivel - 1] - 1]
4 else:
5     beneficio_actual += B[nivel - 1][s[nivel - 1] - 1]
6     beneficio_actual -= B[nivel - 1][s[nivel - 1] - 2]
```

Solución

```
1 def solucion(B, nivel, s):
2     if len(B) < nivel:
3         return False
4     for i in range(0, nivel - 1):
5         if s[i] == s[nivel - 1]:
6             return False
7     return True
```

Criterio

```
1 def criterio(B, nivel, s):
2     for i in range(0, nivel - 1):
3         if s[i] == s[nivel - 1]:
4             return False
5     return nivel < len(B)
```

Más hermanos

```
1 s[nivel-1] < len(B)
```

Retroceder

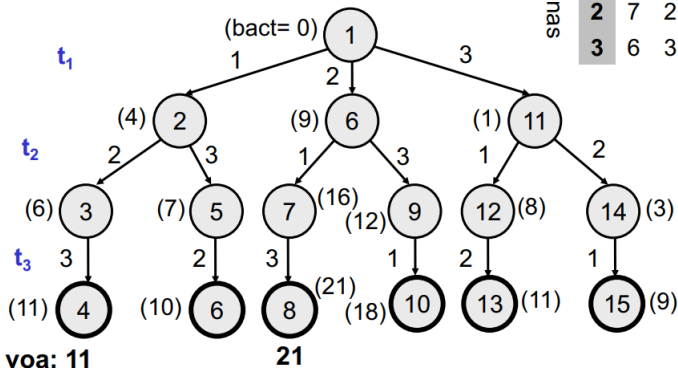
```
1 beneficio_actual -= B[nivel - 1][s[nivel - 1] - 1]
2 s[nivel - 1] = 0
3 nivel -= 1
```

5.3.2. Problema de asignación. Tareas

- Ejemplo de aplicación. $n = 3$

Personas

B	1	2	3
1	4	9	1
2	7	2	3
3	6	3	5



Como en la suma del subconjunto, el algoritmo se puede optimizar evitando el bucle de criterio y solución. La idea es llevar un array de enteros auxiliar que recoge las veces que una tarea i ha sido asignada.

Inicialización y variables auxiliares

```
1 s=[0]*len(B)
2 beneficio_actual = 0
3 asignadas = [0]*len(B)
```

Generar

```
1 s[nivel - 1] += 1
2 if s[nivel - 1] == 1:
3     beneficio_actual += B[nivel - 1][s[nivel - 1] - 1]
4 else:
5     asignadas[s[nivel-1]-2] -= 1
6     beneficio_actual += B[nivel - 1][s[nivel - 1] - 1]
7     beneficio_actual -= B[nivel - 1][s[nivel - 1] - 2]
8 asignadas[s[nivel-1]-1] += 1
```

Solución e inicialización

```
1 def solucion_optimizada(B, nivel, s, asignadas):
2     return asignadas[s[nivel - 1] - 1] == 1 and len(B) == nivel
3
4 def criterio_optimizada(B, nivel, s, asignadas):
5     return nivel < len(B) and asignadas[s[nivel - 1] - 1] == 1
```

Se tiene que recorrer todo el árbol. Siempre podemos intentar usar alguna cota superior fiable $b_{act} + b_{maxresto}$. Por ejemplo, $b_{maxresto}$ puede ser:

Para cada persona restante, asignar tarea restante con mayor rendimiento, sin marcarla como utilizada.

- Backtracking es un recorrido exhaustivo y sistemático del árbol de soluciones.
- Muy ineficientes: órdenes exponenciales o factoriales (debido al tamaño del los árboles).
- Pasos para aplicarlo:
 - 1 Determinar forma de la solución
 - 2 Escoger esquema según el tipo de problema (decisión, optimización o enumeración)
 - 3 Implementar las funciones genéricas
- Se pueden optimizar de dos formas:
 - 1 Minimizar el trabajo por nodo
 - 2 Buena función criterio que pode (esta suele impactar más en el rendimiento)