Unidad Didáctica I. Backtracking y Hashing
Tema 1. Backtracking (vuelta atrás)

Algoritmia

Profesor: Andrés Muñoz

Escuela Politécnica

Índice

- ✓ *Concepto de Backtracking*
- ✓ *Programación con Backtracking*
- ✓ *Ejemplos de algoritmos de Backtracking*
- ✓ *Branch and Bound*

Índice

Introducción

- ✓ Backtracking (o vuelta atrás) es una técnica algorítmica para hacer una búsqueda exhaustiva y sistemática por todas las configuraciones posibles del espacio de búsqueda del problema.
- ✓ Se suele aplicar en la resolución de un gran número de problemas, muy especialmente en los de **decisión** y **optimización**.
  - Problemas de Decisión: Búsqueda de las soluciones que satisfacen ciertas restricciones. *Ejemplo: Problema de n-reinas*
  - Problemas de Optimización: Búsqueda de la mejor solución en base a una función objetivo. *Ejemplo: Problema de la mochila*
- ✓ Los algoritmos de tipo Backtracking suelen ser muy ineficientes
  - Se utilizan para resolver problemas para los que no existe un algoritmo eficiente para resolverlos
  - El uso de programación paralela ayuda a mejorar la eficacia

¿Cómo funciona?

- ✓ De forma general, el método del backtracking prueba todas las posibles combinaciones de un problema de manera sistemática hasta que encuentra la correcta.

- ✓ La forma de actuar consiste en elegir una alternativa del conjunto de opciones en cada etapa del proceso de resolución, y si esta elección no funciona (no nos lleva a ninguna solución), la búsqueda vuelve al punto donde se realizó esa elección, e intenta con otra alternativa.

- ✓ Cuando se han agotado todas los posibles alternativas en una etapa, la búsqueda vuelve a la anterior fase en la que se hizo otra elección entre alternativas. Si no hay más puntos de elección, la búsqueda finaliza.

¿Cómo funciona?

- ✓ Expresado de forma simbólica, el funcionamiento de backtracking es el siguiente:
  1. Se parte de la suposición de que la solución de un problema de backtracking se puede expresar como una tupla $s = (v_1, v_2, \ldots, v_n)$, donde cada v_i es un valor de la solución.

     - Ejemplo de las n-reinas: Cada v_i es la posición de la reina i

     - Ejemplo de la mochila: Cada v_i es un elemento metido en la mochila

¿Cómo funciona?

2. En cada momento, el algoritmo se encontrará en un cierto nivel **k**, con una solución parcial $s_p= (v_1,v_2,\ldots,v_k)$, con ***k<=n***

   - Si puede añadirse un elemento v_{k+1} a la solución parcial se avanza al nivel k+1.
     - Ejemplo de las n-reinas: Puedo colocar otra reina en el tablero
     - Ejemplo de la mochila: Puedo colocar otro elemento en la mochila
   - Si no, se prueban otros valores válidos para v_k.
   - Si no existe ningún valor que sea válido por probar, se retrocede al nivel anterior k-1.
   - Se continua con este proceso hasta que
     - La solución parcial sea una solución del problema (una sola solución), o
     - Hasta que no queden más posibilidades por probar (en el caso de que no se encuentre ninguna solución o se busquen todas las soluciones del problema, o la más óptima).

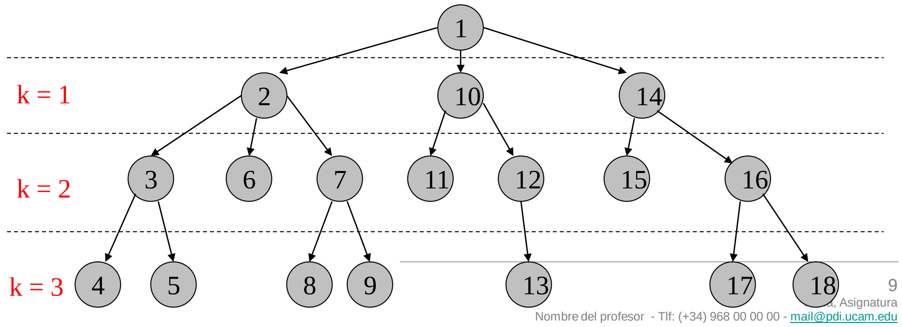
¿Cómo funciona?

- ✓ El resultado es equivalente a hacer un **recorrido en profundidad** en el árbol de soluciones. Sin embargo, este árbol es **implícito**, no se almacena en ningún lugar.
- ✓ Cada nodo del árbol del nivel *k* representa una parte de la solución y está formado por *k* valores de la solución que se consideran ya encontrados.
- ✓ Los hijos de un nodo del nivel *k* 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.

¿Cómo funciona?

✓ **Recorrido en profundidad** del árbol de soluciones ímplicito que realiza la técnica de backtracking.

- Los número de cada nodo marcan el recorrido del árbol
- En cada nodo habrá un vector de soluciones parcial $s_p = (v_1, v_2, \ldots, v_k)$ según el nivel k del árbol en donde se encuentre el nodo.
- Los nodos hoja pueden representar que no hay solución por ese camino <u>y hay que volver atrás</u>, o bien una solución parcial.

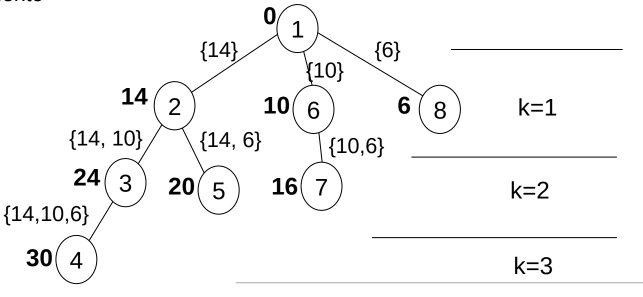


Un ejemplo sencillo

✓ Dado un conjunto de números enteros {14, 10, 6}, encontrar si existe algún subconjunto cuya suma sea exactamente 20. Dibujar el árbol de soluciones creado

Un ejemplo sencillo

- **Solución**. Dado un conjunto de números enteros {14, 10, 6}, encontrar si existe algún subconjunto cuya suma sea exactamente 20.
  - En cada nivel **k** hay que decidir qué elemento se añade: 14, 10 ó 6.
  - La solución se representa como s = (v_1, v_2 ..., v_m), donde $m \leq 3$ y $v_i \in$ {14, 10, 6}.
  - Los números en negrita son la suma parcial alcanzada hasta el momento
  - Cada serie entre llaves {} representa la solución parcial alcanzada hasta el momento



Un ejemplo sencillo

- Intentar resolver el mismo problema anterior, pero ahora los valores de la solución parcial deben ser 0 ó 1, para indicar si se toma o no el valor numérico correspondiente a esa posición.
- Por ejemplo, para los valores numéricos {14, 10, 6}, la solución parcial s_1 = *{1, 0, 1}* indica que es una solución en la que se han tomado los valores 14 y 6. En cambio, la solución parcial s_2 = *{0, 1, 1}* indica que se han tomado los valores 10 y 6, y la solución parcial s_3 = *{1, 1, 1}* indica que se han tomado todos los valores
- ¿Cómo cambia el árbol obtenido ahora con el obtenido en el ejercicio anterior?

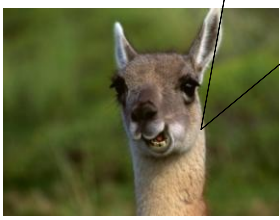
Índice

Programación Backtracking

- La forma más común de programar un algoritmo de backtracking es mediante una función recursiva, aunque también se puede implementar de forma iterativa.

Recursividad

“Para entender la recursividad, hay que entender antes qué es la recursividad”



Recursividad

```
int Misterio (int n) {
        int m;

        if (n==1)
                return 1;

        else {
                m = n / 2;
                return Misterio(m) + Misterio(n-m);
      }
}
```

Llamada inicial → Misterio (5)

Recursividad

Solución:

http://www.livescribe.com/cgi-bin/WebObjects/LDApp.woa/wa/MLSOverviewPage?sid=cC1CxfxPr98k

Programación Backtracking

✓ Esquema general del algoritmo backtracking de forma recursiva (**todas las soluciones**).

```
función BTRec(salida solucion[1..n], int etapa)  // solucion es un vector de dimensión n
   Si (etapa > n) entonces return;
   valores[1..m];                                // valores es un vector de dimensión m
   iniciarValores(valores, etapa);
   repetir
         nuevoValor  =  seleccionarNuevoValor(valores);
         Si alcanzable(solucion, nuevoValor)  entonces
               solucion[etapa] = nuevoValor;
               Si  esSolucion(solucion) entonces
                     procesarSolucion(solucion);
               Si No
                     BTRec(solucion, etapa + 1);
               finSI
               solucion[etapa] = ∅;   // Se elimina nuevoValor de la solución
         finSI
   hasta ultimoValor(valores);fin fun
```

Programación Backtracking

- ✓ iniciarValores(valores, etapa)
  - Genera todos los posibles valores de solución de la etapa indicada
- ✓ seleccionarNuevoValor(valores)
  - Devuelve un nuevo valor de los generados con iniciarValores()
- ✓ alcanzable(nuevoValor)
  - Comprueba si la opción de nuevoValor puede forma parte de la solución, es decir, no incumple ninguna de las restricciones indicadas en el problema
- ✓ esSolucion(solucion)
  - Indica si solucion es una solución para el problema
- ✓ ultimoValor(valores)
  - Indica si ya no quedan más valores de solución (nodos) por expandir

Programación Backtracking

- ✓ El esquema anterior encuentra todas las posibles soluciones del problema.
  - La función *procesarSolucion()* debe ir almacenando las soluciones encontradas
- ✓ Este esquema también sirve para encontrar la mejor solución.
  - En este caso la función *procesarSolucion()* debe ir comparando las soluciones encontradas y quedarse con la mejor.
- ✓ Si es suficiente con la primera solución encontrada, la función *esSolucion()* debe devolver un booleano en una variable llamada "*éxito*" que indique si ya se ha encontrado la solución o no.
- ✓ En caso afirmativo, se utiliza la variable "éxito" para no seguir explorando el resto de valores de cada nivel. Esta variable se debe devolver a cada llamada recursiva de la función recursiva *BTRec()*

Programación Backtracking

✓ Esquema general del algoritmo backtracking de forma recursiva (**sólo la primera solución**).

```
función BTRec(salida solucion[1..n], int etapa): boolean    // la función devuelve un boolean
    valores[1..m]                                          // valores es un vector de dimensión m
    Si (etapa > n) entonces return falso;
    exito = falso                    // exito indica si se ha encontrado la primera solución
    iniciarValores(valores, etapa)
    repetir
        nuevoValor = seleccionarNuevoValor(valores)
        Si alcanzable(solucion, nuevoValor) entonces
            solucion[etapa] = nuevoValor
            Si esSolucion(solucion) entonces
                procesarSolucion(solucion)
                exito = verdadero                    // solución encontrada
            Si No
                exito = BTRec(solucion, etapa + 1)   // devuelve si se ha encontrado la solución
            finSI
            SI exito = falso entonces solucion[etapa] = ∅ finSI    // Se elimina nuevoValor
        finSI
    hasta (exito = verdadero) o (ultimoValor(valores)) devolver exito
fin fun
```


Programación con Backtracking

✓ Ejercicio 1. Ejecutar el algoritmo de backtracking recursivo para todas las soluciones posible con el ejercicio en el que dado un conjunto de números enteros {14, 10, 6}, encontrar si existe algún subconjunto cuya suma sea exactamente 20

✓ Ejercicio 2. Ejecutar el algoritmo de backtracking recursivo para la primera solución con el ejercicio en el que dado un conjunto de números enteros {14, 10, 6}, encontrar si existe algún subconjunto cuya suma sea exactamente 20

✓ Para cada ejercicio, mostrar los valores de las variables solucion, etapa y nuevosValores y el resultado de las funciones en cada llamada recursiva del algoritmo.

Solución Ejercicio 1

```
BTRec([-,-,-], 1)
   valores = {14, 10, 6)

Iteración 1
   nuevoValor = 14
   alcanzable: 14 < 20, OK
   solucion = [14,-,-]
   esSolucion: 14 ≠ 20, NO
   BTRec([14,-,-], 2)
```


Solución Ejercicio 1

```
BTRec([-,-,-], 1)
   valores = {14, 10, 6)

Iteración 1
   nuevoValor = 14
   alcanzable: 14 < 20, OK
   solucion = [14,-,-]
   esSolucion: 14 ≠ 20, NO
   BTRec([14,-,-], 2)
```

```
BTRec([14,-,-], 2)
   valores = {10, 6)
Iteración 1
   nuevoValor = 10
   alcanzable: 14 + 10 > 20, NO

Iteración 2
   nuevoValor = 6
   alcanzable: 14 + 6 = 20, SI
   solucion = [14,6,-]
   esSolucion: 14 = 20, SI
```

Solución Ejercicio 1

BTRec([-,-,-], 1)
valores = {14, 10, 6}

Iteración 1
nuevoValor = 14
alcanzable: 14 < 20, OK
solucion = [14,-,-]
esSolucion: 14 ≠ 20, NO
BTRec([14,-,-], 2)
solucion = [-,-,-]

Iteración 2
nuevoValor = 10
alcanzable: 10 < 20, OK
solucion = [10,-,-]
esSolucion: 10 ≠ 20, NO
BTRec([10,-,-], 2)

BTRec([14,-,-], 2)
valores = {10, 6}
Iteración 1
nuevoValor = 10
alcanzable: 14 + 10 > 20, NO

Iteración 2
nuevoValor = 6
alcanzable: 14 + 6 = 20, SI
solucion = [14,6,-]
esSolucion: **14 = 20, SI**

Solución Ejercicio 1

BTRec([-,-,-], 1)
valores = {14, 10, 6}

Iteración 1
nuevoValor = 14
alcanzable: 14 < 20, OK
solucion = [14,-,-]
esSolucion: 14 ≠ 20, NO
BTRec([14,-,-], 2)
solucion = [-,-,-]

Iteración 2
nuevoValor = 10
alcanzable: 10 < 20, OK
solucion = [10,-,-]
esSolucion: 10 ≠ 20, NO
BTRec([10,-,-], 2)

BTRec([14,-,-], 2)
valores = {10, 6}
Iteración 1
nuevoValor = 10
alcanzable: 14 + 10 > 20, NO

Iteración 2
nuevoValor = 6
alcanzable: 14 + 6 = 20, SI
solucion = [14,6,-]
esSolucion: **14 = 20, SI**

BTRec([10,-,-], 2)
valores = {6}
Iteración 1
nuevoValor = 6
alcanzable: 10 + 6 < 20, SI
solucion = [10,6,-]
esSolucion: 16 ≠ 20, NO
BTRec([10,6,-], 3)

Solución Ejercicio 1

BTRec([-,-,-], 1)
valores = {14, 10, 6)

Iteración 1
nuevoValor = 14
alcanzable: 14 < 20, OK
solucion = [14,-,-]
esSolucion: 14 ≠ 20, NO
BTRec([14,-,-], 2)
solucion = [-,-,-]

Iteración 2
nuevoValor = 10
alcanzable: 10 < 20, OK
solucion = [10,-,-]
esSolucion: 10 ≠ 20, NO
BTRec([10,-,-], 2)

BTRec([14,-,-], 2)
valores = {10, 6)
Iteración 1
nuevoValor = 10
alcanzable: 14 + 10 > 20, NO

Iteración 2
nuevoValor = 6
alcanzable: 14 + 6 = 20, SI
solucion = [14,6,-]
esSolucion: **14 = 20, SI**

BTRec([10,-,-], 2)
valores = {6)
Iteración 1
nuevoValor = 6
alcanzable: 10 + 6 < 20, SI
solucion = [10,6,-]
esSolucion: 16 ≠ 20, NO
BTRec([10,6,-], 3)
solucion = [10,-,-]

BTRec([10,6,-], 3)
valores = {∅}
No hay iteraciones

Solución Ejercicio 1

BTRec([-,-,-], 1)
valores = {14, 10, 6)

Iteración 1
nuevoValor = 14
alcanzable: 14 < 20, OK
solucion = [14,-,-]
esSolucion: 14 ≠ 20, NO
BTRec([14,-,-], 2)
solucion = [-,-,-]

Iteración 2
nuevoValor = 10
alcanzable: 10 < 20, OK
solucion = [10,-,-]
esSolucion: 10 ≠ 20, NO
BTRec([10,-,-], 2)
solucion = [-,-,-]

Iteración 3
continua sig. diapositiva

BTRec([14,-,-], 2)
valores = {10, 6)
Iteración 1
nuevoValor = 10
alcanzable: 14 + 10 > 20, NO

Iteración 2
nuevoValor = 6
alcanzable: 14 + 6 = 20, SI
solucion = [14,6,-]
esSolucion: **14 = 20, SI**

BTRec([10,-,-], 2)
valores = {6)
Iteración 1
nuevoValor = 6
alcanzable: 10 + 6 < 20, SI
solucion = [10,6,-]
esSolucion: 16 ≠ 20, NO
BTRec([10,6,-], 3)
solucion = [10,-,-]

BTRec([10,6,-], 3)
valores = {∅}
No hay iteraciones

Solución Ejercicio 1

BTRec([-,-,-], 1)
valores = {14, 10, 6}

<u>Iteración 3</u>
nuevoValor = 6
alcanzable: 6 < 20, OK
solucion = [6,-,-]
esSolucion: 6 ≠ 20, NO
BTRec([6,-,-], 2)

BTRec([6,-,-], 2)
valores = {∅}
No hay iteraciones

Solución Ejercicio 1

BTRec([-,-,-], 1)
valores = {14, 10, 6}

<u>Iteración 3</u>
nuevoValor = 6
alcanzable: 6 < 20, OK
solucion = [6,-,-]
esSolucion: 6 ≠ 20, NO
BTRec([6,-,-], 2)
solución[-,-,-]

Fin de Iteraciones

BTRec([6,-,-], 2)
valores = {∅}
No hay iteraciones

Programación con Backtracking

- ✓ También es posible escribir el algoritmo de backtracking de manera **iterativa.**
- ✓ Se propone al alumno que busque el esquema general de este algoritmo y lo explique.
- ✓ También se propone que repita los ejercicios 1 y 2 anteriores para este algoritmo iterativo.

Complejidad de Backtracking

- ✓ Por realizar una búsqueda exhaustiva en el espacio de soluciones del problema, los algoritmos de backtracking son bastante ineficientes.
- ✓ En general, se tienen tiempos con órdenes de complejidad factoriales o exponenciales.
- ✓ Por esto, los algoritmos de backtracking se utilizan en problemas para los que no existen un algoritmo eficiente que los resuelva.
- ✓ También tenemos que tener en cuenta que la recursividad contribuye a su ineficiencia, por la memoria utilizada durante las diferentes llamadas recursivas.

Complejidad de Backtracking

- ✓ En particular, para calcular la complejidad de backtracking tenemos que tener en cuenta lo siguiente:
  - El número de nodos del árbol de búsqueda que se visitan para conseguir la solución →*v(n).*
  - El trabajo realizado en cada nodo, esto es, el coste de la función de solución completa o ver si la solución es aceptable hasta el momento. Este coste lo podemos expresar como *p(n),* ya que generalmente será polinómico.
  - **El coste en general será**: *O(p(n)·v(n))*
  - Suponiendo que una solución sea de la forma: (x_1, x_2, ..., x_n), en el peor caso se generarán todas las posibles combinaciones para cada $\mathbf{x_i}$.
    - Si el número de posibles valores para cada $\mathbf{x_i}$ es $\mathbf{m_i}$, entonces se generan:

m_1	nodos en el nivel 1
$m_1 \cdot m_2$	nodos en el nivel 2
...	....
$m_1 \cdot m_2 \cdot \ldots \cdot m_n$	nodos en el nivel n

UCAM | UNIVERSIDAD CATÓLICA SAN ANTONIO

Complejidad de Backtracking

- ✓ **Ejemplo:** para el problema de la suma de números en subconjuntos con número de nodos del árbol m = 2. El número de nodos generados es:

  $t(n) = 2 + 2^2 + 2^3 + \ldots + 2^n = 2^{n+1} - 2$

- ✓ **Ejemplo**: calcular todas las permutaciones de (1, 2, ..., n). En el nivel 1 tenemos n posibilidades, en el nivel 2 n-1, ..., en el nivel n una posibilidad.

  $t(n) = n + n\cdot(n-1) + n\cdot(n-1)\cdot(n-2) + \ldots + n! \in O(n!)$

Ventajas e inconvenientes

Ventajas	Inconvenientes
Si existe una solución, la encuentra	Coste exponencial en la mayoría de los casos
Es un esquema sencillo de implementar	Si el espacio de búsqueda es infinito es posible que no se encuentre la solución aunque exista
Adaptable a las características específicas de cada problema	Consume mucha memoria al tener que almacenar las llamadas recursivas

Antes de programar…

✓ Cuestiones a resolver antes de ponernos a programar:

- ¿Qué tipo de árbol es adecuado para el problema?
  - ¿Cómo es la representación de la solución (tupla)?
  - ¿Qué indica cada $\mathbf{x_i}$ y qué valores puede tomar?
- ¿Cómo generar un recorrido según ese árbol?
  - Generar un nuevo nivel.
  - Generar los hermanos de un nivel.
  - Retroceder en el árbol.
  - Estas cuestiones se solucionan usando el esquema general recursivo o iterativo
- ¿Qué ramas se pueden descartar por no conducir a soluciones del problema?
  - Poda por restricciones del problema.
  - Poda según el criterio de la función objetivo (para problemas de optimización) → *Branch & Bound*.

Antes de programar…

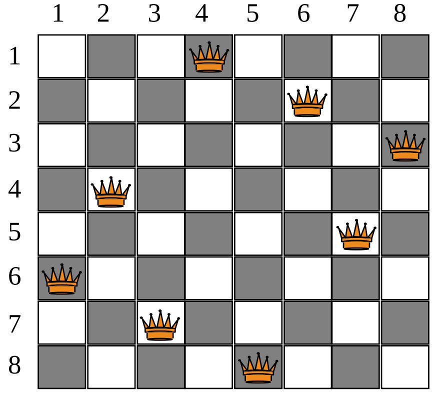
- ✓ Aplicación de backtracking (proceso metódico):
  1. Determinar cómo es la forma del árbol de backtracking, o lo que es lo mismo, cómo es la representación de la solución.
  2. Elegir el esquema de algoritmo adecuado, adaptándolo en caso necesario (no reinventar la rueda!!)
  3. Implementar las funciones genéricas para la aplicación concreta: según la forma del árbol y las características del problema.
  4. Posibles mejoras: usar variables locales con "valores acumulados", hacer más podas del árbol, etc.

Índice

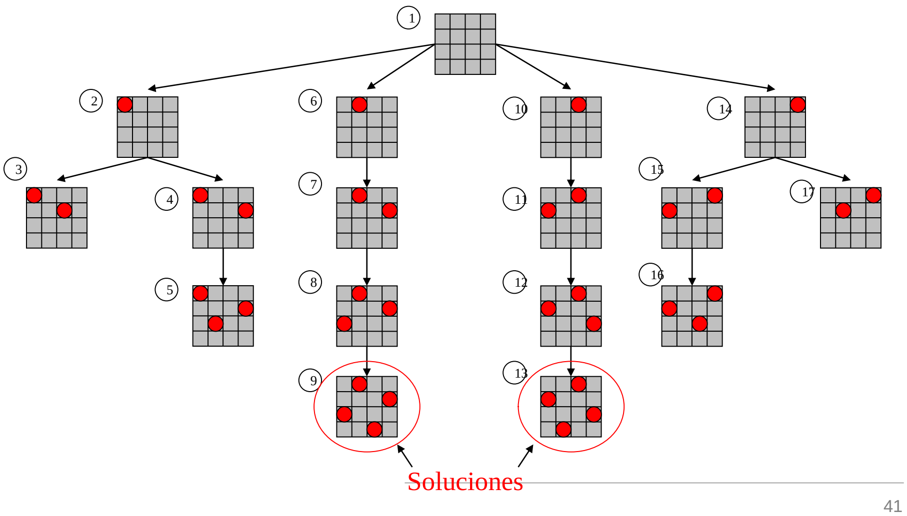
Problema de las n-reinas

- ✓ El problema de las n-Reinas consiste en colocar n reinas en un tablero de ajedrez de tamaño n*n de forma la reinas no se amenacen según las normas del ajedrez
- ✓ Se busca encontrar una solución o todas las soluciones posibles
- ✓ Cualquier solución del problema estará formada por una n-tupla $(x_1,x_2,\ldots,x_n)$, dónde cada x_i indica la columna donde la reina de la fila i-ésima es colocada
- ✓ Las restricciones para este problema consisten en que dos reinas no pueden colocarse en la misma fila, ni en la misma columna ni en la misma diagonal.
- ✓ Por ejemplo, el problema de las 4-Reinas tiene dos posibles soluciones: [2,4,1,3] y [3,1,4,2].

Problema de las n-reinas



Problema de las n-reinas



Problema de las n-reinas

✓ Pseudocódigo para encontrar la primera solución:

```
CONST n = ...; (* numero de reinas; n>3 *)
Function Reinas(salida Solucion = array[1..n] of int, k: int ): boolean {
    IF k>n THEN RETURN false;  // No se encontró solución
    exito = false;
    Solucion[k]:=0;
    REPEAT
        Solucion[k] = Solucion[k] +1;    * seleccion de nueva opcion *
        IF Valido(Solucion, k) THEN    * prueba si la solución parcial es válida *
                IF k<>n THEN
                        éxito:= Reinas(solucion, k+1)        * llamada recursiva *
                ELSE
                        exito:=true
                END
        END
    UNTIL (Solucion[k]=n) OR exito;
    RETURN exito;
}
```

Problema de las n-reinas

✓ Pseudocódigo para encontrar la primera solución:

```
Function Valido(Solucion = array[1..n] of int,  k: int): boolean;
    (* comprueba si el vector solucion construido hasta el paso k es k-prometedor, es decir,
    si la reina puede situarse en la columna k *)
    FOR i:=1 TO k-1 DO
      IF (Solucion[i]==Solucion[k]) OR (ValAbs(Solucion[i],Solucion[k])==ValAbs(i,k)) THEN
            RETURN false;
      END_IF
    END_FOR
    RETURN true;
}
```

✓ La función **ValAbs(x,y)** devuelve el valor absoluto de la resta del primer parámetro menos el segundo: | x –y |

Problema de las n-reinas

✓ La solución anterior sólo encuentra una solución

✓ Se propone a los alumnos que modifiquen el esquema dado para encontrar todas las soluciones

✓ Se propone también que los alumnos encuentren el esquema iterativo para la solución del problema de las n-reinas

Problema de las n-reinas

✓ Cálculo de la complejidad:

- Puesto que no puede haber más de una reina por columna, sólo hace falta que consideremos las n-tuplas $(x_1,\ldots,x_n)$ que sean permutaciones de (1,2,...,n)
- El espacio de soluciones costa de *n!* n-tuplas.
- Ejemplo: Para las 8-reinas, el espacio de búsquedas es

8! 8-tuplas = 40.320 8-tuplas

- Existen algunos métodos para mejorar el coste de la búsqueda:
  - Comenzar a buscar por donde existen menos alternativas para explorar
  - Eliminar simetrías de tablero
  - Realizar una comprobación hacia adelante para ver cuál de las posibles soluciones es la más prometedora.

Problema de las n-reinas

✓ Otros problemas similares al de n-reinas (problemas de decisión)

- Problema del laberinto: El laberinto se representa como una matriz de NxM casillas, y cada valor de la solución es una casilla utilizada en el camino
- Problema de la suma de subconjuntos
- Recorrido del caballo de ajedrez por el tablero.
- Cuadrado mágico: es la disposición de una serie de números enteros en un cuadrado o matriz de forma tal que la suma de los números por columnas, filas y diagonales sea la misma, la constante mágica

4	9	2
3	5	7
8	1	6

Problema de la mochila

- ✓ Se tienen *n* objetos y una mochila con capacidad *C*.
- ✓ El objeto *i* tiene peso p_i y la inclusión del objeto *i* en la mochila produce un beneficio b_i.
- ✓ El objetivo es llenar la mochila sin exceder la capacidad *C*, de manera que se maximice el beneficio.

$$\text{maximizar} \quad \sum_{1 \le i \le n} b_i x_i$$

$$\text{sujeto a} \quad \sum_{1 \le i \le n} p_i x_i \le C$$

$$\text{con} \quad x_i \in \{0,1\},\ b_i > 0,\ p_i > 0,\ 1 \le i \le n$$

Problema de la mochila

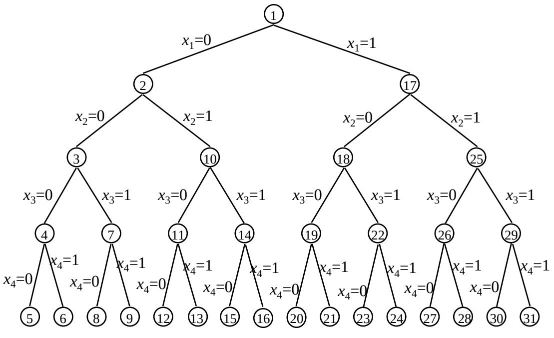
- ✓ La solución del problema se expresa mediante una n-tupla X = (x_1,x_2,…,x_n), dónde cada x_i toma los valores 1 ó 0 dependiendo de si el *i*-ésimo objeto se introduce en la mochila o no.
- ✓ Al ser un problema de optimización, **sólo nos interesa la mejor solución.**
  - En todo momento se debe guardar el coste de la mejor solución encontrada hasta el momento
  - Posible mejora de eficiencia:
    - No seguir explorando nodos del siguiente nivel si en el nodo actual hemos sobrepasado el peso de la mochila
    - Expandir sólo aquellas soluciones que puedan mejorar con respecto a la mejor solución actual.

Problema de la mochila

- ✓ Representación del árbol de espacios

  x_i=0 si el objeto i-ésimo no se introduce

  x_i=1 si el objeto se introduce



UCAM | UNIVERSIDAD CATÓLICA SAN ANTONIO

Problema de la mochila

- ✓ Algoritmo iterativo de backtracking para la mochila (I)

CONST n = ...; * numero de objetos *
CONST CAPACIDAD = ...; * capacidad de la mochila *

TYPE registro = **RECORD** peso, beneficio: float **END**;
objetos = **ARRAY**[1..n] **OF** registro;
mochila = **ARRAY**[1..n] **OF** int;

Problema de la mochila

✓ Algoritmo iterativo de backtracking para la mochila (II)

Function *Mochila(entrada* elem: objetos, *salida* sol :mochila, *salida* peso_final: **float**, *salida* beneficio_final: **float**) {

```
peso_actual, beneficio_actual: float;
sol_actual: mochila;
k: int;
peso_en_curso:=0.0;  beneficio_en_curso:=0.0;
sol_actual:= [-1, -1, …, -1]  // Iniciar a -1 los n valores de la solución
k:=1;
REPEAT
    sol_actual[k]:= Generar (k, sol_actual);    // Devuelve 0 ó 1
    IF sol_actual[k] == 1 THEN    // Se introduce el objeto k en la mochila
            peso_actual:= peso_actual + elem[k].peso;
            beneficio_actual:= beneficio_actual + elem[k].beneficio;
   END_IF
   IF Solucion (k, peso_actual) THEN
            IF beneficio_actual > beneficio_final THEN
                  beneficio_final:= beneficio_actual;
                  peso_final:= peso_actual;
                  sol:= sol_actual;
            END_IF
   …
```


Problema de la mochila

✓ Algoritmo iterativo de backtracking para la mochila (III)

Function *Mochila(entrada* elem: objetos, *entrada* capacidad: **float**, *salida* sol :mochila, *salida* peso_final: **float**, *salida* beneficio_final: **float**) {

```
…
REPEAT
    …
    ELSE IF ExplorarSigNivel(k, peso_actual)
        k = k + 1;
    END_IF
    WHILE k > 0 AND sol_actual[k] ==1 DO    // No se puede avanzar más
        peso_actual = peso_actual – sol_actual[k]*elem[k].peso;
        beneficio_actual = beneficio_actual – sol_actual[k]*elem[k].beneficio;
        sol_actual[k] = -1;
        k = k – 1;
    END_WHILE
UNTIL k == 0;
}
```

Problema de la mochila

- ✓ Algoritmo iterativo de backtracking para la mochila (IV)

** Esta función devuelve 0 si todavía no se ha explorado el objeto del nivel k y 1 en caso contrario **

```
Function Generar(k: int, sol_actual: mochila): int {
        IF sol_actual[k] == -1 THEN
            RETURN 0;
        ELSE
            RETURN 1;
}
```

** Esta función devuelve TRUE si se han explorado todos los objetos (k == n) y el peso actual es menor o igual a la capacidad de la mochila. Devuelve FALSE si alguna de estas dos condiciones no se cumple **

```
Function Solucion(k: int, peso_actual: float): boolean {
        RETURN (k == n) AND (peso_actual <= CAPACIDAD);
}
```


Problema de la mochila

- ✓ Algoritmo iterativo de backtracking para la mochila (V)

** Esta función devuelve TRUE si no se han explorado todos los objetos (k < n) y el peso actual es menor o igual a la capacidad de la mochila. Devuelve FALSE si alguna de estas dos condiciones no se cumple **

```
Function ExplorarSigNivel (k: int, peso_actual: float): boolean {
        RETURN (k < n) AND (peso_actual <= CAPACIDAD);
}
```

- ✓ La llamada al algoritmo de backtracking *Mochila* será de la siguiente manera:

    Mochila(elem, sol, peso, beneficio);

donde:

- **elem** será un vector de tipo "objetos" inicializado con los elementos (peso y beneficio) a introducir en la mochila
- **sol** será un puntero al tipo mochila
- **peso** y **beneficio** serán dos punteros a variables de tipo float

Problema de la mochila

- ✓ Observar que al ser un problema de optimización el algoritmo no acaba hasta que se hayan recorrido todos los nodos. Por tanto, acaba cuando el nivel alcanzado en el árbol es la raíz (k = 0).
- ✓ El bucle WHILE interno de la función *Mochila* permite volver hacía atrás en el árbol de búsqueda cuando un objeto se ha explorado completamente (tanto fuera como dentro de la mochila)

- ✓ **Ejercicio**: Aplicar el algoritmo Mochila para el siguiente problema:

  n = 3

  CAPACIDAD= 7kgs.

  peso = {2,3,4}   beneficio = {3,4,5}

Problema de la mochila

- ✓ Nodos generados = 2^{n+1}-1. El algoritmo es de complejidad $O(2^n)$.
  - Para 1 objeto se generan 3 nodos (raíz + x_1 = 0 + x_1 = 1)
  - Para 2 objetos se generan 7 nodos
  - Para 3 objetos se generan 15 nodos
  - Para n objetos se generan 2^{n+1}-1 nodos
- ✓ **Problema**: Con este algoritmo se generan todos los nodos posibles. La función *ExplorarSigNivel()* es siempre cierta excepto para los nodos hoja
- ✓ **Solución**: Intentar eliminar algunos (podar) nodos del árbol de soluciones, con una función *ExplorarSigNivel()* más restrictiva.
  - Para cada nodo, hacer una estimación del máximo beneficio que se podría obtener a partir del mismo.
  - Si esta estimación es menor que el mayor beneficio de la solución actual (beneficio_final) entonces rechazar ese nodo y sus descendientes.

Problema de la mochila

- ✓ La estimación del beneficio para el nivel y nodo actual será:

  beneficio_estimado = beneficio_actual+ *Estimacion* (k + 1, CAPACIDAD – peso_actual)

- ✓ **Estimacion (k, Q)**: Estimar una cota superior de beneficio para el problema de la mochila, usando los objetos de **k hasta n** con una mochila de capacidad máxima **Q**.
- ✓ La función *Estimación* nos va a permitir realizar la poda del árbol para aquellos nodos que no lleven a la solución óptima.
  - Vamos a considerar que los objetos en el vector *elem* están ordenados de forma decreciente por su ratio beneficio/peso.
  - Supongamos que nos encontramos en el nivel *k-ésimo,* y que disponemos de un beneficio acumulado Bk. Sabemos que

$$B_k = \sum_{i=1}^{k} sol[i] * elem[i].beneficio$$

UCAM | UNIVERSIDAD CATÓLICA SAN ANTONIO

Problema de la mochila

- ✓ Para calcular el valor máximo (B_M) que podríamos alcanzar desde este nivel *k* vamos a suponer que rellenáramos el resto de la mochila con el mejor de los elementos que nos quedan por analizar.
- ✓ Como tenemos los objetos dispuestos en orden decreciente de ratio beneficio/peso, este mejor elemento será el siguiente (*k+1).*
- ✓ Este valor de beneficio, aunque no tiene por qué ser alcanzable, nos permite dar una cota superior del valor al que podemos "aspirar" si seguimos por esa rama del árbol:

$$B_M = B_k + \left(capacidad - \sum_{i=1}^{k} sol[i] * elem[i].peso \right) \frac{elem[k+1].beneficio}{elem[k+1].peso}$$

Problema de la mochila

✓ La implementación de esta mejora requiere las siguientes modificaciones:

- <u>En la función *ExplorarSigNivel*:</u>

** Esta función devuelve TRUE si no se han explorado todos los objetos (k < n) y el peso actual es menor o igual a la capacidad de la mochila, y además el beneficio estimada es mayor que el beneficio máximo alcanzado. Devuelve FALSE si alguna de estas condiciones no se cumple **

```
Function ExplorarSigNivel (k: int, peso_actual: float, beneficio_actual float,
                           beneficio_final: float): boolean {
   beneficio_estimado: float;
   IF (peso_actual > CAPACIDAD) OR (k == n)  THEN RETURN false;
   END_IF
   beneficio_estimado:= beneficio_actual + Estimacion(k+1, CAPACIDAD- peso_actual);
   RETURN beneficio_estimado > beneficio_final;
}
```


Problema de la mochila

✓ La implementación de esta mejora requiere las siguientes modificaciones:

- <u>Nueva función *Estimacion*:</u>

** Esta función devuelve TRUE si no se han explorado todos los objetos (k < n) y el peso actual es menor o igual a la capacidad de la mochila, y además el beneficio estimada es mayor que el beneficio máximo alcanzado. Devuelve FALSE si alguna de estas condiciones no se cumple **

```
Function Estimacion (k: int, capac: float): float {
  beneficio_acum, peso_acum: float;
  beneficio_acum = 0.0; peso_acum: 0.0;
  FOR i:= k TO n DO
    peso_acum:= peso_acum + elem[k].peso;
    IF peso_acum < capac THEN beneficio_acum:= beneficio_acum + elem[k].beneficio;
    ELSE
          RETURN (beneficio_acum + (1.0 – (peso_acum-capac) / elem[i].peso)* (elem[i].beneficio));
    END_IF
    RETURN beneficio_acum;
}
```

Problema de la mochila

- La implementación de esta mejora requiere las siguientes modificaciones:
  - En la función *Mochila*:

```
…
REPEAT
…
    ELSE ExplorarSigNivel(k, peso_actual, beneficio_actual, beneficio_final)
        k = k + 1;
    END_IF
    WHILE (k > 0 AND sol_actual[k] ==1)   DO    // No se puede avanzar más
                peso_actual = peso_actual – sol_actual[k]*elem[k].peso;
                beneficio_actual = beneficio_actual – sol_actual[k]*elem[k].beneficio;
                sol_actual[k] = -1;
                k = k – 1;
    END_WHILE
UNTIL k == 0;
```


Problema de la mochila

- **Ejercicio**: Repetir el ejercicio anterior con esta versión de mochila.
- Con la poda realizada mediante la función *Estimacion()* se eliminan nodos pero a costa de aumentar el tiempo de ejecución en cada nodo.
- ¿Cuál será el tiempo de ejecución total?
  - Suponiendo los objetos ordenados por $beneficio_i/peso_i$
  - Tiempo de la función *ExplorarSigNivel* en el nivel **i** (en el peor caso)
    1 + Tiempo de *Estimacion()* = 1 + n - i
  - Entonces, tiempo en el peor caso: Número de nodos * Tiempo de cada nodo (*ExplorarSigNivel*) = $O(2^n)$ * $O(n)$ = $O(n \cdot 2^n)$… ¿peor tiempo que sin realizar la poda????.
  - NO, este cálculo no es correcto. Demonstración:

$$t(n) = \sum_{i=1}^{n} 2^i \cdot (n - i + 1) = (n+1)\sum_{i=1}^{n} 2^i - \sum_{i=1}^{n} i \cdot 2^i = 2 \cdot 2^{n+1} - 2n - 4$$

Problema de la mochila

- ✓ En el peor caso, el orden de complejidad sigue siendo un $O(2^n)$.
  - En promedio se espera que la poda elimine muchos nodos, reduciendo el tiempo total.
  - Pero el tiempo sigue siendo muy malo. ¿Cómo mejorarlo?

- ✓ Posible modificación: Generar primero los nodos con valores 1 y después con valores 0 (invertir el orden de los valores generados de cada x_i)

- ✓ Posible modificación: Representar la solución de manera que cada x_i represente el identificador del objeto introducido en la mochila

  Solución = $(x_1, x_2, ..., x_m)$ donde $m \leq n$ y $x_i \in \{1, 2, ..., n\}$, los identificadores de los objetos

  ¿Qué complejidad tendrá esta solución?

Problema de la mochila

- ✓ Otros problemas similares al de la mochila (problemas de optimización)
  - **Parejas estables**: *n* hombres y *n* mujeres, y dos matrices M y H que contienen las preferencias de los unos por los otros. El problema es encontrar el emparejamiento entre hombres y mujeres de forma que las parejas formadas sean estables (son las mejores parejas posibles)
  - **Asignación de tareas**: Dadas *n* personas *y n* tareas*,* queremos asignar a cada persona una tarea. El coste de asignar a la persona *i* la tarea *j* viene determinado por la posición [*i,j*] de una matriz dada *(coste).* El algoritmo debe asignar una tarea a cada persona minimizando el coste de la asignación total
  - **El viajante de comercio**: Se parte de un grafo con *n* ciudades. El objetivo es encontrar una ruta que, comenzando y terminando en una ciudad concreta, pase una sola vez por cada una de las ciudades y minimice la distancia recorrida por el viajante. La distancia entre cada ciudad viene dada por la matriz D: [NxN], donde D[x, y] representa la distancia que hay entre la ciudad X y la ciudad Y

Índice

Branch and Bound

- **Branch and Bound** (Ramificación y Poda) es una variante del esquema de backtracking
- Al igual que backtracking, se basa en el recorrido del árbol de expansión en busca de soluciones
- Se aplica sobre todo a problemas de optimización (búsqueda de la mejor solución a un problema)
- La principal diferencia con backtracking es que la generación de nodos se puede realizar aplicando distintas **estrategias de ramificación**
  - Además de recorrido en profundidad del árbol de expansión, también se puede hacer recorrido en anchura o buscando el nodo más prometedor
- Además se utilizan **estrategias de poda**: cotas que permiten podar ramas que no conducen a una solución optima (se evita ramificar nodos)

Estrategias de ramificación

- ✓ En el esquema de backtracking, el recorrido del árbol de expansión es siempre en profundidad
- ✓ En B&B, la generación de los nodos del árbol de expansión puede seguir varias estrategias:
  - En profundidad (LIFO)
  - En anchura (FIFO)
  - Seleccionar el nodo más prometedor
- ✓ El objetivo es utilizar la estrategia que permita encontrar la solución más rápidamente

Estrategias de ramificación

- ✓ Para determinar qué nodo va a ser expandido, dependiendo de la estrategia de ramificación, necesitamos una estructura capaz de almacenar aquellos nodos pendientes de ser expandidos
- ✓ Para hacer el recorrido se utiliza una lista de nodos llamada Lista de Nodos Vivos (LNV)
  - Un nodo vivo del árbol el que tiene posibilidades de ser ramificado, es decir, el que ha sido creado y no ha sido explorado ni podado todavía
  - **La lista de nodos vivos contiene nodos pendientes de tratar por el algoritmo**

Estrategias de ramificación

✓ La pasos a seguir para la ramificación son los siguientes:

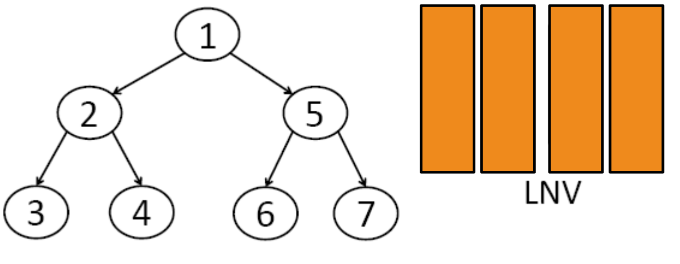
1. Sacar un elemento N de la lista de nodos vivos (LNV)
2. Generar todos los descendientes de N
3. Si no se podan y no son solucion, se introducen en la LNV

✓ El recorrido del árbol depende de cómo se maneje la LNV

- Recorrido en profundidad → La lista se trata como una **pila**
- Recorrido en anchura → La lista se trata como una **cola**
- Estrategia de mínimo coste → Se utiliza una **cola con prioridades** para almacenar nodos ordenados por su coste

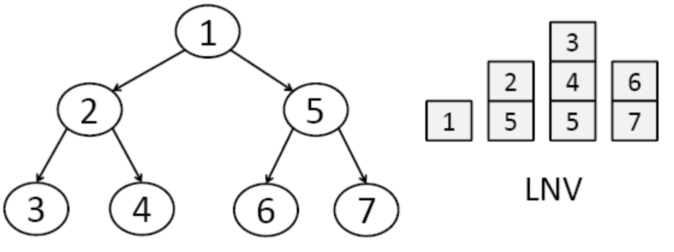
Estrategias de ramificación

✓ Recorrido en profundidad (LIFO, pila)



Estrategias de ramificación

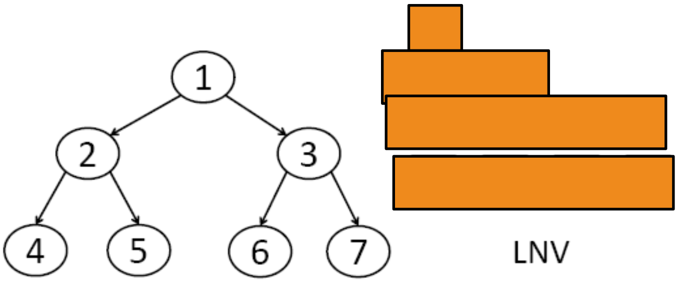
✓ Recorrido en profundidad (LIFO, pila)



UCAM | Universidad Católica San Antonio

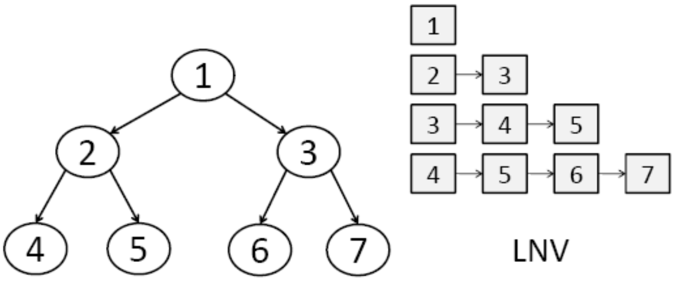
Estrategias de ramificación

✓ Recorrido en anchura (FIFO, cola)



Estrategias de ramificación

✓ Recorrido en anchura (FIFO, cola)



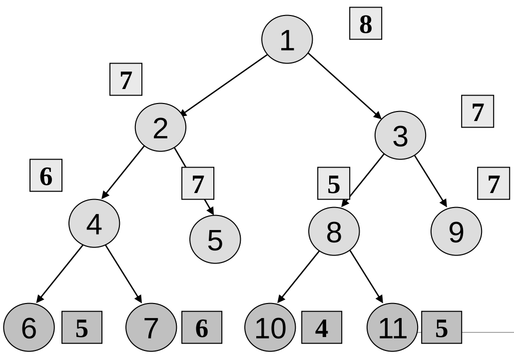
Estrategias de ramificación

✓ Recorrido seleccionando el nodo más prometedor

- Entre todos los nodos de la lista de nodos vivos, elegir el que tenga mayor beneficio (o menor coste) para explorar a continuación.
- En caso de empate (de beneficio o coste estimado) deshacerlo usando un criterio **FIFO** o **LIFO**:
  - **Estrategia Coste-FIFO:** Seleccionar de la LNV el que tenga mayor beneficio y en caso de empate escoger el primero que se introdujo (de los que empatan).
  - **Estrategia Coste-LIFO:** Seleccionar de la LNV el que tenga mayor beneficio y en caso de empate escoger el último que se introdujo (de los que empatan).

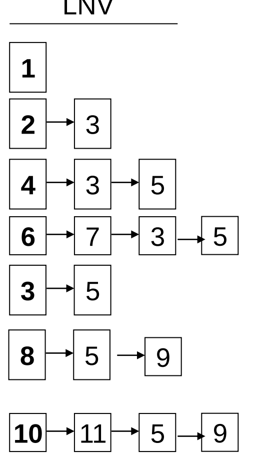
Estrategias de ramificación

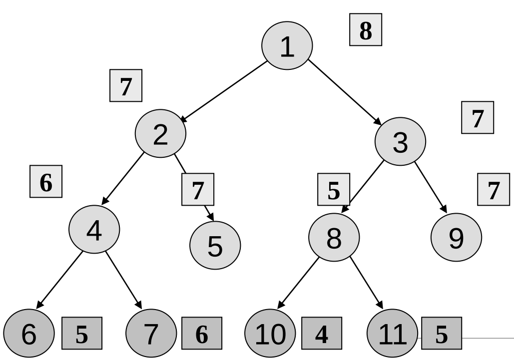
- ✓ Recorrido seleccionando el nodo más prometedor
  - Suponiendo un problema de minimización y una estrategia Coste-FIFO, ¿cómo se insertarían los nodos en la lista?.



Estrategias de ramificación

- ✓ Recorrido seleccionando el nodo más prometedor
  - Solución





Estrategias de poda

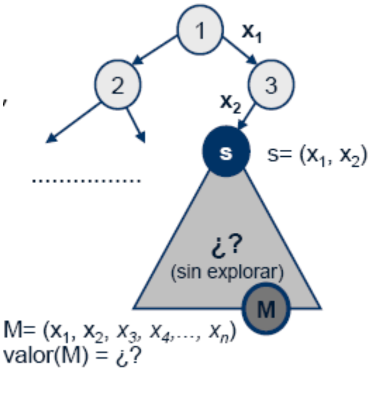
- ✓ Para cada nodo establecemos una estimación de la mejor solución posible a partir de él (**cota**)
- ✓ Las cotas determinan cuando se puede realizar una poda del árbol: si el valor de la cota es peor que la mejor solución obtenida hasta ese momento no se exploran sus hijos
- ✓ Para cada nodo *i* podemos tener:
  - Cota Inferior(*i*) de la mejor solución alcanzable a partir del nodo *i*
  - Cota Superior(*i*) de la mejor solución alcanzable a partir del nodo *i*
  - Estimación del beneficio (o coste) que se puede encontrar a partir de ese nodo. Se puede obtener a partir de las cotas: usar la media o una de ellas.
- ✓ Si M(*i*) es la mejor solución alcanzable a partir del nodo *i*, se debe verificar lo siguiente:

CotaInferior(i) <= M(i) <= CotaSuperior(i)

Estrategias de poda

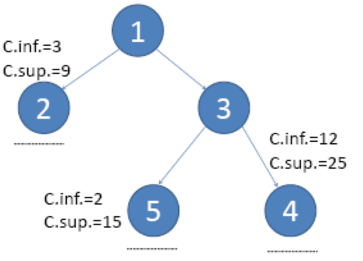
Antes de explorar **s,** se acota el valor de la mejor solución **M** alcanzable desde **s.**

CI(s) ≤ valor(M) ≤ CS(s)



Estrategias de poda

- **Ejemplo**: Supongamos que tenemos un problema de maximización. Tenemos varios nodos y para cada uno de ellos se calcula la cota superior, y la cota inferior



- Si nos encontramos en el nodo 2, ¿merece la pena expandir dicho nodo? ¿Y el nodo 5?
- Si se tratase de un problema de minimización, ¿qué respuestas se darían a las preguntas anteriores? ¿Se debería expandir el nodo 4?

Estrategias de poda

- ¿Cómo realizar la poda? (caso maximización)
  - Para podar un nodo *i* se tiene que verificar la siguiente condición:

    CotaSuperior (i) $\leq$ Valor (s)

  donde *Valor(s)* es el valor de la mejor solución encontrada hasta el momento

- Para que esta poda pueda funcionar, es necesario encontrar una primera solución del problema (o asignar el peor valor posible a *Valor(s)*).
- Además, para podar un nodo del árbol solamente podemos utilizar las soluciones encontradas hasta el momento.
  - Cuanto más tarde encontremos la mejor solución, menos efecto tendrá la poda

Estrategias de poda

- ¿Cómo realizar la poda? (caso maximización)
  - Se puede mejorar la poda anterior utilizando las dos cotas
  - Supongamos que ya hemos generado un nodo *j* para el que hemos calculado su cota inferior
  - En este caso, podemos podar el nodo *i* si se cumple que

CotaSuperior (i) < CotaInferior (j)

- ¿Qué quiere decir esto? → la mejor solución a partir del nodo *i* va a ser peor que la cota inferior de la mejor solución que provenga del nodo *j*
- Combinando esta idea con la poda anterior, se puede podar el nodo *i* si se cumple:

CotaSuperior (i) < *Cota*

siendo *Cota* = max(CotaInferior (j), Valor (s))

UCAM | UNIVERSIDAD CATÓLICA SAN ANTONIO

Estrategias de poda

- ¿Cómo realizar la poda? (caso maximización)
  - Para poder utilizar cotas inferior y superior se debe cumplir la siguiente propiedad: “Todo nodo del árbol de expansión que no sea hoja debe tener al menos un descendiente que sea solución del problema”
  - Esto es debido a que la poda se produce en función de cotas de otros nodos, en lugar de soluciones que han sido encontradas
  - Podría producirse el caso siguiente:
    - Encontramos un nodo *j* muy prometedor, con un valor de CotaInferior (j) muy alto que provoca la poda de muchas ramas del árbol de expansión.
    - Sin embargo, *j* no tiene ningún descendiente que sea solución del problema, por lo que se han podado ramas que si podrían llevar a soluciones del problema

- **¿Cómo se realizaría la poda para el caso de minimización?**
  - Se deja al alumno que plantee el esquema de la poda

Esquema de Branch & Bound

- ✓ En un algoritmo de ramificación y poda se realizan tres etapas:
  1. <u>Etapa de Selección</u>: Se encarga de extraer un nodo de la LNV. La forma de escogerlo depende de la estrategia de ramificación
  2. <u>Etapa de Ramificación</u>: Se generan los posibles hijos del nodo seleccionado en la etapa anterior
  3. <u>Etapa de Poda</u>: se estudian los nodos generados en la etapa de ramificación. Solo aquellos que pasan cierto filtro se introducen en la LNV. El resto de nodos son podados

- ✓ Se repiten estas tres etapas mientras la LNV tiene nodos pendientes de procesar.
- ✓ <u>La representación de la solución actual se incluye ahora en cada nodo</u>

Esquema de Branch & Bound

- ✓ Esquema iterativo Branch & Bound (<u>maximización</u>)

<u>Inicialización</u>

```
func ByBMax(): nodo {            // Devuelve el nodo solución del árbol
    sol, x: nodo;
    cota: int;                   // o real (cambiar tipo según el dominio del problema)
    hijos: array [1..n] of nodo;

    crear(nodoRaiz);             // Nodo con la tupla de solución vacía
    crear(LNV);          // Creación de la Lista de Nodos Vivos (pila/cola/cola con prioridad)
    introducir(LNV, nodoRaiz);
    actualizar(cota);            // Peor valor posible
    sol:= ∅;                     //  No hay solución inicialmente
    …
```

✓ Esquema iterativo Branch & Bound (maximización)

Selección, ramificación y poda

```
proc ByBMax(): nodo {
   …
  mientras not(vacia(LNV)) hacer
        x:= siguiente(LNV);
        si cotaSup(x) >= cota entonces            // Si no se cumple, se poda x
             hijos:=  calcularHijos(x);
             desde i = 1 hasta ultimo hijo de x hacer
                  si (esSolucion(hijos[i])) AND (Valor(hijos[i]) > Valor(sol)) entonces
                       sol:=   hijos[i];
                       cota:= max{cota, Valor(hijos[i])};
                  sino si  (not esSolucion(hijos[i])) AND (cotaSup(hijos[i]) >= cota)
                       introducir (LNV, hijos[i ]);
                       cota:= max{cota, cotaInf(hijos[i]);
                  fin_si
             fin_desde
        fin_si
fin_mientras
devuelve sol;
```


✓ Esquema iterativo Branch & Bound (minimización)

Selección, ramificación y poda (Inicialización igual que en el caso de maximización)

```
proc ByBMax(): nodo {
   …
  mientras not(vacia(LNV)) hacer
        x:= siguiente(LNV);
        si cotaInf(x) <= cota entonces            // Si no se cumple, se poda x
             hijos:=  calcularHijos(x);
             desde i = 1 hasta ultimo hijo de x hacer
                  si (esSolucion(hijos[i])) AND (Valor(hijos[i]) < Valor(sol)) entonces
                       sol:=   hijos[i];
                       cota:= min{cota, Valor(hijos[i])};
                  sino si  (not esSolucion(hijos[i])) AND (cotaInf(hijos[i]) <= cota)
                       introducir (LNV, hijos[i ]);
                       cota:= min{cota, cotaSup(hijos[i]);
                  fin_si
             fin_desde
        fin_si
fin_mientras
devuelve sol;
```

Programación Backtracking

- ✓ **actualizar(cota)**
  - Se debe asignar el valor inicial a la cota:
    - Maximización: Obtener la cota inferior del nodo raíz o si se desconoce poner el valor/coste más bajo que se puede obtener (-∞)
    - Maximización: Obtener la cota superior del nodo raíz o si se desconoce poner el valor/coste más alto que se puede obtener (∞)
- ✓ **siguiente(LNV)**
  - Devuelve el siguiente nodo a evaluar y **lo elimina** de la lista. Dependiendo del tipo de recorrido realizado, devolverá el tope de la pila o la cabecera de la lista (priorizada)
- ✓ **calcularhijos(x)**
  - Devuelve un array con los nodos descendientes del nodo *x*. Estos nodos hijos deben contener la siguiente información:
    - Cota superior, cota inferior y valor estimado
    - La tupla de la solución parcial alcanzada hasta ese nodo
- ✓ **valor(nodo)**
  - Devuelve el valor estimado del nodo

Tiempo de ejecución B&B

- ✓ El tiempo de ejecución de un algoritmo B&B depende de:
  - El numero de nodos recorridos → depende de la efectividad de la poda!!
  - El tiempo empleado en cada nodo → tiempo necesario para hacer las estimaciones de coste y gestionar la lista de nodos vivos en función de la estrategia de ramificación.

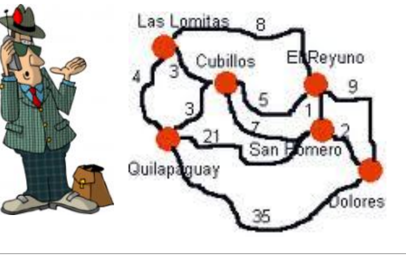
- ✓ En el **peor caso,** el tiempo de un algoritmo B&B será igual al de un algoritmo backtracking (o peor incluso, si tenemos en cuenta el tiempo que requiere la LNV).
- ✓ En el **caso promedio,** se suelen obtener mejoras con respecto a backtracking.

Tiempo de ejecución B&B

- ✓ El tiempo de ejecución de un algoritmo B&B depende de:
  - El numero de nodos recorridos → depende de la efectividad de la poda!!
  - El tiempo empleado en cada nodo → tiempo necesario para hacer las estimaciones de coste y gestionar la lista de nodos vivos en función de la estrategia de ramificación.
- ✓ En el **peor caso,** el tiempo de un algoritmo B&B será igual al de un algoritmo backtracking (o peor incluso, si tenemos en cuenta el tiempo que requiere la LNV).
- ✓ En el **caso promedio,** se suelen obtener mejoras con respecto a backtracking.
- ✓ El "truco" está en buscar un equilibrio en la precisión de las cotas calculadas
  - Muy precisas → Mayor poda, se recorren menos nodos, pero aumenta el tiempo en realizar estimaciones
  - Poco precisas → Menor poda, se recorren más nodos, pero disminuye el tiempo de las estimaciones

Ejemplo B&B. El viajante de comercio

- ✓ **Especificación**: Encontrar un recorrido de longitud mínima para una persona que tiene que visitar varias ciudades y volver al punto de partida, conocida la distancia existente entre cada dos ciudades.
- ✓ Se trata de encontrar un circuito de longitud mínima que comience y termine en el mismo vértice y pase exactamente una vez por cada uno de los vértices restantes → *Ciclo hamiltoniano de longitud mínima*



Ejemplo B&B. El viajante de comercio

Representación de la información

- ✓ Las distancias entre las ciudades se pueden representar mediante la matriz de adyacencia del grafo que representa el mapa.

	Las Lomitas	Cubillos	El Reyuno	San Romero	Quilapa…	Dolores
Las Lomitas	0	3	8	∞	4	∞
Cubillos	3	0	5	7	3	∞
El Reyuno	8	5	0	1	∞	9
San Romero	∞	7	1	0	21	2
Quilapa…	4	3	∞	21	0	35
Dolores	∞	∞	9	2	35	0

Ejemplo B&B. El viajante de comercio

Representación de la solución

- ✓ La solución del problema será un vector N+1 (donde N es el número de ciudades) que indica el orden en el que se deben visitar los vértices del grafo de ciudades.
  - Cada elemento del vector debe contener un número entre 1 y N (número de ciudades)
  - Inicialmente el vector solución contiene un solo elemento que representa el vértice de origen (el 1).
  - El último valor del vector también es el vértice de origen. Por tanto, el vector solución será → sol:[1, x_2, x_3, …, x_n, 1]
  - Como se deben considerar los ciclos hamiltonianos, en la solución no se puede repetir ningún vértice (excepto el vértice origen).

- ✓ Las posibles soluciones del problema estarán sólo en los nodos hoja, correspondientes a la etapa N+1. El resto de nodos del árbol contienen soluciones parciales.

Ejemplo B&B. El viajante de comercio

Estimación de la cota

- ✓ Como es un problema de minimización, debemos calcular una cota inferior para cada nodo del árbol de expansión.
- ✓ Podemos utilizar como cota inferior de un nodo la siguiente información:
  i. La suma de las longitudes de las aristas mínimas que salen de los vértices pendientes de incluir en la solución,
  ii. más la longitud de la arista mínima que sale del último vértice visitado
  iii. más la longitud del camino que forma la solución parcial obtenida hasta el momento
- ✓ Como se va a calcular el camino de longitud mínima, no es necesario calcular la cota superior de cada nodo.
- ✓ Puede haber ramas del árbol que no lleven a ninguna solución: la cota solamente se actualiza cuando se encuentra una solución.

Ejemplo B&B. El viajante de comercio

Estimación de la cota (Ejemplo)

- ✓ Dada la siguiente matriz de adyacencia:

0	14	4	10	20
14	0	7	8	7
4	5	0	7	16
11	7	9	0	2
18	7	17	4	0

  - El camino de longitud mínima es [1, 4, 5, 2, 3, 1] → Coste = 30
  - Supongamos que ya tenemos una solución parcial [1, 4]
    - La longitud de la solución parcial es 10.
    - Las aristas de longitud mínima que salen de los vértices no visitados son:
      2: min(14,7,7)= 7 | 3: min(4,5,16)= 4 | 5: min(18,7,17)= 7
    - La arista de longitud mínima que salen del último vértice es:
      4: min(7,9,2)= 2
    - No hemos considerado las aristas dirigidas al vértice 4, pues no es posible volver a visitar este vértice, ni la arista del 4 al 1
  - Por tanto, la cota inferior de la solución parcial [1, 4] es 10 + 7 + 4 + 2 + 7 = 30

Ejemplo B&B. El viajante de comercio

Estructura del nodo

- ✓ Cada nodo del árbol debe contener toda la información necesaria para realizar la ramificación y la poda, así como la solución parcial obtenida hasta el momento.
- ✓ La estructura de cada nodo puede ser:

Solución actual
Nivel actual
longitud solución actual
Cota inferior

donde “Solución actual “contiene el vector que representa la solución parcial hasta el momento

UCAM | UNIVERSIDAD CATÓLICA SAN ANTONIO

Ejemplo B&B. El viajante de comercio

- ✓ Esquema iterativo Branch & Bound

Inicialización

```
proc ByB_Viajante(D: int[1..N, 1..N] , salida mejorSol: int[1..N], salida Cota: int) {
                        // D es la matriz de distancia de ciudades
   x: nodo;
   hijos: array [1..n] of nodo;
   numHjios: int;

   x:= crearNodoRaiz(D);            // x es nodo raíz
   crearListaPrioridad(LNV);        // Lista de Nodos Vivos como cola con prioridad
   introducir(LNV, x);
   cota = Suma de todos los valores de D;      // Peor valor posible
   …
```

✓ Esquema iterativo Branch & Bound (viajante)

Selección, ramificación y poda

```
…
mientras not(vacia(LNV)) hacer
      x:= siguiente(LNV);
      si x.cotaInf < cota entonces          // Si no se cumple, se poda x
            numHijos:=  calcularHijos(x, D, hjios);
            desde i = 1 hasta numHijos hacer
                  si (hijos[i].nivel == N) AND (hijos[i].cotaInf) < cota) entonces
                        longCamino:=   hijos[i].longitud + D[hijos[i].sol[N],1];
                        si longCamino < cota entonces
                              mejorSol:= hijos[i].sol;
                              cota:= longCamino;
                        fin_si
                  sino si  (hijos[i].nivel < N) AND (hijos[i].contaInf < cota) entonces
                        introducir (LNV, hijos[i]);  // Introducir ordenados por hijos[i].cotaInf
                  fin_si
            fin_desde
      fin_si
fin_mientras
```


✓ Funciones auxiliares

```
func crearnodoRaiz(D: int[1..N, 1..N]): nodo {
   raiz: nodo; // asignar memoria a raiz;
    raiz.sol[1]:=  1;
    raiz.nivel:=   1
    raiz.longitud :=  0;
    raiz.cotaInf:= calcularCotaInf(raiz,D);
   devueve raiz;
}

func  visitado(ciudad: int, sol: int[1..N], nivel: int): boolean {
    desde i: 1 hasta nivel hacer
          si ciudad = sol[i] entonces devolver true;
   fin_desde
   devolver false;
}
```

✓ Funciones auxiliares

```
func  calcularHijos(padre: nodo, D: int[1..N, 1..N], hijos: int[1..N]):int {
    contadorHijos: int;
    contadorHijos:= 1;
    desde i: 1 hasta N hacer
            si  not(visitado(i, padre.sol, padre.nivel)) AND (D[padre.sol[padre.etapa],i] ≠ 0 entonces
                    hijos[contadorHijos].sol: =  copiar(padre.sol);
                    hijos[contadorHijos].sol[padre.etapa+1]: = i;
                    hijos[contadorHijos].etapa:=  padre.etapa +1;
                    hijos[icontadorHijos.longitud:=   padre.longitud + D[padre.sol[padre.etapa],i];
                    hijos[contadorHijos].cotaInf:=   calcularCotaInf(hijos[contadorHijos],D);
                    contadorHijos++;
            fin_si
    fin_desde
    devolver contadorHijos;
```


UCAM | UNIVERSIDAD CATÓLICA SAN ANTONIO

✓ Funciones auxiliares

```
fun calcularCotaInf(n: nodo,D: int[1..N,1..N]): int {
    cotaInf, minFila: int;
    cotaInf:=   n.longitud;
    desde i: 1 hasta N hacer
            si not(visitado(i, n.sol, (n.etapa)-1) entonces
                    minFila:= ∞;
                    desde j :1 hasta N hacer
                            si (i≠j) AND (not(visitado(j, n.sol, n.etapa)) OR (j=1)) entonces
                                    minFila:=  min(minFila, D[i,j]);
                            fin_si
                    fin_desde
            cotaInf:=  cotaInf+ minFila;
            fin_si
    fin_desde
    devolver cotaInf;
}
```

Backtracking Vs. Branch & Bound

- ✓ En backtracking, tan pronto como se genera un nuevo hijo del nodo en curso, dicho hijo pasa a ser el nodo en curso.
- ✓ En B&B, se generan todos los hijos del nodo en curso antes de que cualquier otro nodo vivo pase a ser el nuevo nodo en curso (**no se realiza un recorrido en** profundidad por defecto).

- ✓ **En consecuencia:**
  - En backtracking, los únicos nodos vivos son los que está en el camino de la raíz al nodo en curso.
  - En B&B puede haber mas nodos vivos que en backtracking, que se almacenan en una lista de nodos vivos.

UCAM | UNIVERSIDAD CATÓLICA SAN ANTONIO

Backtracking Vs. Branch & Bound

- ✓ En backtracking, el test de comprobación realizado por la funciones de evaluación nos indica únicamente si un nodo concreto nos puede llevar a una solución o no.
- ✓ En B&B, sin embargo, se acota el valor de la solución a la que nos puede conducir un nodo concreto, de forma que esta acotación nos permite:
  - Podar el árbol (si sabemos que no nos va a llevar a una solución mejor de la que ya tenemos), y
  - Establecer el orden de ramificación (de modo que comenzaremos explorando las ramas mas prometedores del árbol).

Resumen de cotas B&B

Estrategia de poda en Branch & Bound

	Problema de maximización	Problema de minimización
Valor	Beneficio	Coste
Podar si…	**CL < CG**	**CL > CG**
Cota local	CL ≥ Óptimo local	CL ≤ Óptimo local
	Interpretación: No alcanzaremos nada mejor al expandir el nodo.	
Cota global	CG ≤ Óptimo global	CG ≥ Óptimo global
	Interpretación: La solución óptima nunca será peor que esta cota.	

Bibliografía

- ✓ Yolanda García Ruíz, Jesus Correas**. Esquemas algorítmicos**. Departamento de Sistemas Informáticos y Computación. Universidad Complutense de Madrid

- ✓ Rosa Guerequeta y Antonio Vallecillo. **Técnicas de Diseño de Algoritmos**. Segunda Edición. Servicio de Publicaciones de la Universidad de Málaga, 2000.