Universidad Simón Bolívar
Teoría de Algoritmos
Tarea II

Fabiola Di Bartolo
Carnet: 09-87324

11 de noviembre de 2009

1. *a*) Brassard - Ejercicio 8.30. Sean u y v dos cadenas de caracteres. Deseamos transformar u en v con el menor número posible de operaciones de los tres tipos siguientes: borrar un caracter, añadir un caracter o modificar un caracter. Por ejemplo podemos transformar *abbac* en tres etapas:

   abbac → abac (borrar b)
   → ababc (añadir b)
   → abcbc (transformar a en c)

   ⋆ Demostrar que esta optimización no es óptima.

   Si fuese óptima no existiría una solución que transforme u en v en una cantidad menor de operaciones, y existe la siguiente solución, la cual no emplea operaciones innecesarias de eliminación e inserción:

   abbac → abcac (transformar b en c)
   → abcbc (transformar a en c)

   ⋆ Escribir un algoritmo de programación dinámica que busque el número mínimo de operaciones necesarias para transformar u en v y que nos diga cuales son esas operaciones. En función de las longitudes u y v, ¿cuánto tiempo requiere este algoritmo?

   Para transformar la cadenas de caracteres u (de tamaño $n1$) en v (de tamaño $n2$), se utilizará una matriz m de dimensiones $(n1+1)$ x $(n2+1)$ donde cada posición $m[i, j]$ corresponde al número mínimo de operaciones necesarias para transformar $u[1..i]$ en $v[1..j]$. El algoritmo trabajaría así:

   - ⋆ Si tenemos que $n1 = 0$, entonces el mínimo número de operaciones para transformar u en v es $n2$, y sólo serían inserciones.
   - ⋆ Si tenemos que $n2 = 0$, entonces el mínimo número de

operaciones para transformar u en v es $n1$, y sólo serían eliminaciones.

- ⋆ Si tenemos que $n1 \neq 0$ y $n2 \neq 0$, entonces al menos u y v tienen un caracter $c1$ y $c2$ respectivamente, llamemos $s1$ y $s2$ las cadenas que quedan de restarle el caracter $c1$ y $c2$ a u y v y $d(x, y)$ al costo de transformar x en y.
  - ▷ Si $c1$ es igual a $c2$, entonces no hay que hacer ninguna operación $d(c1, c2) = 0$ y el costo total es $d(s1, s2)$.
  - ▷ Sino, entonces $c1$ puede ser cambiado en $c2$ generando el costo de transformar $s1$ en $s2$ más uno por el caracter cambiado, $d(s1, s2) + 1$.
  - ▷ Otra posibilidad puede ser que haya que eliminar $c1$ y transformar $s1$ en $s2 + c2$, $d(s1, s2 + c2) + 1$.
  - ▷ La ultima posibilidad es que haya que agregar un elemento, esto es transformar $s1 + c1$ en $s2$ e insertar $c2$, $d(s1 + c1, s2) + 1$. De esas tres alternativas, se toma la menos costosa.
- ⋆ Para reconstruir la secuencia de operaciones, lo que se hace es iterar partiendo de la última operación $n1, n2$ y buscando una a una las operaciones anteriores dependiendo de la operación realizada (0 = nada, 1 = eliminacion, 2 = insercion, 3 = modificacion) se va almacenando cada paso, en una cola (`paso`), donde cada elemento es de la forma (o, c), o es la operación a realizar y c es el caracter a agregar, eliminar o que indica el cambio a realizar en u.

```
proc transformar(u[1..n1], v[1..n2])
  /* m es la matriz con n1+1 filas y n2+1 columnas */
  m[0,0] := 0
  for i := 1 to n1 do
      m[i,0] := i
      op[i,0] := 1/* eliminación */
  od
  for j := 1 to n2 do
      m[j,0] := j
      op[i,j] := 2/* inserción */
  od
```

```
for j := 1 to n2 do
    for i := 1 to n1 do
        if u[i] = v[j] then
                            m[i, j] := m[i − 1, j − 1]
                            op[i, j] := 0
                        else
                            m[i, j] := min(
                            m[i − 1, j] + 1, /* 1=eliminacion */
                            m[i, j − 1] + 1, /* 2=insercion */
                            m[i − 1, j − 1] + 1/* 3=modificacion */
                            )
                            op[i, j] := op/* op: 1,2,3 */
        fi
    od
od
/* Reconstruye secuencia de operaciones */
j := n2
i := n1
while i >= 0 ∧ j >= 0 do
      if op[i, j] = 0 then /* nada */
                            o = operacion(0,'')
                            pasos.insert(o)
                            i := i − 1
                            j := j − 1
      fi
      if op[i, j] = 1 then /* eliminacion */
                            o = operacion(op[i, j], v[i])
                            pasos.insert(o)
                            i := i − 1
                            continue
      fi
      if op[i, j] = 2 then /* insercion */
                            o = operacion(op[i, j], u[j])
                            pasos.insert(o)
                            j := j − 1
                            continue
      fi
      if op[i, j] = 3 then /* modificacion */
```

$o = operacion(op[i, j], u[j])$
$pasos.insert(o)$
$i := i - 1$
$j := j - 1$
fi
od
return $m[n1, n2]$

.

⋆ El orden de algoritmo en tiempo es $O(n1 * n2)$ para todo $n1$ y $n2$ correspondientes a las longitudes de la cadena a transformar y a la que se quiere llegar. En espacio se necesita $O(n1 * n2)$ para almacenar la matriz que indica la cantidad de operaciones.

b) Brassard - Ejercicio 8.31. Se dispone de n objetos que es necesario ordenar empleando las relaciones $' <'$ y $' ='$. Dar un algoritmo de programación dinámica que pueda calcular como función de n el numero de ordenaciones posibles. El algoritmo debe necesitar un tiempo que este en $O(n^2)$ y un espacio que esté en $O(n)$.

Las formas de ordenar n objetos empleando las relaciones $' <'$ y $' ='$, es equivalente a la suma del número posible de permutaciones que pueden tener los objetos considerando dentro de ese conjunto desde 1,2, hasta n objetos indistiguibles. Es decir, el número de las distintas formas de ordenar n elementos empleando las relaciones $' <'$ y $' ='$, se puede calcular en términos del factorial descendiente, quedando una suma del número de formas en que se puede ordenar cada patrón distinto (desde 1 hasta n elementos iguales), $Ord_n = 1 + n^{\underline{2}} + n^{\underline{3}} + ... + n^{\underline{n}} = \sum_{k=2}^{n} n^{\underline{k}} + 1$. Donde $n^{\underline{k}}$ es la forma de ordenar n objetos suponiendo que k son distintos, es decir, $n - k$ iguales y se le suma 1 debido al caso en el que todos los elementos sean iguales.

Creando una recurrencia para calcular Ord_n en función del anterior Ord_{n-1} queda: $Ord_n = n(Ord_{n-1} - 1) + n(n - 1) + 1$. Teniendo

como caso base $Ord_o = 0$, si hay 0 elementos, no hay ningún orden y $Ord_1 = 1$, si existe un sólo elemento, existe un sólo orden.

El cual se puede calcular con en el siguiente algoritmo, donde cada iteración se crea a partir del número de ordenes distintos de la iteración anterior. La solución para n objetos quedaría almacenada en `ord[n]`

```
proc calcularOrdenes(n)
  /* ord es un arreglo que contiene el numero de ordenes de 0 a n objetos */
  ord[0] = 0
  ord[1] = 1
  for k := 2 to n do
      ord[k] := k * (ord[k - 1] - 1) + k * (k - 1) + 1
  od
.
```

Como se puede observar, el orden del algoritmo en tiempo y espacio es $O(n)$ para todo n.

2. Se quiere que resuelva con backtracking el siguiente problema: Dado un conjunto A de n elementos, y un conjunto F con m subconjuntos de A, determinar, si existe, un subconjunto de F que sea una partición de A.

   a) Describir claramente el grafo implícito (vértices y sucesores) sobre el cual realizará el DFS. Defina criterios razonables de poda. (Note que F lo puede ver como una matriz mxn de ceros y unos, donde cada fila representa un conjunto en F)

   Grafo Implicito

   - ⋆ Vértices: un vértice es un conjunto B de subconjuntos de A.
   - ⋆ Sucesores: los sucesores de B son todos los conjuntos $C = B \cup \{x\}$ tal que si $D \in C \Rightarrow D \in B \vee D \cap B_i = \phi$, donde B_i es cualquier elemento de B. Es decir, los subconjuntos restantes a ser escogidos para formar la solución, no pueden

tener ningún elemento en común con los agregados en los pasos anteriores.

- ⋆ El algoritmo terminará al encontrar la primera solución, ya que lo que se busca es determinar si existe o no un subconjunto de F que sea una partición de A. Para hallar la solución, descartará aquellas ramas cuyas hojas no sean una partición de A.

b) Corra paso a paso su algoritmo con el siguiente ejemplo:
$A = \{a, b, c, d, e, f, g\}$,
$F = \{\{c, e, f\}, \{a, d, g\}, \{b, c, f\}, \{a, d\}, \{b, a, g\}, \{b, g\}\}$

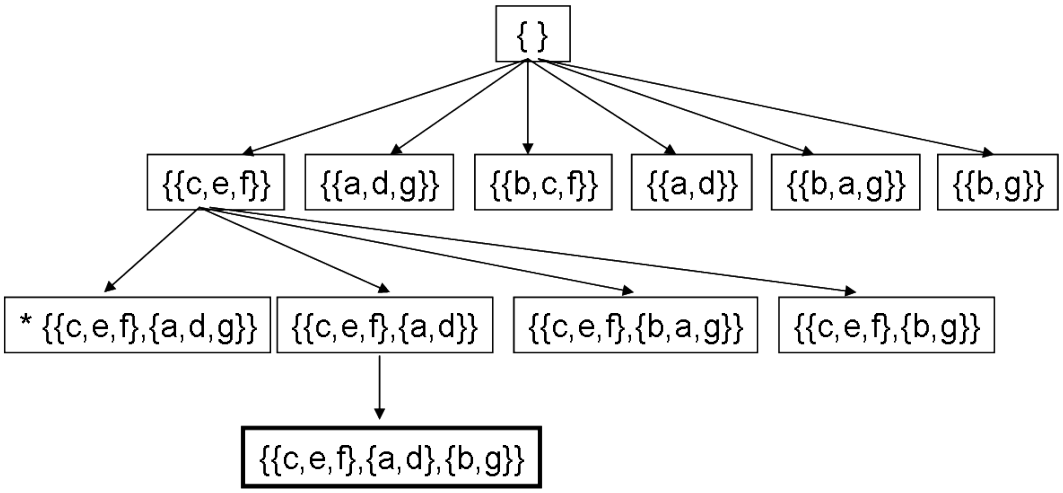


Figura 1: Arbol implicíto para el problema de hallar una partición de A

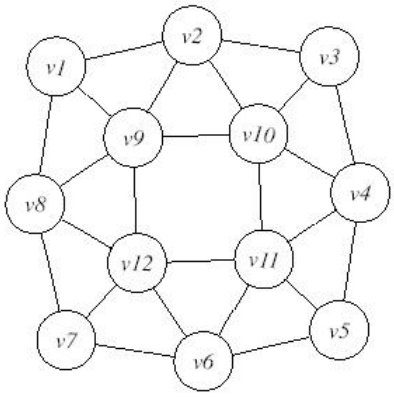
Ejecución del algoritmo

La corrida del algoritmo puede ser observada en la Figura 1. Inicialmente expande la raíz con todos los sucesores, los cuales son los conjuntos formados por cada uno de los elementos de F. Toma el primero $\{\{c, e, f\}\}$, lo visita y lo expande a sus sucesores, $\{\{c, e, f\}, \{a, d, g\}\}$, $\{\{c, e, f\}, \{a, d\}\}$, $\{\{c, e, f\}, \{b, a, g\}\}$

y $\{\{c, e, f\}, \{b, g\}\}$. Es importante resaltar, que no se toma como sucesor $\{\{c, e, f\}, \{b, c, f\}\}$ porque viola la definición de partición al dejar de ser vacía la intersección de $\{c, e, f\}$ con $\{b, c, f\}$. Tomando el primer sucesor, $\{\{c, e, f\}, \{a, d, g\}\}$, lo visita, sin embargo, ya no se puede expandir más esta rama y como el vértice no es una partición de A, se descarta (indicado en la figura con un $*$). Entonces, toma el siguiente sucesor, $\{\{c, e, f\}, \{a, d\}\}$ lo visita y lo expande, generando el vértice $\{\{c, e, f\}, \{a, d\}, \{b, g\}\}$, el cual si es una partición de A, por lo tanto, al existir una, termina la ejecución.

3. Aplicar Branch-and-Bound, paso a paso, para determinar un conjunto independiente de vértices de cardinalidad máxima en el grafo siguiente:

Grafo implícito: un vértice será un conjunto X independiente de vértices. Los sucesores de X son todos los conjuntos $X \cup \{x\}$ que son independientes. Emplee un algoritmo greedy para hallar una solución inicial que permita junto a la función de cota (bound), podar el árbol. Para la cota (bound) que permitirá podar el árbol, use la siguiente: estando en un vértice X, una cota superior del máximo independiente que contiene a X es $\mid X \mid + \mid \{Y : Y \quad \textit{es sucesor de} \quad X\} \mid$ ¿por qué?.



Algoritmo *Greedy*

Se ordenan los nodos de menor a mayor $(v1, v2, ..., v12)$, y se va construyendo iterativamente el conjunto independiente de vértices C_i, tomando un vértice vi $(vi \in V)$, en el orden dado, tal que no tenga alguna arista $e \in E$ para el grafo $G = (V, E)$ cuyo extremo sea un vértice vj $(vj \in C_i)$. En el primer paso toma $v1$ y lo agrega a C_i, luego $v2$, pero $v2$ ya tiene una arista que lo conecta con $v1$, asi que no hace nada con él y pasa al siguiente, $v3$, el cuál lo agrega también al conjunto. Esto se realiza sucesivamente hasta llegar al nodo $v7$ el cual es el último a ser incluido en el conjunto C_i porque todos los posteriores a el poseen aristas que los conectan con los vértices en C_i. Como resultado devuelve el conjunto $C_i = \{v1, v3, v5, v7\}$, el cual tiene cardinalidad 4 y se tomará como solución inicial para ejecutar el algoritmo *Branch and Bound*.

Ejecución *Branch and Bound*

Parte de la corrida del algoritmo puede ser observada en la Figura 2. Inicialmente cada vértice del grafo G es un vértice del grafo implícito, sucesor de la raíz. Se visita primero $\{v1\}$, se expande, se le calcula la cota (9), la cual sigue siendo mejor que la solución inicial (4). Se visita el primer sucesor $\{v1, v3\}$, se expande y se calcula su cota (7), como es mayor que la solución inicial, continua expandiendo a los sucesores, se visita el primero $\{v1, v3, v5\}$ (cota 5), se expande proporcionando dos posibles soluciones $\{v1, v3, v5, v7\}$ y $\{v1, v3, v5, v12\}$, ninguna es mejor que la solución inicial, así que no se sustituye el valor de la solución por la cardinalidad de estos conjuntos. Luego se devuelve un nivel, visita el nodo $\{v1, v3, v6\}$ pero al tener una cota menor a 4, se descarta, se denota con un asterisco $(*)$ en la figura. Continúa con cada uno de los sucesores de $\{v1, v3\}$, descartando los que tengan una cota menor o igual a 4 ya que estos resultados no son relevantes si se quiere encontrar el conjunto con cardinalidad máxima. Este procedimiento se repite para los demás sucesores de $\{v1\}$ y luego con los otros sucesores de $\{\}$. Una vez encontrados todos los conjuntos, se escoge el de cardinalidad

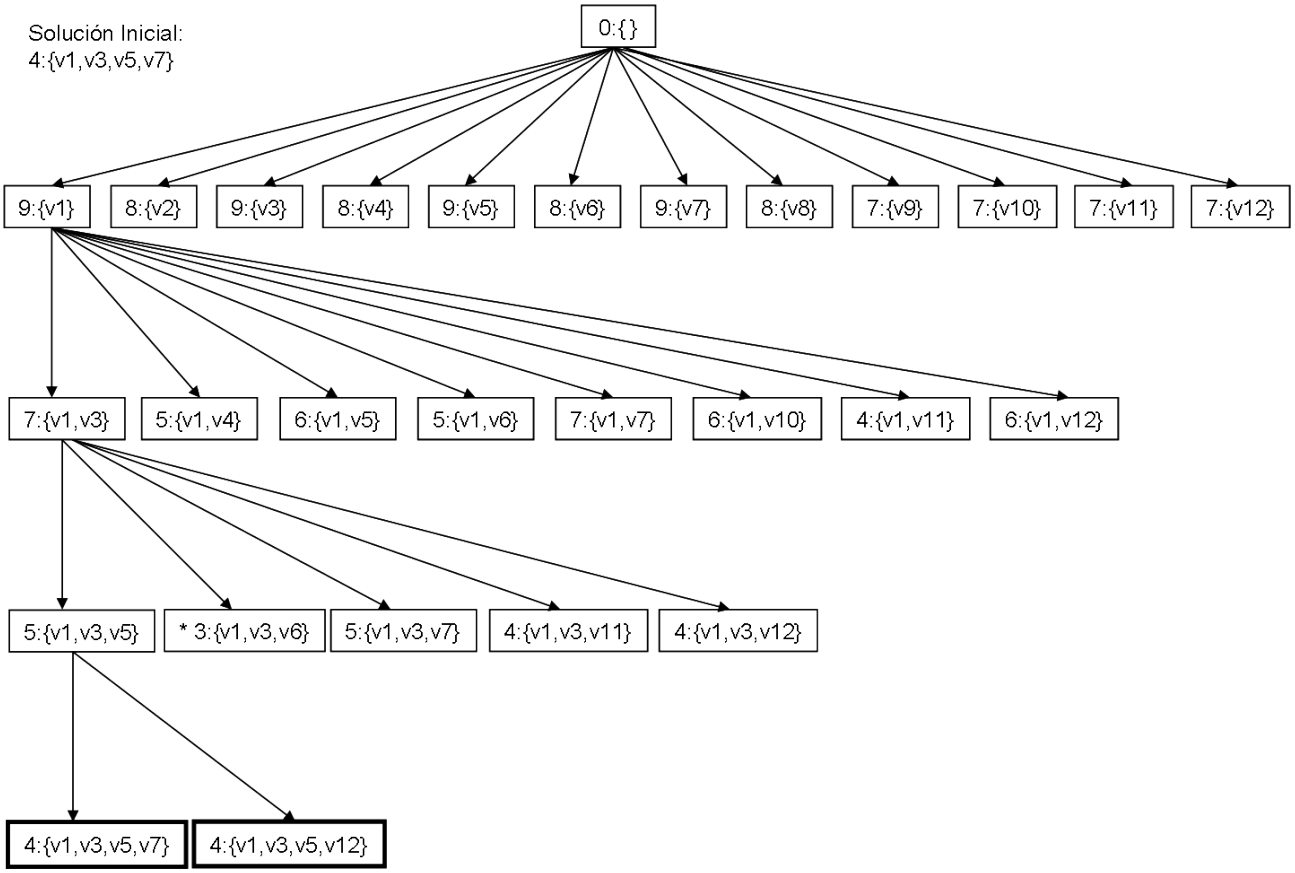


Figura 2: Arbol generado hasta cierta iteración de la ejecución *Branch and Bound*

máxima.

Ejecutando el algoritmo paso por paso, entre los conjuntos de cardinalidad máxima que se obtuvieron, se encuentran: $\{v1, v3, v5, v7\}$, $\{v1, v3, v5, v12\}$, $\{v1, v3, v11, v7\}$, $\{v1, v3, v7, v5\}$, $\{v1, v7, v5, v10\}$, siendo de la misma cardinalidad de la solución inicial.

Es evidente que la expresión $\mid X \mid + \mid \{Y : Y \quad es\ sucesor\ de \quad X\} \mid$ es una cota superior para el máximo independiente que contiene a X, estando en un vértice X, ya que es la suma de la cardinalidad del conjunto independiente de vertices (correspondiente al vértice actual) más la cantidad de elementos que crearían nuevos conjuntos independientes, sin embargo, la expresión no considera la existencia de posibles aristas entre esos elementos. Por ejemplo, si $X \cup \{x1\}$ y X $\cup \{x2\}$ son sucesores de X, si existe una arista $e = (x1, x2)$ con $e \in E$, entonces $X \cup \{x1, x2\}$ no es un conjunto independiente.

Demostración

$\mid X \mid + \mid \{Y : Y \quad es\ sucesor\ de \quad X\} \mid \geq max \mid C_i \mid$ tal que $X \subseteq C_i$

$\equiv <$ Suponiendo el máximo independiente es de la forma: $X \cup \{x1, ..., xm\} >$

$\mid X \mid + \mid \{Y : Y \quad es\ sucesor\ de \quad X\} \mid \geq \mid X \mid + \mid \{x1, ..., xm\} \mid$

$\equiv <$ Por definición de sucesor en el grafo implícito, cada conjunto $X \cup \{x\}$ que sea independiente, es un sucesor de X, por lo tanto: $\{x1, ..., xm\} \subseteq \{Y : Y \quad es\ sucesor\ de \quad X\} >$

$True$

4. Se quiere que resuelva con ramificación y acotamiento (Branch-and-Bound) el siguiente problema: Dado un multiconjunto con n enteros no negativos $A = \{x1, x2, x3, ..., xn\}$, hallar una partición de A en dos multiconjuntos A_1 y A_2, tal que la diferencia, en valor absoluto, de la suma de los elementos en cada multiconjunto sea la mínima posible. Es

decir:
$\mid \sum_{e \in A_1} e - \sum_{e \in A_2} e \mid = min_{\{A,B\} \text{ particion de } A} \mid \sum_{e \in A} e - \sum_{e \in B} e \mid$

a) Describir claramente el grafo implícito (vértices y sucesores) sobre el cual realizará la ramificación. Indique el criterio que utilizará para decidir cuál es el siguiente nodo a expandir.

Grafo Implicito

- ⋆ Vértices: un vértice es un multiconjunto B de dos multiconjuntos B_1 y B_2 de A tal que la unión de B_1 con $B2$ sea subconjunto de A y la intersección de los mismos sea ϕ.
- ⋆ Sucesores: los sucesores de B, siendo B_1 y B_2 los elementos de B, son todos los conjuntos $\{B_1 \cup \{x\}, B_2\}$ y $\{B_1, B_2 \cup \{x\}\}$ tal que $(B_1 \cup \{x\}) \cup B_2 \subseteq A$, $(B2 \cup \{x\}) \cup B_1 \subseteq A$ y $(B_1 \cap \{x\}) \cap B_2 = \phi$, $(B_2 \cap \{x\}) \cap B_1 = \phi$ respectivamente.
- ⋆ Siguiente nodo a expandir: aquel cuya cota sea menor.

b) Describir claramente la función de acotamiento (bounding). Note que puede hacer un pre-procesamiento, por ejemplo, ordenar los elementos de A antes de ir colocándolos en A_1 y A_2.

Función de Acotamiento

Sea B un vértice formado por los multiconjuntos B_1 y B_2, $\mid B_2 \mid \leq \mid B_1 \mid$, y $R = A - (B_1 \cup B_2)$, una cota de la diferencia mínima de los elementos de B viene dada por la siguiente expresión: $\sum_{e \in B_1} e - \sum_{e \in B_2} e - \sum_{e \in R} e$ [1]. Se desea expandir los vértices que tengan el menor valor dado por esta función, esto es que la diferencia entre los multiconjuntos B_1 y B_2 sea menor y tomando en cuenta los elementos que no han sido escogidos hasta el momento, para no caer sólo en soluciones *Greedy*. La idea de podar el árbol de esta forma es para mantener la menor diferencia encontrada y se realiza llevando el multiconjunto de menor suma al valor de la suma del otro multiconjunto. Se podará aquellas ramas cuyo valor sea mayor que la diferencia dada por la solución inicial. Inicialmente $\sum_{e \in R} e = \sum_{e \in A} e$ y una vez escogidos todos los elementos

$\sum_{e \in R} e = 0$. Los elementos en R serán escogidos en forma decreciente.

c) Corra paso a paso su algoritmo con el siguiente ejemplo: $A = \{10, 2, 15, 9, 6, 11\}$

Algoritmo *Greedy*

Para conseguir una solución *Greedy* inicial, ordenamos la entrada de mayor a menor, y se va tomando cada elemento y se guarda en otro conjunto mientras la suma de los elementos de ese conjunto sea menor a $\frac{\sum_{e \in A} e}{2}$. Esto es porque se quiere llegar lo mas cercano posible a la mitad de la suma de los elementos en A para minimizar la diferencia de las sumas de los elementos de la partición. De esta forma, se obtiene la partición $\{\{15, 11\}, \{10, 9, 6, 2\}\}$ cuya diferencia de sus sumas es 1. Esta partición es llamada Partición Perfecta [2], ya que la mínima diferencia es 0 si la suma de los elementos en cualquier conjunto es par y 1 si es impar.

Ejecución *Branch and Bound*

Partiendo de la solución inicial, se inicia el recorrido *Branch and Bound*, Figura 3, podando aquellas ramas cuya diferencia sea mayor a la solución inicial (estas están marcadas con un asterisco, $*$, en el árbol), en cada paso, para extender el vértice (no descartado), se toma el mayor elemento no asignado y con el se crean dos sucesores, agregando el elemento al conjunto cuya suma sea mayor y al conjunto cuya suma sea menor.

La búsqueda se realiza en profundidad partiendo de la raíz, tomando primero el vértice $\{\{15\}\{\}\}$ (similar a la exploración partiendo del nodo $\{\{\}\{15\}\}$), luego como el valor de la diferencia es menor a la solución inicial, se expande a sus sucesores, $\{\{15\}\{11\}\}$ y $\{\{15, 11\}\{\}\}$, se visita primero el sucesor $\{\{15\}\{11\}\}$ debido a que el valor de la diferencia es menor, y se continua con este pro-

cedimiento, expandiendo siempre el más izquierdo hasta llegar a las hojas y luego se retrocede un nivel y expande el hijo derecho, y así sucesivamente, hasta recorrer el árbol completo.

Ya sabíamos que no íbamos a encontrar una solución mejor debido a que la solución *Greedy* era una partición perfecta, sin embargo, se encontraron las otras Particiones Perfectas del conjunto A.

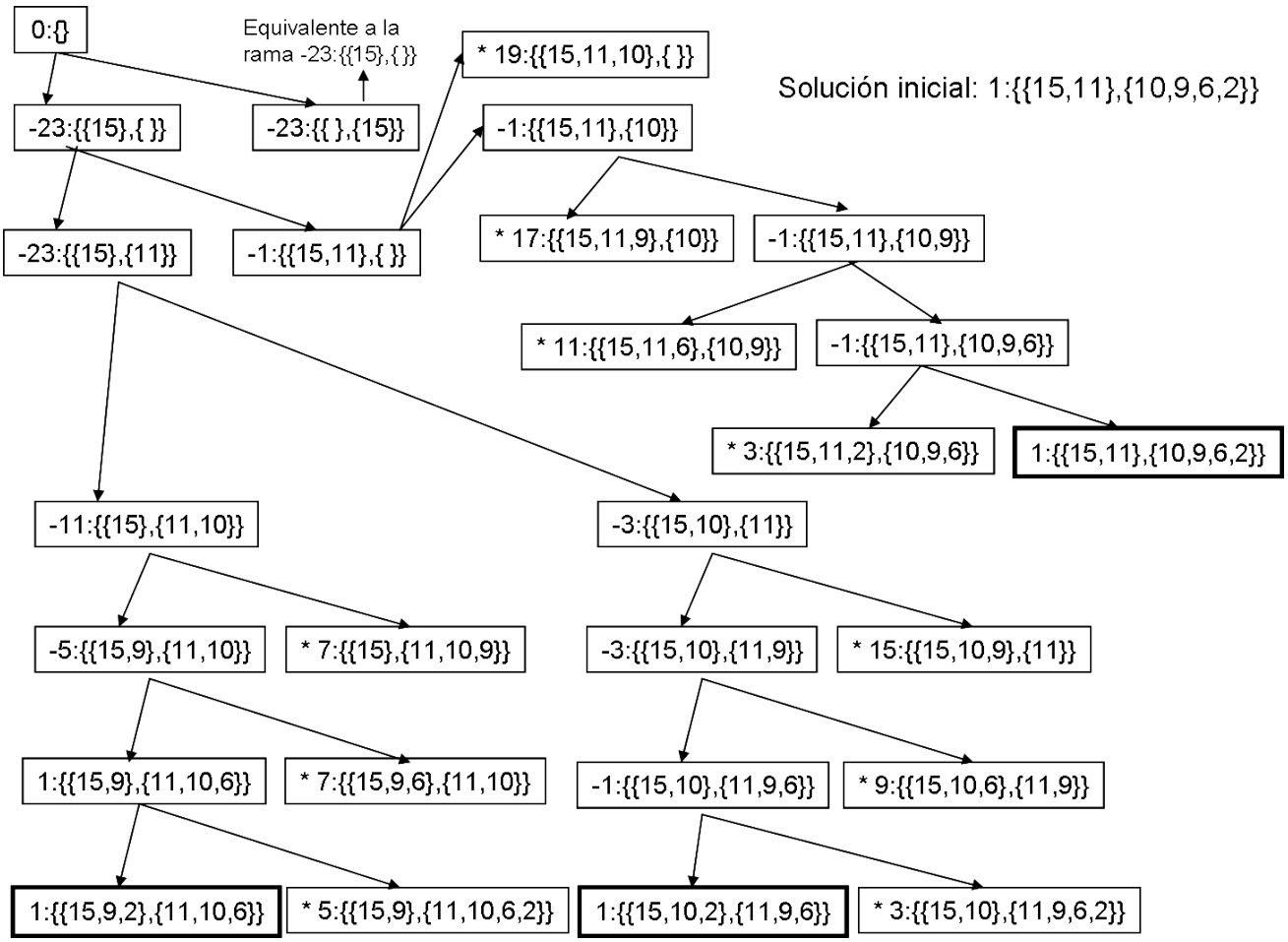


Figura 3: Arbol de la ejecución *Branch and Bound*

Bibliografía

[1] R. E. Korf, "A complete anytime algorithm for number partitioning," *Artif. Intell.*, vol. 106, no. 2, pp. 181–203, 1998.

[2] S. Mertens, "The easiest hard problem: Number partitioning," 2003.