Capítulo 3 Algoritmos Exactos

3.1 Fundamentos de los Algoritmos Exactos

Usando una computadora para contestar preguntas como : ¿Cuántos caminos hay para…..?, ¿Listar todas las posibles soluciones para…?, ¿Hay un camino para…?, usualmente requiere de una búsqueda exhaustiva dentro del conjunto de todas las soluciones potenciales, por eso los algoritmos que resuelven este tipo de problemas reciben el nombre de *algoritmos exactos* o de *búsqueda exhaustiva*. Por ejemplo, si se desean encontrar todos los números primos menores de 10^4, no hay método conocido que no requiera de alguna manera, el examinar cada uno de los números enteros entre 1 y 10^4. De otra manera si se desean encontrar todos los caminos de un laberinto, se deben examinar todos los caminos iniciando desde la entrada [REI96].

Se describirán dos técnicas para organizar tales búsquedas. La primera, "retroceso" (backtracking), ésta trabaja tratando continuamente de extender una solución parcial. En cada etapa de la búsqueda, si una extensión de la solución parcial actual no es posible, se "va hacia atrás" para una solución parcial corta y se trata nuevamente. El método "retroceso" es usado en un amplio rango de problemas de búsqueda, incluyendo el analizar gramaticalmente (parsing), juegos, y calendarización (scheduling). La segunda técnica, "tamiz o criba" (sieves), es el complemento lógico de "retroceso" en que se tratan de eliminar las no-soluciones en lugar de tratar de encontrar la solución. El método "tamiz" es útil principalmente en cálculos numéricos teóricos [REI96].

Se debe tener en mente, sin embargo, que "retroceso" y "tamiz" son solamente técnicas generales. Su aplicación resultará en algoritmos cuyos requerimientos en tiempo son prohibitivos; en general la velocidad de las computadoras no es práctica para una búsqueda exhaustiva de más de 10^8 elementos. Así, para que estas técnicas sean útiles, ellas deben considerar solamente una estructura dentro de la cual se aproxima el problema. La estructura debe ser hecha a la medida, a menudo con gran ingeniosidad, para cuadrar con el problema particular así que el algoritmo resultante será de uso práctico [REI96].

3.1.1 Método de Retroceso (Backtracking)

La idea fundamental de "retroceso" puede ser más fácilmente entendida en el contexto de un juego de laberinto : Se inicia en algún cuadro específico con la meta en algún otro cuadro especificado por una secuencia de movimientos desde un cuadro al próximo cuadro. La dificultad está en que se está restringido por la existencia de barreras que prohiben ciertos movimientos, como se ilustra en la figura 3.1. Un camino para recorrer el laberinto es viajar desde un cuadro de inicio hasta un cuadro final llamado meta de acuerdo a dos reglas [REI96]:

1. Desde el cuadro actual, tomar cualquier camino no explorado previamente.

2. Si el cuadro actual no tiene caminos inexplorados, volver hacia atrás un cuadro en el último camino que se siguió para encontrar caminos viables de explorar.

La primera regla dice como extender el camino actual, si es posible, y la segunda regla dice como regresar cuando se ha estancado. Esta es la esencia del método de "retroceso" : expandiendo la solución actual tanto como sea posible; y cuando la solución ya no pueda ser expandida, regresar y tratar una alternativa en el estado más reciente para el cual hay una alternativa no probada [REI96].

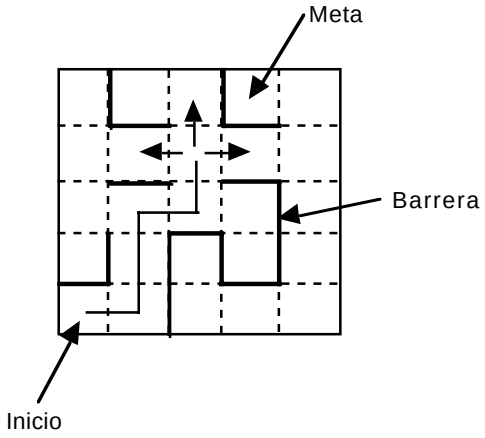


Figura 3.1 Recorrido de un laberinto

3.2 Funcionamiento de un Algoritmo Exacto

En el caso más general, se asume que la solución a un problema consiste de un vector (a_1 ,a_2,...) de longitud finita pero indeterminada, satisfaciendo ciertas restricciones. Cada a_i es un miembro de un conjunto A_i linealmente ordenado. Así la búsqueda exhaustiva debe considerar los elementos A_1 x A_2 x ... x A_i, para i = 0,1,2,..., como solución potencial. Primero se inicia con el vector nulo () como la solución parcial, y las restricciones indican cual de los miembros de A_1 son candidatos para a_1; llamado este subconjunto S_1. Se escoge el menor elemento de S_1 como a_1, y se tiene ahora la solución parcial (a_1). En general, las diversas condiciones que describen las soluciones nos indican cual subconjunto S_k de A_k constituye candidatos para la extensión de la solución parcial (a_1,a_2,...,a_{k-1}) para (a_1,a_2,...,a_{k-1},a_k). Si la solución parcial (a_1,a_2,...a_{k-1}) no admite posibilidades para a_k, entonces $S_k = \varnothing$, y así se regresa hasta el nodo padre y se hace una nueva selección para a_{k-2}, y así sucesivamente [REI96].

El subconjunto de A_1 x A_2 x ... A_i para i = 0,1,2,... que es buscado es representado como un árbol de búsqueda como sigue. La raíz del árbol (el nivel 0) es el vector nulo. Sus hijos son la selección para a_1; y en general, los nodos como el k-ésimo nivel son las selecciones para a_k, dando las selecciones hechas por a_1,a_2,...,a_{k-1} como indicado por los ancestros de estos nodos. En el árbol presentado en la figura 3.2 recorrer los nodos hacia atrás como se indica con las líneas punteadas [REI96].

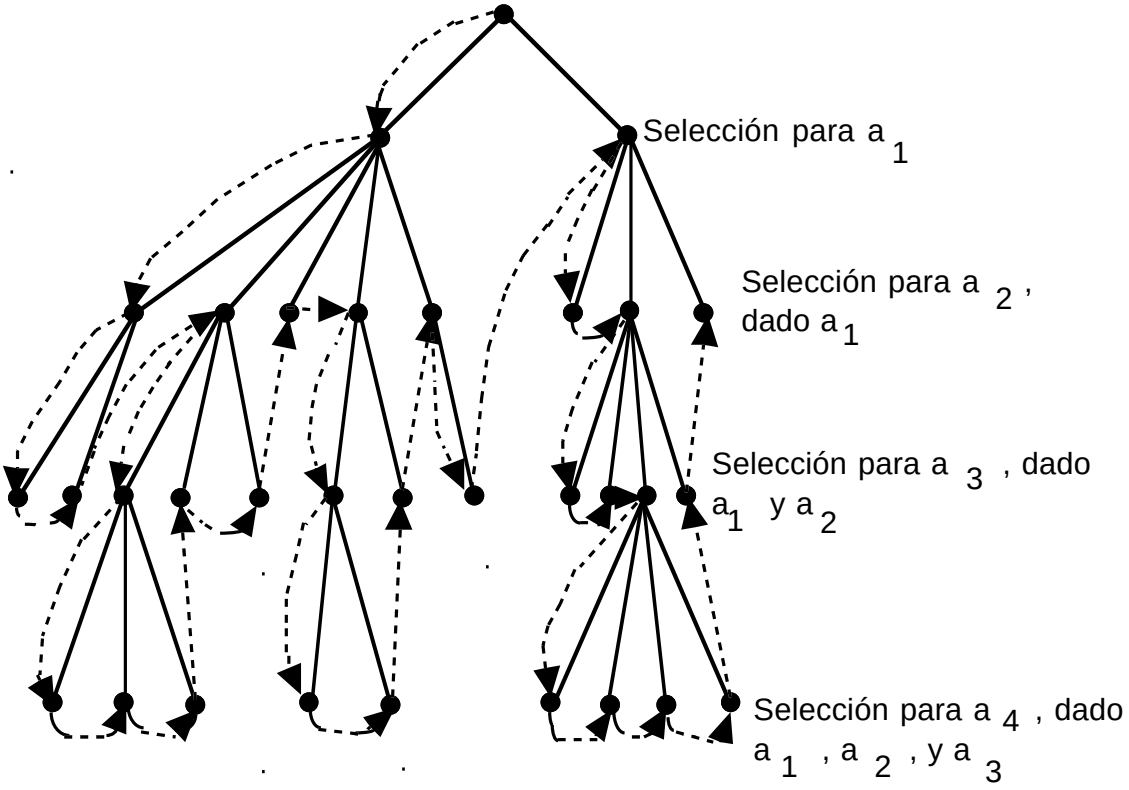


Figura 3.2 Arbol de búsqueda de las soluciones parciales

Cuando se pregunta si un problema tiene una solución (a_1,a_2,….), en realidad se está preguntando si cualquier nodo en el árbol es una solución. Si se pregunta por todas las soluciones, se desean tener todos esos nodos [REI96].

3.3 Algoritmo retroceso generalizado

El algoritmo "retroceso" (backtracking) para encontrar todas las soluciones es descrito formalmente en la figura 3.3, donde si solamente una solución es encontrada, entonces el programa debe parar después de dar una solución; en este caso la terminación al final del ciclo while significa que no existe una solución. Entonces el proceso para, el valor de *contador* es el número de nodos en el árbol que han sido examinados [REI96].

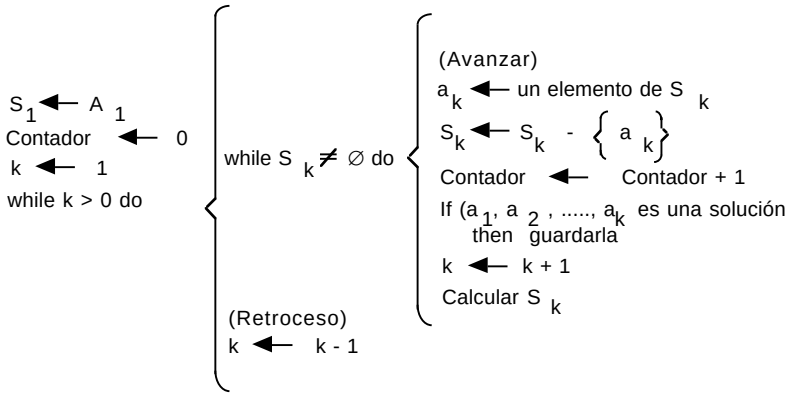


Figura 3.3 Algoritmo retroceso (backtracking) generalizado

3.3.1 Refinamientos

En el programa "retroceso" para encontrar todas las soluciones en un problema determinado, puede hacerse un análisis para acortar el proceso de búsqueda este es llamado método de *Prevención* o *Poda de ramas* (preclusion o branch pruning), por el efecto de remover subárboles del árbol. Un problema puede podarse aún más y las soluciones pueden ser consideradas como equivalentes si se hace una serie de reflexiones. Es claro que si se encuentran soluciones equivalentes, se puede producir fácilmente un conjunto de todas las soluciones y de esa manera se poda un gran subárbol del árbol [REI96].

Otro refinamiento de esta técnica es el método de *Fusión u Ordenamiento de Ramas* (fusion o branch merging), donde la idea es evitar hacer dos veces el mismo trabajo : si dos o más subárboles del árbol son isomórficas, se puede buscar solamente en uno de ellos. El hecho de ahorrar con este método no es trivial dado que puede haber un gran ahorro en el número total de nodos que intervienen en la búsqueda [REI96].

Además para el método de fusión u ordenamiento, el cual claramente puede acortar la búsqueda si se busca la solución en el árbol entero, hay una técnica heurística que puede ser útil cuando la existencia de una solución está en duda o cuando solamente se encuentra una solución en lugar de todas las soluciones. Esta técnica, es llamada *árbol de arreglos* (tree arrangement) y puede ser usado en varias formas. Primero, si la evidencia indica que todas las soluciones tendrán una forma particular, entonces esta es podada para estructurar la búsqueda así que las soluciones potenciales de esa forma son inspeccionadas antes que otras soluciones. Segundo; si es posible, el árbol debe ser rearreglado así que los nodos de bajo grado (por ejemplo esos con relativamente pocos hijos) estén cerca de la parte de arriba del árbol; por ejemplo, el árbol de la figura 3.4.a es preferible a uno como el de la figura 3.4.b [REI96].

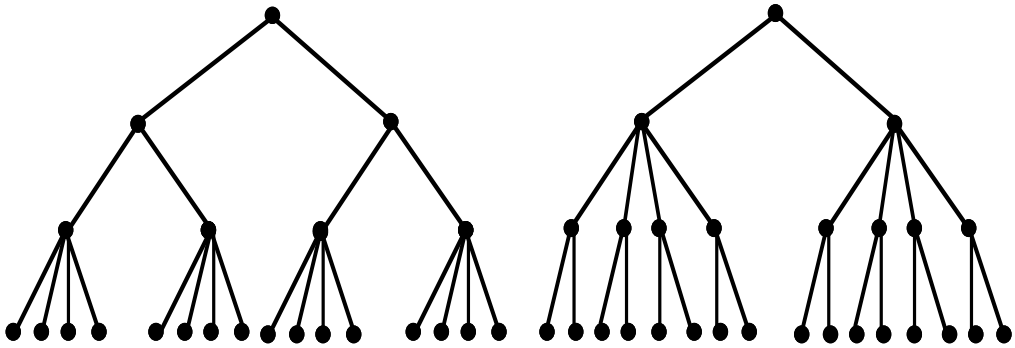


Figura 3.4 Dos posibles árboles para ser examinados con retroceso

De esto nace la pregunta de ¿porqué esto es útil?; generalmente, para descubrir que camino no puede ser tratado para una solución, varias restricciones deben ser acumuladas, y normalmente esto sucede en una profundidad fijada desde la parte de arriba de los árboles. El resultado es que más nodos serán podados si una gran parte de

los abanicos (fanout) ocurren cerca de las hojas que cerca de la raíz. Por supuesto, debe entenderse que este tipo de arreglo de árbol puede no ayudar si se debe buscar en el árbol entero. Más aún, si en el árbol entero no se necesita buscar, la búsqueda en el rearreglo puede mover las soluciones más cerca pero más lejos del inicio de la búsqueda [REI96].

Un último refinamiento es la técnica del problema de descomposición (problem decomposition), donde se descompone el problema en k subproblemas, se resuelven los subproblemas, y entonces se componen las soluciones para los subproblemas en una solución del problema original. La demanda de almacenamiento puede ser alta porque todas las subsoluciones deben ser guardadas, pero el incremento en la velocidad puede ser considerable. Por ejemplo si toma en tiempo $c \cdot 2^n$ para resolver el problema de tamaño n, entonces el tiempo es reducido a $k \cdot c \cdot 2^{n/k}$ + T, donde T es el tiempo requerido para componer las subsoluciones. Si el número de subsoluciones es pequeño y si no es tan difícil componerlas, entonces T será relativamente pequeño y la técnica resultará en un enorme ahorro [REI96].

Las variaciones de retroceso como son la técnica de Búsqueda Exhaustiva Ingenua (Naive Exhaustive Search), la de Ramificación y Acotamiento (Naive Branch and Bound) y la búsqueda de Una Mejor Ramificación y Acotamiento (A better Branch and Bound) son un tipo específico de prevención. La prevención basada en el hecho de que cada una de las soluciones tiene un costo asociado y que la solución óptima (el costo que sea menor o mínimo) será encontrada [REI96].

La tres técnicas de Búsqueda antes mencionadas están basadas en un árbol de búsqueda, donde en cada etapa todas las posibles soluciones del problema son particionadas en dos o más subconjuntos, cada uno representado por nodos en un árbol de decisiones [LAW85]. Esta búsqueda (particionamiento) se realiza de acuerdo a alguna heurística, la cual reduce la búsqueda que conduce a la solución óptima. Después de ramificar, las cotas más pequeñas son computadas sobre el costo de cada uno de los dos subconjuntos [LAW85].

3.4 Método de Búsqueda Exhaustiva Ingenua

El algoritmo de Búsqueda Exhaustiva Ingenua consta de un conjunto *N* de puntos llamados nodos, éstos son [CS-95]:

$$N = \{1, 2, ..., n\}$$

Donde *n* es el número total de nodos en el grafo (número de vértices o número de ciudades), y cuyo objetivo será generar todas las posibles soluciones y tomar las más cortas. *S* es un conjunto ordenado de nodos el cual incluye un camino parcial (que contiene una lista ordenada de *k* enteros) y la suma de los *pesos* de sus arcos [CS-95] :

$$S = (k, [i1, i2, ..., ik], peso)$$

Donde W(i, j) es la matriz de adyacencia o de costos que representa un problema PAV, un ejemplo de una matriz de adyacencia para 4 ciudades se muestra a continuación [SMI89] :

De \ al	V1	V2	V3	V4
V1	0	3	5	12
V2	3	0	6	4
V3	5	7	0	2
V4	8	1	4	0

Tabla 3.1 Ejemplo de PAV de 4 ciudades

Se genera un circuito y se compara el peso de ese circuito con el peso del circuito inicial que recibe el nombre de Best_S_soFar. Si el circuito recién calculado resulta ser mejor entonces este es nuestro nuevo conjunto Best_S_soFar, de lo contrario se retiene el anterior. El proceso se detiene cuando ya ningún circuito mejora en peso al del conjunto Best_S_so_Far [CS-95].

Los procedimientos que realizan la tarea de búsqueda exhaustiva son dos. El primero de ellos descrito en la figura 3.5.a que se llama Búsqueda_Exhaustiva, este es el encargado entre otras cosas de calcular el peso inicial y de llamar a Búsqueda_Exhaustiva_Rec. En la figura 3.5.b Búsqueda_Exhaustiva_Rec recursivamente va generando el árbol de búsquedas hasta encontrar el costo mínimo [CS-95].

Este algoritmo busca todos los (n - 1)! posibles caminos iniciando en 1, guardando el mejor. Hay mucho de paralelismo, porque después de k recursivos llamados a Búsqueda_Exhaustiva_Rec, hay (n - 1)*(n - 2)*…..*(n - k) subárboles independientes para buscar, el cual puede ser realizado en muchos procesadores. Ya que los subárboles son igualmente largos se dice que el balance de carga es perfecto [CS-95]. El hecho de que se realice una búsqueda de todos los (n-1)! distintos caminos en el peor de los casos implica que tiene un O(n!) [CS-95].

```
{01} Procedure  Búsqueda_Exhaustiva_Ingenua
{02}    peso              = w(1, 2) + w(2, 3) + w(3, 4) + ….. + w(n-1, n) + w(n, 1);
{03}    Best_S_so_far = ( n, [ 1, 2, 3, ….., n-1, n ], peso );
{04}    S                 = ( 1, [ 1 ], 0 );
{05}    Búsqueda_Exhaustiva_Rec ( S, Best_S_so_far);
{06}    Imprime_Resultados          ( Best_S_so_far);
{07} end de Búsqueda_Exhaustiva_Ingenua;
```

Figura 3.5.a Algoritmo que llama a Búsqueda_Exhaustiva_Rec

```
{08} Procedure Búsqueda_Exhaustiva_Rec ( S, Best_S_so_far )
{09}    Let  S                  = ( k, [ i1, i2, ….., ik ], peso );
{10}    Let Best_S_so_far = ( n, [ i1B, i2B, ….., inB ], pesoB );

{11}    If k = n
{12}       then
```

```
{13}          new_peso = peso + A(ik, i1);
{14}          if new_peso < pesoB
{15}            then
{16}               Best_S_so_far = ( k, [ i1, i2, ….., ik ], new_peso );
{17}          end if;
{18}       else
{19}          for all j not in [ i1, i2, ….., ik ];
{20}             new_peso = peso + A(ik, j);
{21}             New_S = ( k + 1, [ i1, i2, ….., ik, j ], new_peso );
{22}             Búsqueda_Exhaustiva_Rec ( New_S, Best_S_so_far );
{23}          end for;
{24}       end if;
{25} return
{26} end de Búsqueda_Exhaustiva_Rec;
```

Figura 3.5.b Algoritmo recursivo de Búsqueda Exhaustiva Ingenua

En todos los algoritmos se numerarán las líneas en orden ascendente para poder explicar en caso necesario alguna de ellas haciendo referencia al número. Así en la Figura 3.5 :

- {02} Calcular el valor inicial del peso *peso* , donde el peso depende del recorrido inicial que puede ser uno escogido arbitrariamente, así que en este método se escogió {1,2,3,4, …..n}, el cálculo del peso matemáticamente se representa como :

$$peso = A(n,1) + \sum_{i=1}^{m=n-1} A(i, i+1)$$

  se agrega el valor A(n,1) para que pueda cerrarse el circuito y su peso respectivo sea agregado.
- {03} Inicializar el circuito como :

  nodo(i) = i con i = 1,2,3,….., n

  donde n es el número total de nodos, vértices o de ciudades. El nodo(i) se deja en el registro Best_S_so_Far.
- {04} El registro S contendrá el circuito que se va formando en la recursividad, debe ser inicializado en (1, [1], 0) que significa que es el primer nodo, el conjunto que contiene la ruta que se va formando contiene al nodo No. 1 cuyo peso es 0.
- {05} Llamar al procedimiento de Búsqueda_Exhaustiva_Rec para la búsqueda del costo mínimo y la ruta respectiva.
- {09} y {10} Recibir la información que llega a las variables del procedimiento
- {11} Si k = n significa que ya se terminó de generar un circuito parcial, es decir deja de ser circuito parcial para ser circuito total y debe,
- {12} terminar de calcular el peso New_peso, al que se le sumará el peso que se encuentra en la posición A(S,nodo(k),1). Donde k es el último nodo del circuito. La columna debe ser la 1 para cerrar el circuito al nodo 1, porque este fue el inicial.
- {14} a - {17} Guardar en el registro New_S ← S siempre y cuando el peso New_peso recién calculado sea menor que el peso pesoB anexando los datos recién calculados.
- {19}a - {24} Se crean los circuitos con sus respectivos pesos y repetir la operación en

forma recursiva.

3.5 Método Ramificación y Acotamiento ingenuo

El método Ramificación y Acotamiento ingenuo es también llamado Naive Branch and Bound. Este se obtiene al hacerle una mejora al método de Búsqueda Exhaustiva Ingenua, esta consiste en hacer una poda al árbol de búsqueda. ¿Pero cómo?, se observa cada una de las rutas parciales y si alguna de ellas ya es mayor que la de la mejor solución encontrada con anterioridad no hay razón para seguir buscando por ese camino. Por lo tanto se evita seguir en él y esto es lo que se llama poda del árbol de búsqueda. Esto se ve reflejado en el procedure de Ramificación_Acotamiento_ingenuo_Rec que se muestra en la figura 3.6 [CS-95].

```
{01}Procedure Ramificación_Acotamiento_ingenuo_Rec ( S, Best_S_so_far )
{02}   Let  S                = ( k, [ i1, i2, ….., ik ], peso )
{03}   Let Best_S_so_far = ( n, [ i1B, i2B, ….., inB ], pesoB )

{04}   If k = n
{05}     then
{06}          new_peso = peso + A(ik, i1)
{07}          if new_peso < pesoB
{08}            then
{09}               Best_S_so_far = ( k, [ i1, i2, ….., ik ], new_peso )
{10}          end if
{11}     else
{12}        for all j not in [ i1, i2, ….., ik ]
{13}          new_peso = peso + A(ik, j)
{14}          if new_peso < pesoB
{15}            then
{16}               New_S = ( k + 1, [ i1, i2, ….., ik, j ], new_peso )
{17}               Naive_Branch_and_Bound_Search ( New_S, Best_S_so_far )
{18}          end if
{19}        end for
{20}     end if
{21}  return
{22} end de Ramificación_Acotamiento_ingenuo_Rec
```

Figura 3.6 Algoritmo recursivo de Búsqueda Exhaustiva Ingenua

Como puede verse el algoritmo es similar al de Búsqueda Exhaustiva Ingenua, por lo que solo se hará hincapié en las líneas que marcan la diferencia, que son :

-{14} a -{18} Se compara si el peso recién calculado new_peso es menor que el pesoB anterior. Si es menor guarda como Best_S_so_Far al calculado, pero si resulta ser mayor el peso nuevo New_peso entonces ya no llama a la recursividad. ¡He aquí el ahorro!, dado que ya no termina los circuitos parciales que están en estas circunstancias reduciendo las búsquedas de los (n-1)! posibles caminos del

método anterior. Pero cabe aclarar que también en el peor de los casos hace la búsqueda de los (n-1)! caminos siendo por ello de O(n!) [CS-95].

3.6 Método Una Mejor Ramificación y Acotamiento

El método Una Mejor Ramificación y Acotamiento también recibe el nombre de A Better Branch and Bound. Para que sea aplicable, el costo debe estar bien definido, es decir la matriz de adyacencia debe representar un grafo completo [REI96].

Este método está basado como los anteriores en un árbol de búsquedas, donde en cada etapa todas las posibles soluciones del problema son particionadas en dos subconjuntos, cada una representada por nodos en un árbol de decisiones. Al particionar en dos subconjuntos en cada etapa, el subconjunto de la izquierda contendrá una arista específica (i,j) y el subconjunto de la derecha no tendrá esa arista (i,j). Esta ramificación se realiza de acuerdo a la heurística descrita en la sección 3.6.1, la cual reduce la cantidad de búsquedas que conducen a la solución óptima. Después de ramificar se calculan las cotas para cada uno de los dos subconjuntos. En el próximo espacio de soluciones que se busque se debe escoger uno de ellos, que es el mínimo de las dos cotas a ser comparadas. Este proceso se repite recursivamente hasta encontrar el ciclo hamiltoniano. Entonces, solamente dentro de los subconjuntos de soluciones se buscará la cota que sea más baja que el valor de la cota inicial o de arranque [REI96].

Esta forma de ramificar y acotar la solución nos permite descartar un buen número de subconjuntos de solución, esto evita realizar una búsqueda infructuosa. La *heurística consiste en reducir* en el proceso computacional las cotas más bajas, dicha reducción se describirá en seguida, para ello se usará una matriz de adyacencia que contiene 6 ciudades para la solución del problema del agente viajero (PAV) el cual fue adaptado de [REI96], ver la tabla 3.2 [ARR96].

3.6.1 Heurística de Reducción para el método Una Mejor Ramificación y Acotamiento

Un ciclo hamiltoniano de longitud n contiene exactamente un *elemento de cada fila* de la matriz de costo o adyacencia *W* y exactamente un *elemento de cada columna* de W. Si a la matriz W[i,j] se le resta una constante *q* de cualquier fila o de cualquier columna, el costo de todos los ciclos Hamiltonianos (circuitos) son reducidos por *q* .Más aún los costos relativos de los diferentes ciclos restantes también son reducidos por q, lo mismo que el circuito óptimo. Si tal substracción es hecha de las filas y las columnas, de tal manera que cada fila y cada columna contenga al menos un cero pero además que no guarde elementos negativos, entonces la cantidad restada será la cota más baja en el costo de cualquier solución. Este proceso de restar constantes de las filas y las columnas es llamada *reducción* , ver la tabla 3.2 [REI96].

i\j	V1	V2	V3	V4	V5	V6
V1	∞	3	93	13	33	9
V2	4	∞	77	42	21	16
V3	45	17	∞	36	16	28
V4	39	90	80	∞	56	7
V5	28	46	88	33	∞	25
V6	3	88	18	46	92	∞

Tabla 3.2 Matriz de Adyacencia de 6 ciudades para el PAV

Donde las Vi son los vértices, nodos o ciudades del PAV y la matriz W(i,i) debe contener infinitos (∞) para excluir estos nodos que se encuentran sobre la diagonal. Esto se debe a que la matriz W representa un grafo completo. Para reducir la matriz de adyacencia o de costos se localizan los menores de cada fila y se restan a la matriz de costos. Se localizan los menores de cada columna y se restan a la matriz de costos, dejando la matriz reducida. Por otro lado se suman todos los menores de cada fila a todos los menores de cada columna para obtener la cota actual mas baja que es la cota mas baja de todas las soluciones del problema. Para este ejemplo se restarían 3, 4, 16, 7, 25 y 3 de las filas y 15 y 8 de las columnas tres y cuatro respectivamente, y la cota actual mas baja es de 81 [ARR96]. Ver la tabla 3.3 que muestra cada paso descrito anteriormente.

	V1	V2	V3	V4	V5	V6	menores
V1	∞	3	93	13	33	9	3
V2	4	∞	77	42	21	16	4
V3	45	17	∞	36	16	28	16
V4	39	90	80	∞	56	7	7
V5	28	46	88	33	∞	25	25
V6	3	88	18	46	92	∞	3
							58

a) Matriz de Adyacencia y menores de cada fila

	V1	V2	V3	V4	V5	V6
V1	∞	0	90	10	30	6
V2	0	∞	73	38	17	12
V3	29	1	∞	20	0	12
V4	32	83	73	∞	49	0
V5	3	21	63	8	∞	0
V6	0	85	15	43	89	∞

b) Matriz de Adyacencia despues de restar menores de cada fila

	V1	V2	V3	V4	V5	V6	menores
V1	∞	0	90	10	30	6	3
V2	0	∞	73	38	17	12	4
V3	29	1	∞	20	0	12	16
V4	32	83	73	∞	49	0	7
V5	3	21	63	8	∞	0	25
V6	0	85	15	43	89	∞	3
menores	0	0	15	8	0	0	81

c) Matriz de Adyacencia y menores de cada columna

	V1	V2	V3	V4	V5	V6
V1	∞	0	75	2	30	6
V2	0	∞	58	30	17	12
V3	29	1	∞	12	0	12
V4	32	83	58	∞	49	0
V5	3	21	48	0	∞	0
V6	0	85	0	35	89	∞

cotaactual = 81

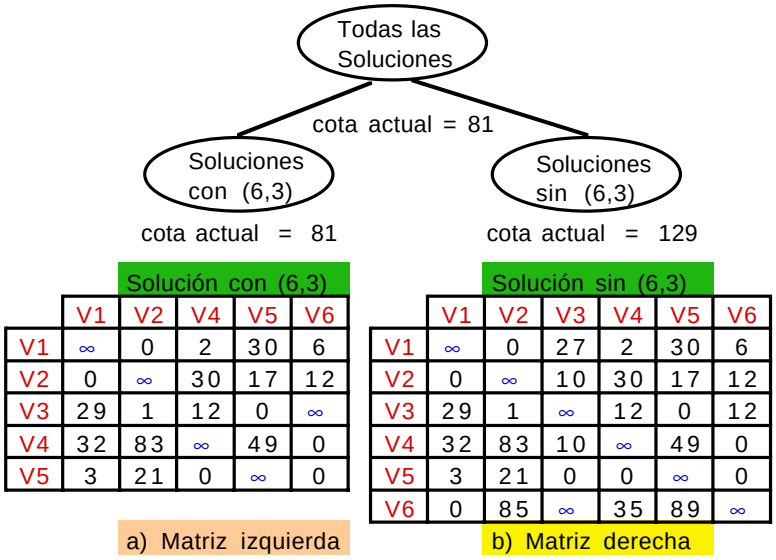
d) Matriz de Adyacencia o de Costos Reducida despues de restar menores de cada columna

Tabla 3.3 Pasos para calcular la primer Matriz Reducida

Ahora la pregunta sería ¿Cómo se divide el conjunto de todas las soluciones en dos clases?. Supóngase que se escogen los nodos (6, 3) entonces se divide el espacio de soluciones en dos, generando dos matrices. La matriz derecha será el conjunto que contendrá todas las soluciones que excluyen los nodos (6, 3), y por lo tanto sabiendo que (6, 3) son excluidos se debe cambiar la entrada de la matriz de costos en esa dirección a $W_{6,3} \leftarrow \infty$ [ARR96]. Lo que implica que a la matriz de costos se le restará 48 de la tercera columna y nada de la sexta fila, así se obtiene la cota actual mas baja de 81 + 48 = 129

para todas las soluciones que excluyen los nodos (6, 3) [ARR96]. En este mismo paso de división de soluciones se obtiene otra matriz, la matriz izquierda que contiene todas las soluciones que incluyen los nodos (6, 3), por lo que la sexta fila y la tercera columna debe ser borrada de la matriz de costos reducida porque hora ya nunca se podrá ir desde el nodo 6 a ningún otro nodo o llegar al nodo 3 desde ningún otro nodo [ARR96].

El resultado es una matriz de costos de dimensión (5x5) que es menor en uno que la anterior de (6x6), mas aún, ya que todas las soluciones de este subconjunto usan los nodos (6, 3), los nodos (3, 6) no serán usados nunca mas, por lo que se debe cambiar la entrada de la matriz de costos en esa dirección a $W_{3,6} \leftarrow \infty$ para prohibir estos nodos. El árbol de búsqueda binario en este momento es como se presenta en la figura 3.7 con sus respectivas matrices derecha e izquierda [ARR96].



	V1	V2	V4	V5	V6
V1	∞	0	2	30	6
V2	0	∞	30	17	12
V3	29	1	12	0	∞
V4	32	83	∞	49	0
V5	3	21	0	∞	0

	V1	V2	V3	V4	V5	V6
V1	∞	0	27	2	30	6
V2	0	∞	10	30	17	12
V3	29	1	∞	12	0	12
V4	32	83	10	∞	49	0
V5	3	21	0	0	∞	0
V6	0	85	∞	35	89	∞

Figura 3.7 Primera división del espacio de búsqueda

Los nodos (6, 3) fueron usados para la solución porque, de todos los nodos, estos causaron el mayor incremento en la cota actual mas baja del subárbol derecho (en el árbol de búsqueda de la figura 3.7). De manera que la solución óptima se encontrará en las ramas izquierdas más que en las ramas derechas, las ramas izquierdas reducen la dimensión del problema, sin embargo las ramas derechas solo agregan otro ∞ y quizás otros pocos ceros sin cambiar la dimensión de la matriz. Para seleccionar los mejores nodos, se escoge el cero que cuando cambie a ∞ permita restarle a esta fila y esa columna la mayor cantidad posible, esto se muestra en la tabla 3.4 [ARR96].

Así en la matriz de costos (5x5) que representa el problema del subárbol izquierdo en la figura 3.7, el mejor cero esta en (4, 6) y este es cambiado a ∞. Esto implica que a la matriz de costos se le resta 32 de la cuarta fila y nada de la sexta columna. Este cero es el que tiene la mayor suma de los menores de entre todos los ceros que hay en la matriz. Por lo tanto la siguiente división se hará con los nodos (4, 6). Esto incrementa la cota actual mas baja de todas las soluciones que incluyen al nodo (6, 3) y que excluyen los nodos (4,

6) a 81 + 32 = 113 [ARR96]. Esto decrementa la dimensión de la matriz en la rama izquierda a una matriz de (4x4), ya que debe borrarse la fila 4 y la columna 6. Esta situación se presenta en la figura 3.8, donde se muestra el árbol generado hasta el momento con sus respectivas matrices derecha e izquierda [ARR96].

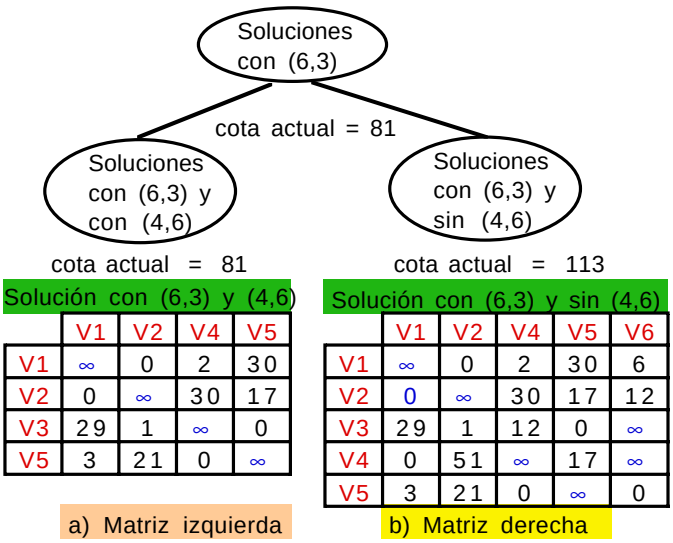
	V1	V2	V3	V4	V5	V6	menores
V1	∞	0	75	2	30	6	2
V2	0	∞	58	30	17	12	12
V3	29	1	∞	12	0	12	1
V4	32	83	58	∞	49	0	32
V5	3	21	48	0	∞	0	0
V6	0	85	0	35	89	∞	0
menores	0	1	48	2	17	0	

a) Matriz de costos Reducida

Coordenadas	cotamejornd	Coordenadas	cotamejornd
1,2	2+1 = 3		
2,1	12+0 = 12		
3,5	1+17 = 18		
4,6	32+0 = 32		
5,4	0+2 = 2	5,6	0+0
6,1	0+0 = 0	6,3	0+48
	(r, c) = (6,3)	cotamejornd =	48

b) Coordenadas del mejor cero y su respetiva cota

Tabla 3.4 Selecciona el primer mejor cero



	V1	V2	V4	V5
V1	∞	0	2	30
V2	0	∞	30	17
V3	29	1	∞	0
V5	3	21	0	∞

	V1	V2	V4	V5	V6
V1	∞	0	2	30	6
V2	0	∞	30	17	12
V3	29	1	12	0	∞
V4	0	51	∞	17	∞
V5	3	21	0	∞	0

Figura 3.8 Segunda división del espacio de búsqueda

Note que ya que los nodos (4, 6) y (6, 3) son incluidos en la solución, los nodos (3, 4) no se usarán nunca mas, esto se refuerza poniendo como entrada en $W_{3,4} \leftarrow \infty$. En general, si el nodo que se agrega a la ruta parcial va desde i_u a j_1 y la ruta parcial contiene caminos (i_1, i_2, ….., i_u) y (j_1, j_2, ….., j_k), los nodos cuyo uso deben prohibirse están en (j_k, i_1). Cuando se hace una nueva división se encuentran nuevamente los mejores nodos que

son (2,1), así que a la matriz de costos derecha se le prohiben los nodos (2, 1) al hacer $W_{2,1} \leftarrow \infty$ y esto hace que se le reste 17 a la segunda fila y 3 a la primera columna como se muestra en la tabla 3.5 [ARR96].

Solución con (6,3) y (4,6)

	V1	V2	V4	V5	Coordenadas	cotamejornd
V1	∞	0	2	30	1,2	2 + 1 = 3
V2	0	∞	30	17	2,1	17 + 3 = 20
V3	29	1	∞	0	3,5	1 +17 = 18
V5	3	21	0	∞	5,4	3 + 2 = 5

a) (r,c) = (2,1) cotamejornd = 20

Solución con (6,3) y (4,6) y (2,1)

	V2	V4	V5
V1	∞	2	30
V3	1	∞	0
V5	21	0	∞

b) Matriz izquierda

Solución con (6,3) con (4,6) y sin (2,1)

	V1	V2	V4	V5
V1	∞	0	2	30
V2	∞	∞	13	0
V3	26	1	∞	0
V5	0	21	0	∞

c) Matriz derecha

Tabla 3.5 Selección del tercer mejor cero y división del espacio de búsqueda

Después de dividir el espacio de búsqueda en la matriz derecha e izquierda de la tabla 3.5, la matriz de costos izquierda es una matriz de (3x3). En la que se han incluido los nodos (2,1) a la solución izquierda por lo que los nodos (1, 2) se prohiben en la matriz derecha al hacer $W_{1,2} \leftarrow \infty$.

Se realiza nuevamente la reducción de la matriz de (3x3) restando 1 de la columna dos y 2 de la fila uno ahora quedando una nueva matriz de costos reducida. La cota actual mas baja también se actualiza dado que en la solución de este subconjunto es de 81 + 1 +2 = 84, como se muestra en la tabla 3.6 [ARR96].

Solución con (6,3) y (4,6) y (2,1)

	V2	V4	V5	menores
V1	∞	2	30	2
V3	1	∞	0	0
V5	21	0	∞	0
				2

a) Matriz de Adyacencia y menores de cada fila

	V2	V4	V5
V1	∞	0	28
V3	1	∞	0
V5	21	0	∞

b) Matriz de Adyacencia después de restar menores de cada fila

	V2	V4	V5	menores
V1	∞	0	28	2
V3	1	∞	0	0
V5	21	0	∞	0
menores	1	0	0	3

c) Matriz de Adyacencia y menores de cada columna

Solución con (6,3) y (4,6) y (2,1)

	V2	V4	V5
V1	∞	0	28
V3	0	∞	0
V5	20	0	∞

cotaactual = 81+2+1 = 84

d) Matriz de Adyacencia o de Costos Reducida después de restar menores de cada columna

Tabla 3.6 Pasos para calcular la tercer Matriz Reducida

Nótese que después de n-2 nodos seleccionados, la matriz de costos es una matriz de dimensión (2x2), en este momento las dos últimas parejas de nodos se forzan para formar un camino. En el ejemplo se tienen los nodos (6, 3), (4, 6), (2, 1), y (1, 4) así que solo queda agregar los nodos (3,5) y (5,2) para completar la ruta del PAV [ARR96].

Se obtiene una ruta al ir guardando las filas que se van quitando, es decir cuando se encontraron los nodos (6,3) se guardo en la dirección de la fila 6 la dirección 3, cuando se encontraron los nodos (4,6) se guardo en la dirección de la fila 4 dirección 6 y así sucesivamente, como se muestra en la tabla 3.7 [ARR96].

(6,3)

filas	1	2	3	4	5	6
quitarfilas						3

a) El nodo 6 va al nodo 3

(6,3) con (4,6)

filas	1	2	3	4	5	6
quitarfilas		1		6		3

b) El nodo 4 va al nodo 6

6,3), (4,6) con (2,1)

filas	1	2	3	4	5	6
quitarfilas		1		6		3

c) El nodo 2 va al nodo 1

6,3), (4,6), (2,1) con (1,4)

filas	1	2	3	4	5	6
quitarfilas	4	1		6		3

d) El nodo 1 va al nodo 4

6,3), (4,6), (2,1), (1,4) con (3,5)

filas	1	2	3	4	5	6
quitarfilas	4	1	5	6		3

e) El nodo 3 va al nodo 5

6,3), (4,6), (2,1), (1,4), (3,5) con (5,2)

filas	1	2	3	4	5	6
quitarfilas	4	1	5	6	2	3

f) El nodo 5 va al nodo 2

Tabla 3.7 Ruta encontrada a partir de los mejores nodos seleccionados

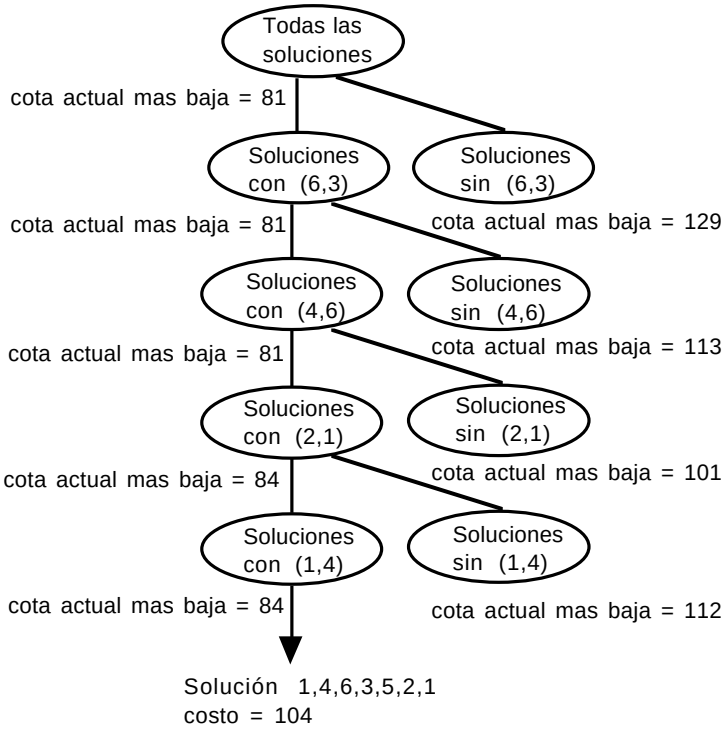


Figura 3.9 Arbol binario de A Better Branch and Bound

Con la historia de las filas que se quitaron se encuentra la ruta final siguiendo las direcciones. La ruta final para este ejemplo es 1, 4, 6, 3, 5, 2, 1, cuyo costo es de 104, los nodos en el árbol de búsqueda se muestran en la figura 3.9 [ARR96]. Al analizar el costo y las cotas actuales mas bajas del árbol binario de la figura 3.9 puede notarse que hay un subárbol con la cota actual más baja = 101 que es menor que el costo de la ruta que se obtuvo por lo que ese subárbol debe ser examinado y expandido porque existe la posibilidad de que haya otra ruta con un costo menor que el de la ruta anterior. La cota actual mas baja = 101 incluye los nodos (6, 3) y (4,6) pero excluye los nodos (2, 1), la matriz de costos asociada a estos nodos es la que se presenta en la tabla 3.8 [ARR96].

Solución con (6,3) con (4,6) y sin (2,1)

	V1	V2	V4	V5
V1	∞	0	2	30
V2	∞	∞	13	0
V3	26	1	∞	0
V5	0	21	0	∞

b) Matriz derecha

Tabla 3.8 Matriz de costos del subárbol con cota actual mas baja = 101

Esta matriz de costos (4x4) debe ser dividida al encontrar el mejor cero en (5, 1), esto excluye los nodos (5, 1) y suma 26 a la cota actual mas baja dando 101+ 26 = 127 generando el subárbol que se muestra en la figura 3.10 [ARR96].

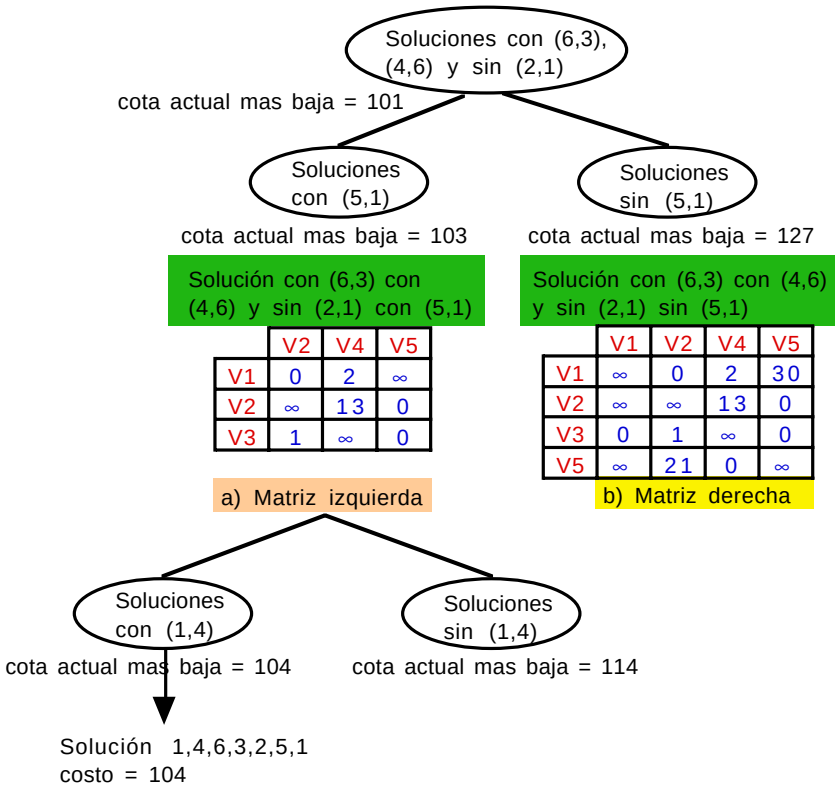


Figura 3.10 División del espacio de búsqueda del subárbol con cota actual mas baja = 101

De esta manera se obtienen dos soluciones óptimas o dos rutas diferentes (1, 4, 6, 3, 5, 2, 1) y (1,4,6,3,2,5,1) cada una con un costo de 104 que es el costo menor de todas las rutas. Como puede verse en este ejemplo para la solución del PAV no necesariamente se obtiene una ruta como solución óptima puede haber mas de una solución óptima con un solo costo mínimo y rutas diferentes. También cabe aclarar que no siempre la solución se encuentra en la primera solución como ocurrió en este ejemplo, en este caso particular se examinaron 13 nodos lo cual es un gran ahorro de búsqueda [ARR96].

Se mencionó anteriormente que la matriz después de reducir su dimensión una y otra vez llegaba a una matriz de (2x2). Entonces se forzaban los últimos pares de nodos para cerrar la ruta correspondiente dado que ninguna otra división del espacio de búsqueda podía realizarse. Así que cuando esto ocurre simplemente se agregan los últimos nodos sin que se puedan seleccionar y se forma el ciclo Hamiltoniano. Esto se debe a que la matriz de (2x2) solo puede tener dos formas como se ve en la tabla 3.9 [ARR96] :

	w	x
u	∞	0
v	0	∞

a)

	w	x
u	0	∞
v	∞	0

b)

Tabla 3.9 Matriz Izquierda de (2x2)

En esta matriz de costos (2x2) los nodos u,v,w, y x generan dos casos, en cualquiera de los estos solamente se requiere una entrada para cerrar el ciclo Hamiltoniano. Véase la fila y columna (1,1) para determinar cuales nodos serán incluidos. Así sí W(1,1) = ∞ agregar los nodos (u,x) y (v,w) que son los nodos que tienen cero, en caso contrario agregar los nodos (u,w) y (v,x) para cerrar la ruta que forma el ciclo Hamiltoniano [ARR96].

3.6.2 Algoritmo del método Una Mejor Ramificación y Acotamiento

Este algoritmo puede dividirse en varios subalgoritmos que resuelvan tres preguntas importantes porque son ellas las que determinan la heurística a seguir [ARR96] :

a) ¿Cómo acotar los pesos de la solución de cada subárbol?
b) ¿Cómo se representa el conjunto de soluciones?
c) ¿Cómo se divide y como se escogen los mejores nodos en el espacio de búsqueda?

Estas preguntas se irán contestando conforme se describan las etapas a seguir.

Primero se acotan las soluciones, es decir se reduce de la matriz W restando una constante de manera que los valores remanentes no sean negativos. Esto cambiará los pesos de cada viaje, pero no el conjunto de viajes legales y sus pesos relativos. Se describe en la figura 3.11 la función Reduce que contesta el inciso a [ARR96].

```
{01} Function Reduce (W)
{02}begin
{03}    sumamenores = 0;
{04}   for i ← 1 to ntamanio do
{05}     begin
{06}        filred(i) = menor de los elementos de la fila i
{07}        if (filred(i)  > 0 )
{08}          begin
{09}            resta filred(i)  de cada elemento de la matriz A en la fila i
{10}            sumamenores = sumamenores + filred(i);
{11}          end del if
{12}     end del for
{13}   for j ← 1 to ntamanio do
{14}     begin
{15}        colred(i) = menor de los elementos de la columna j
{16}        if (colred(i)  > 0 )
{17}          begin
{18}            resta colred(j)  de cada elemento de la matriz W en la columna j
{19}            sumamenores = sumamenores + colred(j);
{20}          end del if
{21}     end del for
{22}   return sumamenores;
{23}end de la function Reduce
```

Figura 3.11 Algoritmo Reduce

Se describirán las líneas mas importantes

-{01} En sumamenores se guardará el valor de la reducción

-{04} ntamanio es el tamaño de la matriz de adyacencia W

-{04} a -{12} Se localiza el menor de cada fila i guardándolo en filred(i), y se resta este menor de cada elemento de la matriz A en la fila i, tantas veces como filas existan. Acumulando en sumamenores todos los menores de todas las filas.

-{13} a –{21}Se realiza la misma operación con las columnas guardando los menores de cada columna en colred(i), acumulando en sumamenores todos los menores de todas las columnas.

La matriz de adyacencia o de costos es usada para representar el conjunto de soluciones, lo que responde el inciso b. Para ramificar o dividir el espacio de soluciones se debe determinar el mejor cero, es decir en que posición (r, c) se encuentra el cero que es capaz de reducir la matriz W cuando se coloca un ∞ de manera que maximice la cantidad a ser restada desde la fila y la columna. La posición del mejor cero determina cuales son los mejores nodos [ARR96]. Esto responde la mitad del inciso c y se lleva a cabo con el siguiente procedimiento de la figura 3.12.

```
{01} Procedure selecciona_los_mejores_nodos ( ntamanio, r, c, a, cotamejornd)
{02}    begin
{03}      cotamejornd  = -∞;
{04}      for i ← 1 to ntamanio do            // fila
{05}        for j ← 1 to ntamanio do           // columna
{06}          if A (i,j) = 0
{07}            begin
{08}              buscar menor_en_fil;
{09}              buscar menor_en_col;
{10}              total = menor_en_fil + menor_en_col;
{11}              if total > cotamejornd
{12}                begin
{13}                  cotamejornd = menor_en_fil + menor_en_col;
{14}                  r ← i;
{15}                  c ← j;
{16}                end del if
{17}            end del if
{18}       A(i,j) - menor_en_ fil;
{19}       A(i,j) - menor_en_col;
{20} end del Procedure selecciona_ mejor_cero
```

Figura 3.12 Algoritmo Selecciona los mejores nodos

Se hará una breve descripción de las líneas más importantes [ARR96] :

- {04} a {09} Para cada fila y para cada columna se localiza el mejor cero. Se localiza un cero en A (i,j) en la fila i, en menor_en_fil se guarda el menor de la fila exceptuando A(i,j) que tiene 0. Es decir se localiza el menor de cada fila diferente de cero. Si hay dos o mas ceros el menor es cero. En la columna j donde se

localizó el cero, en menor_en_col se guarda el menor de la columna exceptuando A(i,j) que tiene 0. Es decir se localiza el menor de cada columna diferente de cero. Si hay dos o mas ceros el menor es cero.

- {10} a {17} Si total que es la suma de menor_en_fil y menor_en_col resulta ser mayor que la cotamejornd, se toma esta suma como la nueva cotamejornd. Guardando además en r el índice de la fila que será uno de los mejores nodos y en c al índice de la columna que será el otro mejor nodo
- {18} Resta a la matriz de costos reducida A(i,j) menor_en_fil de cada elemento de la fila r, excepto a ceros e infinitos.
- {19} Resta a la matriz de costos reducida A(i,j) menor_en_col de cada elemento de la columna c, excepto a ceros e infinitos.

Tanto la función Reduce como el procedimiento selecciona_ mejor_cero son llamados en el procedimiento recursivo llamado ABetterBranchBoundRec que contesta la otra mitad del inciso c, este se describe en la figura 3.13 [ARR96].

```
{01} Procedure ABetterBranchBoundRec (numnds, W, new_reg)
{02}   begin
{03}      cotaactual =  cotaactual + reduce (A);
{04}      if cotaactual < cotainf
{05}         if numnds = n_vertices - 2
{06}           begin
{07}              Los dos últimos nodos son forzados;
{08}              Se guarda la nueva solución;
{09}              cotainf  ← cotaactual;
{09}            end
{10}         else
{11}           begin
{12}              selecciona_mejor_cero(ntamanio, r, c, a, cotamejornd);
{14}              Se prevén y prohiben subciclos
{15}              NewA ← A- columna c - fila r;
{16}              BetterBranchBoundRec(numnds+1, New_A, new_reg);
{17}              restaura A agregando columna c y fila r;
{18}              if cotamasbaja < cotainf
{19}                 begin
{20}                    A(r,c) ← INFINITO;
{21}                   BetterBranchBoundRec(numnds,a,new_reg );
{22}                   a(r,c) ← 0;
{23}              end del if cotamasbaja < cotainf;
{24}           end del if
{25}        end del if numnds = n_vertices - 2
{26}     restaura_matriz_izq de la reducción;
{27} end del Procedure BetterBranchBoundRec
```

Figura 3.13 Algoritmo ABetterBranchBoundRec

Se hará una breve descripción de las líneas más importantes :

- {15} En NewA se almacena la matriz A después de borrar la fila r y la columna c.
- 16} Se genera el subárbol izquierdo al hacer el llamado recursivo al procedimiento BetterBranchBoundRec; con el número de nodos más uno, y la matriz New_A.
- {21} Aquí la segunda recursividad manda a la matriz derecha "a" con el mismo número de nodos.
-{22} Restaura en la matriz "a" los nodos excluidos que fueron en su oportunidad la dirección del mejor cero haciendo a la matriz en la fila r y la columna c igual a cero.

El método Una Mejor Ramificación y Acotamiento es en el peor de los casos básicamente el método de Búsqueda Exhaustiva, así que en ese caso se examinaría todo el árbol para analizar todas las posibles soluciones. Para un problema PAV Asimétrico de n ciudades hay (n-1)! Ciclos Hamiltonianos diferentes; y en el peor de los casos el algoritmo de Una Mejor Ramificación y Acotamiento es de O(n!). En un caso típico sin embargo, la situación no es tan mala. Por ejemplo para el caso particular de 6 ciudades estudiado anteriormente se examinaron solo 13 nodos de los 120 ciclos distintos para este problema de 6 ciudades [(n-1!) = (6-1)! = 120]. Por lo que fueron 13 reducciones , 13 divisiones del espacio de búsqueda, etc., que es mucho menor que 120 = 5! [ARR96]. De hecho el tiempo de ejecución es extremadamente dependiente de las instancias del problema PAV [ARR96]. Se sugiere ver el Capitulo 5, donde se puede apreciar como crece rápidamente el tiempo de ejecución con el tamaño de la matriz de adyacencia (número de ciudades).