UNIVERSITAT POLITÈCNICA DE CATALUNYA

Tesis doctoral:

METALGORITMO DE OPTIMIZACIÓN COMBINATORIA MEDIANTE LA EXPLORACIÓN DE GRAFOS

Director de la tesis:

Dr. D. Albert Corominas Subias

Para optar al Grado de Doctor Ingeniero Industrial, presenta:

Rafael Pastor Moreno

Barcelona, Junio 1999

DATA DE LECTURA 23 de setembre de 1999

A Angie por su paciencia y a mis padres, Victoria y Rafael, por su esfuerzo.

Mi reconocimiento más sincero a mi director de tesis que ha hecho posible su realización. Gracias Albert.

Gracias también a mis compañeros por sus valiosos consejos y ánimos: Don Ramón, Anna, Joaquín y Manel.

ÍNDICE

1. INTRODUCCIÓN. JUSTIFICACIÓN. OBJETIVOS. ALCANCE.

Actualmente, aunque existen procedimientos específicos para resolver algunos problemas concretos de optimización combinatoria, la mayoría se deben solucionar con técnicas generales de exploración del espacio de soluciones, y más concretamente mediante procedimientos de exploración enumerativos en árboles y grafos de búsqueda. El objetivo de esta tesis consiste en formalizar el escenario, confuso y disperso, que en estos momentos engloba a dichos procedimientos: se realiza una amplia recopilación, estudio y análisis de los procedimientos y estrategias de resolución de problemas de optimización combinatoria existentes, y se propone una formalización general de un metalgoritmo de exploración de grafos que engloba a dichos procedimientos, y que, además, incorpora la posibilidad de utilización de nuevas herramientas que se están desarrollando en el campo de la inteligencia artificial (técnicas de consistencia y de propagación local de restricciones) y de la ingeniería informática (paralelismo).

1.1. Introducción y justificación: marco general de los problemas de optimización combinatoria.

A continuación se realiza una breve introducción a los problemas de optimización combinatoria, para exponer de manera informal cuál es el problema que se presenta y justificar la razón de ser de esta tesis. Concretamente, en el apartado 1.1.1. se describe un conocido problema industrial, que resulta ser un problema de optimización combinatoria, y un procedimiento de resolución por enumeración de soluciones factibles; a través de este ejemplo, se pretende mostrar el enorme número de soluciones que se pueden presentar en un problema de optimización combinatoria y varios conceptos que aparecen frecuentemente en los procedimientos enumerativos que se utilizan para su resolución. En el apartado 1.1.2. se presentan diferentes procedimientos enumerativos de resolución de problemas de optimización combinatoria. Finalmente, en el apartado 1.1.3. se introducen las controvertidas y confusas relaciones que existen entre estos procedimientos.

1.1.1. Problemas de optimización combinatoria.

En el momento de plantear el diseño de un sistema productivo, la distribución de la planta es una decisión importante; en general, se puede pensar en dos tipos posibles de diseño: el diseño por proceso (que lleva a una distribución por secciones) y el diseño por producto (que conduce al concepto de línea de producción o montaje).

En un diseño orientado a línea de montaje, una unidad de producto pasa por el puesto de cada operario (estación de trabajo) quien ejecuta las tareas que se le han asignado, y

transcurrido un tiempo prefijado (tiempo de ciclo) un transportador la lleva a la estación siguiente; por tanto, en cada estación se hace el mismo trabajo una y otra vez. Cada operario puede realizar más de una tarea, aunque cada tarea se asigna a un único operario. El problema consiste en asignar el conjunto de tareas a realizar entre las diferentes estaciones de trabajo. Habitualmente, la asignación de tareas debe realizarse atendiendo a una serie de premisas: existen relaciones de precedencias entre tareas, pueden existir estaciones en paralelo, puede ser que las estaciones de trabajo no sean iguales, que existan tareas incompatibles entre sí o forzadas a realizarse en la misma estación, etc.

A continuación se muestra mediante un ejemplo, en qué podría consistir el proceso de resolución de este tipo de problemas. Sea un problema de diseño de líneas de producción de 10 tareas, cuyas duraciones, tareas precedentes inmediatas y grafo de precedencias, se describen en la tabla y grafo siguientes:

Tarea	Duración	Precedentes
A	5	
B	10	A
C	5	A
D	2	B,C
E	7	D
F	5	D
G	10	F
H	2	E
I	5	G,H
J	7	I

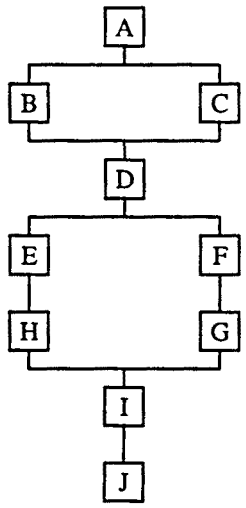


Tabla de datos y grafo, de un ejemplar del problema de diseño de líneas, de Mize et al. (1973).

Además, supóngase que se requiere un tiempo de ciclo de 10 unidades.

El proceso de asignación de las tareas a las estaciones puede resumirse como una secuencia de decisiones entre un conjunto de posibilidades:

* a la primera estación se puede asignar la tarea A y ninguna más por relaciones de precedencia, lo que implica:

Estación	Tareas	Tiempo carga	Tiempo disponible	¿Estación cerrada?
1	A	5	5	No
2				

* la siguiente tarea que se puede asignar es la B (utilizando una nueva estación de trabajo) o la C (en la primera estación o utilizando una nueva):

Estación	Tareas	Tiempo carga	Tiempo disponible	¿Estación cerrada?
1	A	5	5	Sí
2	B	10	0	Sí
3				

Estación	Tareas	Tiempo carga	Tiempo disponible	¿Estación cerrada?
1	A,C	10	0	Sí
2				

Estación	Tareas	Tiempo carga	Tiempo disponible	¿Estación cerrada?
1	A	5	5	Sí
2	C	5	5	No
3				

* supóngase que se selecciona la primera posibilidad, entonces únicamente se puede asignar la tarea C utilizando una nueva estación de trabajo:

Estación	Tareas	Tiempo carga	Tiempo disponible	¿Estación cerrada?
1	A	5	5	Sí
2	B	10	0	Sí
3	C	5	5	No
4				

* a partir de esta situación sólo se puede asignar la tarea D, pero se puede decidir si a la estación 3 o a la 4; de forma similar se adoptarían el resto de decisiones.

Como se puede comprobar, en cada proceso de decisión se selecciona una asignación factible entre las de un conjunto finito, que viene determinado por la secuencia de asignaciones anteriores. Así, si en vez de haber elegido la primera posibilidad en el segundo turno de decisión, se hubiera seleccionado cualquiera de las otras dos, la solución del problema podría haber sido totalmente diferente; aunque tal vez no, ya que las dos asignaciones parciales siguientes conducen a la misma situación: son equivalentes.

Estación	Tareas	Tiempo carga	Tiempo disponible	¿Estación cerrada?
1	A,C	10	0	Sí
2	B	10	0	Sí
3	D,E	9	1	Sí
4	F,H	7	3	No
5				

Estación	Tareas	Tiempo carga	Tiempo disponible	¿Estación cerrada?
1	A,C	10	0	Sí
2	B	10	0	Sí
3	D,E	9	1	Sí
4	H,F	7	3	No
5				

Cada una de las diferentes asignaciones factibles que se pueden realizar constituyen una solución del problema de líneas de montaje planteado. Se dice que un problema con un conjunto de soluciones finito o infinito pero enumerable (imagínese un circuito, en un problema de buscar un camino entre dos vértices de un grafo), es un problema combinatorio.

Usualmente, la asignación de las tareas a las estaciones de trabajo se debe realizar de forma que partiendo de un número de estaciones de trabajo conocido, el tiempo ciclo sea el menor posible, o que partiendo de una tasa de producción (y por tanto de un tiempo ciclo máximo), el número de estaciones de trabajo necesarias sea mínimo. Se dice que un problema combinatorio para el que se busca el valor óptimo (mínimo o máximo) de una función objetivo sobre la región de soluciones factibles, es un problema de optimización combinatoria.

Un concepto que aparece frecuentemente en los problemas de optimización combinatoria es el de cota, que se corresponde con un valor tal que no es posible encontrar ninguna solución factible con un valor mejor de la función objetivo. Volviendo al ejemplo anterior, supóngase que se requiere un tiempo ciclo de 10 unidades y que se desea minimizar el número total de estaciones de trabajo. El tiempo total necesario para realizar una unidad de producto suma 58 unidades de tiempo, por tanto, la cota (en este caso inferior) del número mínimo de estaciones es 6 ($[58/10]^+$); de todas formas, podría no existir ninguna solución con tan sólo 6 estaciones de trabajo.

El problema de líneas de producción o montaje es un problema de optimización combinatoria, pero existen muchos otros problemas industriales que también lo son: diseño de rutas de recogida de personas, basuras, etc., gestión de proyectos con recursos limitados, problemas de carga de camiones y contenedores, diseño de rutas de transporte y distribución, diseño de redes eléctricas y de telecomunicaciones, localización de servicios e instalaciones, secuenciación de diferentes modelos de coches en las líneas de montaje, corte de bobinas para la fabricación de neumáticos, etc.

1.1.2. Procedimientos de resolución enumerativos.

Como se ha comentado, cualquier asignación factible de las tareas a las estaciones constituye una solución del problema de líneas de montaje. En un problema de optimización combinatoria el número de soluciones factibles puede llegar a ser enormemente elevado, y, a pesar de la gran velocidad de cálculo de los ordenadores actuales, no se debe caer en el error de intentar generar y evaluar todas las soluciones y seleccionar la mejor. Sin tener en cuenta las restricciones impuestas por las relaciones de precedencia e incompatibilidades, en una línea formada por n tareas existen n! diferentes órdenes posibles de asignación de éstas: para tan sólo 20 tareas existen 2.43 E+18 posibles asignaciones. Usualmente, los requisitos tecnológicos de precedencias e incompatibilidades reducen considerablemente el número de asignaciones factibles; pero incluso así, la enumeración y evaluación de todas las soluciones excede a la capacidad de los ordenadores. Si existen r precedencias, para analizar todas las posibles soluciones se deben generar aproximadamente $n! / 2^r$ secuencias factibles (Coves 1994), además de un conjunto de soluciones que no son factibles y, por tanto, una vez determinada su infactibilidad no son estudiadas: para 20 tareas y 20 restricciones existen aproximadamente 2.32 E+12 asignaciones factibles. El hecho de que un reducido número de elementos genere un elevado número de soluciones es conocido con el término explosión combinatoria.

En la segunda etapa del proceso de resolución del ejemplar anterior, se ha seleccionado la primera opción a falta de algún criterio. Si el objetivo es utilizar el número mínimo de estaciones, parece más lógico seleccionar la segunda posibilidad descartándose, aunque sea temporalmente, las otras dos. Para ello se puede asociar una función de evaluación a cada opción, que descarte las poco prometedoras y se concentre en las que parece que conducen a situaciones consideradas mejores. Un ejemplo de función de evaluación en el problema de líneas de montaje es $e = n + t / TC$, donde n es el número de estaciones ya utilizadas, t es el tiempo ocupado de la estación que está siendo completada y TC es el tiempo de ciclo; en este caso, se trata de seleccionar la opción que presenta el menor valor de dicha función de evaluación.

La representación más clara de la evolución del proceso de resolución de los problemas de optimización combinatoria, y sobre todo para el uso de este tipo de funciones, es mediante un árbol o grafo de búsqueda. A continuación se recogen, aunque con ciertos matices, las definiciones que Scholl (1986) y Brunat (1996) realizan sobre una serie de términos relacionados con dicha forma de representación; éstas constituyen la terminología básica que se utiliza en esta tesis:

* Grafo: conjunto de elementos que se denominan vértices y una lista de aristas que unen parejas de dichos vértices.

* Grafo simplemente conexo: grafo en el que existe una cadena (secuencia de aristas) entre cualquier pareja de vértices.

* Árbol: grafo conexo acíclico.

* Vértice hijo: vértice sucesor (descendiente inmediato) de otro.

* Vértice padre: vértice precedente (ascendente inmediato) de otro u otros.

* Vértices hermanos: vértices hijos del mismo vértice padre.

* Semigrado exterior de un vértice: número de sus vértices hijos.

* Semigrado interior de un vértice: número de sus vértices padres.

* Vértice no terminal: vértice de semigrado exterior no nulo.

* Vértice terminal u hoja: vértice de semigrado exterior nulo.

* Anchura de un árbol: número máximo de vértices en un mismo nivel.

* Profundidad de un árbol: nivel máximo de los vértices del árbol.

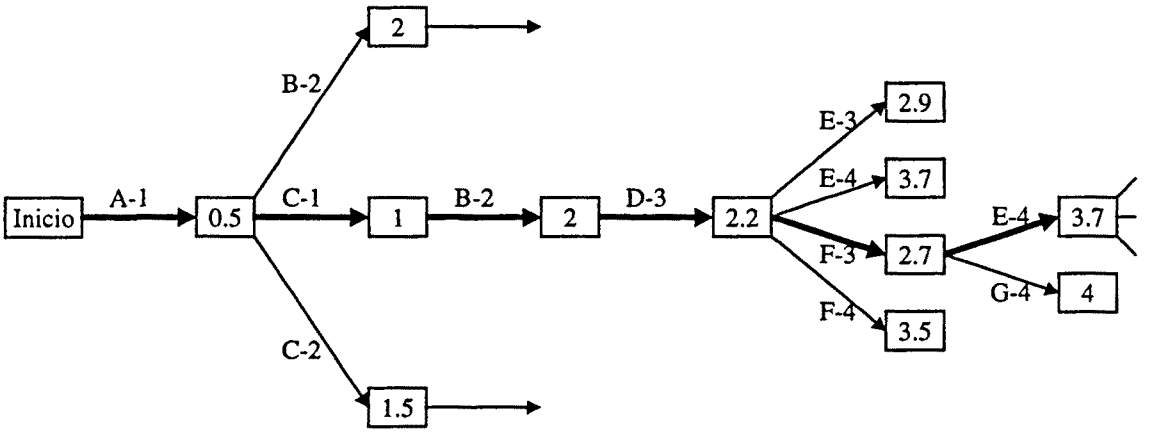
* Arborescencia: grafo simplemente conexo con un único vértice con semigrado interior nulo (denominado vértice raíz) y donde todos los demás tienen semigrado interior igual a uno.

* Arborescencia binaria: arborescencia en la que el semigrado exterior de todos los vértice es, como máximo, igual a dos.

* Nivel (profundidad) de un vértice: número de vértices del camino que va de la raíz hasta ese vértice.

Aunque en ciertas terminologías se distingue claramente entre árbol y arborescencia, existen otras en las que únicamente se utiliza el término árbol. En este texto se utilizan indistintamente los términos árbol y arborescencia, aunque siempre en referencia a un grafo orientado.

El árbol de búsqueda asociado al ejemplar del problema de líneas de montaje anterior es el que se muestra en el grafo siguiente, donde a cada arco se asocia la asignación tarea-estación realizada, y en el interior de cada vértice se indica el valor de la función de evaluación e = n + t / TC; para dirigir la exploración se utiliza el criterio de seleccionar el vértice de cada etapa, de menor valor de e:



Árbol de búsqueda asociado al ejemplar del problema de diseño de líneas.

Los procedimientos de resolución de problemas de optimización combinatoria que se basan en explorar el árbol o grafo con el que se puede representar el espacio de estados del problema, se denominan procedimientos de resolución enumerativos. La idea principal de este tipo de procedimientos consiste en partir (ramificar) sucesivamente el espacio de soluciones posibles en subconjuntos más pequeños y resolver el problema sobre cada subconjunto. En el árbol de enumeración, los problemas resultantes son llamados subproblemas o vértices. Si fuera posible construir todo el árbol, se podría garantizar la optimalidad de la solución obtenida, pero como se ha mostrado, es inviable para problemas de dimensiones interesantes. Para reducir el número de subproblemas a resolver, se calculan cotas solucionando problemas relajados de los definidos en los vértices; si la cota de la solución de un subproblema es peor o igual que la mejor solución disponible, el subproblema (la rama del árbol) es eliminado, es podado.

Como ya se ha comentado, la representación más clara de la evolución del proceso de resolución de los problemas de optimización combinatoria es mediante árboles o grafos

de búsqueda. Ambas representaciones se pueden clasificar atendiendo a cómo el procedimiento de partición (ramificación) divide el espacio de soluciones:

* árboles y grafos OR: aquéllos en los que el proceso de ramificación divide el espacio de soluciones en subconjuntos, de forma que la solución del problema en cualquiera de los subconjuntos proporciona una solución del problema original.

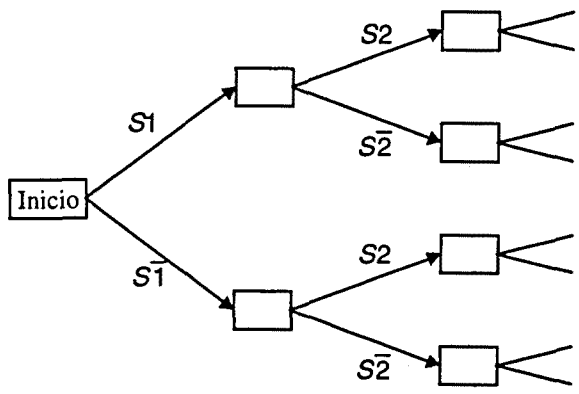
* árboles y grafos AND/OR: aquéllos en los que el proceso de ramificación divide el espacio de soluciones en subconjuntos, de forma que la solución del problema original requiere la resolución de los subproblemas de dos o más subconjuntos y la combinación (suma, resta, etc.) de dichas soluciones.

A modo de ejemplo, sea un problema de cubrimiento que consiste en seleccionar el número mínimo de puestos de servicios necesarios para cubrir una serie de zonas, a partir de la matriz que indica qué zonas cubre cada puesto de servicio:

	Servicio1	**Servicio2**	**Servicio3**	**Servicio4**	**Servicio5**	**Servicio6**
Zona1	X	X				
Zona2	X					
Zona3				X	X	
Zona4			X		X	
Zona5				X	X	
Zona6						X

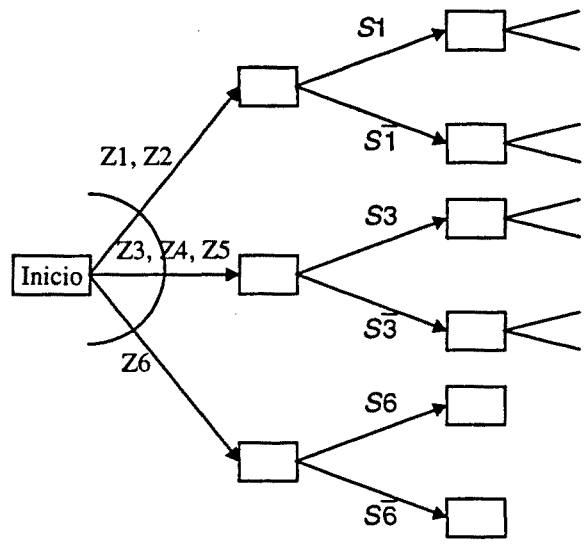
Matriz de datos de un ejemplar del problema de cubrimiento.

El espacio de estados del problema se puede representar mediante un árbol OR, en el que la ramificación se basa en seleccionar o no, un puesto de servicio. De esta manera se puede obtener una solución del problema, resolviendo de forma factible el subproblema asociado a cualquiera de los vértices del árbol.



Árbol OR del ejemplar del problema de cubrimiento.

Por otro lado, este mismo ejemplar también admite una representación mediante un árbol AND/OR: se divide la zona a cubrir en tres subzonas totalmente independientes entre sí (zonas Z1 y Z2, zonas Z3, Z4 y Z5, y zona Z6), se resuelve un subproblema en cada una de ellas (mediante una representación en árbol OR), y se combinan las soluciones obtenidas para obtener la solución del problema original. Las ramas AND se representan uniéndolas mediante un semicírculo, como se muestra en la siguiente figura.



Árbol AND/OR del ejemplar del problema de cubrimiento.

Una vez introducido el funcionamiento general de los procedimientos de resolución enumerativos, a continuación se exponen brevemente unos conjuntos de técnicas de este tipo, que son estudiadas y sintetizadas posteriormente en el presente trabajo.

1.1.2.1. Procedimientos basados en la "fuerza bruta".

Los procedimientos cuya herramienta de exploración principal consiste en el uso de la "fuerza bruta", son técnicas enumerativas de búsqueda en las que el orden en el que se explora para encontrar la solución sólo depende de la posición de los vértices en el árbol de búsqueda.

Esta clase de procedimientos son considerados impracticables para problemas no triviales (Barr & Feigenbaum 1981), al explorar inútilmente una gran región del espacio de soluciones factibles y producirse, como ocurre en el problema de líneas de montaje, la temida explosión combinatoria. Las diferentes técnicas de este tipo existentes difieren en el orden en el que se exploran los vértices, y entre éstas cabe destacar la búsqueda en profundidad (*depth first search*), la búsqueda en anchura (*breadth first search*), la búsqueda primero el de menor coste (*uniform cost search* o *cheapest first strategy*), la búsqueda primero el mejor (*best first search*), etc.

1.1.2.2. Procedimientos que utilizan "búsqueda heurística".

Los procedimientos enumerativos que incorporan "búsqueda heurística" son aquéllos en los que el orden en el que se explora para encontrar la solución está guiado mediante un indicador; de esta forma, se selecciona el siguiente vértice a explorar basándose en información heurística que se dispone de cada uno de los vértices candidatos.

Este tipo de procedimientos son los que habitualmente se utilizan para resolver problemas de dimensiones industriales.

1.1.2.3. Programación dinámica.

Como ya se ha comentado en el problema de líneas de montaje, a partir de varios vértices diferentes de un nivel se puede alcanzar un mismo vértice del nivel siguiente. La idea principal que se pone en práctica en los procedimientos basados en programación dinámica, consiste en explorar el árbol de búsqueda eliminando los vértices redundantes y conservando únicamente el mejor de todos ellos: agrupando los vértices equivalentes.

1.1.2.4. *Branch and bound.*

El término *branch and bound* se refiere a una familia de procedimientos de resolución enumerativos, cuya principal característica consiste en que dedican un gran esfuerzo a reducir el número de vértices a explorar, calculando una cota del valor de una función objetivo asociada a los subproblemas: si la cota de un vértice es peor o igual que el valor de una solución factible (o en particular, peor que la mejor solución conocida, que se suele denominar preferible), el vértice es eliminado. Como comentan Gass & Harris (1996), las técnicas *branch and bound* son frecuentemente utilizadas para solucionar problemas de programación entera.

1.1.2.5. Otros procedimientos guiados y del área de la inteligencia artificial.

Además de las técnicas descritas en los apartados anteriores, existen algunos otros procedimientos de resolución también desarrollados en el área de la investigación operativa: *branch and price*, *branch and cut*, *branch and reduce*, *bounded dynamic programming*, etc.

En el área de la inteligencia artificial también se han diseñado, o al menos formalizado, muchos otros procedimientos de exploración enumerativos que utilizan diversas técnicas para dirigir la búsqueda, hacerla lo más breve posible e incluso convertirla en un procedimiento heurístico. Entre éstos cabe destacar sobre todo el A*, que consiste en

un procedimiento semejante al *branch and bound* pero con una forma especifica de selección del próximo vértice a explorar. Otros procedimientos desarrollados en el área de la inteligencia artificial son el procedimiento A, el IDA*, el WA*, el WIDA*, el *depth first iterative deepening*, el *depth-m search*, etc.

1.1.3. Relaciones entre los diferentes procedimientos de resolución.

La mayoría de procedimientos de búsqueda enumerativos han sido utilizados tanto en el área de la investigación operativa como en el de la inteligencia artificial. En numerosas ocasiones, algunos de éstos han sido bautizados con diferentes nombres aún tratándose de la misma técnica (así por ejemplo, el algoritmo de búsqueda en profundidad o búsqueda primero en profundidad ha sido denominado hasta de otras seis formas diferentes: *linear search*, *single branch search*, estrategia de búsqueda LIFO, búsqueda hacia el fondo, búsqueda vertical y estrategia en profundidad), y algunos otros presentan conexiones tan profundas entre ellos que podrían considerarse redundantes: en Barr & Feigenbaum (1981) y Kumar & Kanal (1983b), se comenta que el *branch and bound* (B&B), la programación dinámica (PD) y el A* son muy parecidos, aunque el B&B y la PD se han utilizado en investigación operativa y el A* en inteligencia artificial. Además, las relaciones entre ellos son controvertidas.

Por ejemplo, Nilsson (1980), Mompín et al. (1987), Shirai & Tsujii (1987) y Companys (1989a) definen el algoritmo A como un algoritmo A* que no asegura la optimalidad de la solución, mientras que muchos otros autores (Barr & Feigenbaum 1981, Korf 1990, Raphael 1990, Cortés et al. 1993, Ginsberg 1993 y Rich & Knight 1994) consideran que sólo existe el algoritmo A*, y que asegura o no la optimalidad de la solución en función del cumplimiento de la propiedad de admisibilidad. Además, en este caso también existen definiciones particulares del A*: Cortés et al. (1993), además de considerar los costes de los caminos en número de etapas, asumen la agrupación de estados equivalentes y la conservación únicamente del mejor camino hasta éstos (como también hacen Rich & Knight 1994); por su parte, Pearl (1984) considera el algoritmo A* como una especialización del Z* (técnica especial de búsqueda primero el mejor que obtiene una solución óptima, descrita por él mismo). Como comentan diversos autores (Pearl 1984, Mompín et al. 1987, Companys 1989a, Cortés et al. 1993 y Greenberg 1996 entre otros), si no se dispone de información heurística y el coste asociado a un vértice se define como su profundidad, el algoritmo A o el A* correspondiente equivale a la exploración en anchura. Si el coste asociado se define como menos la profundidad del vértice, el algoritmo A* equivale a la búsqueda en profundidad (Pearl 1984). Para Nilsson (1971), Barr & Feigenbaum (1981) y Pearl (1984) si no se dispone de información heurística, el algoritmo A* es el procedimiento de búsqueda primero el de menor coste; por su parte, Rich & Knight (1994) especifican un mayor número de posibilidades. Además, Korf (1990) considera que el algoritmo A* es un procedimiento

de búsqueda primero el mejor. Según Greenberg (1996), otro caso particular del A* en investigación operativa es el *branch and bound* para programación entera, en el que la función de evaluación toma el valor objetivo de la relajación lineal en el vértice (sin comentar la característica principal del B&B: el podado mediante cotas).

Barr & Feigenbaum (1981) denominan *ordered state space search* a la búsqueda primero el mejor (*best first search*) y especifican que el *breadth first*, el *uniform cost* y el *depth first* son casos especiales de ésta; por otro lado, estos mismos autores, Nilsson (1971) y Pearl (1984) definen específicamente la búsqueda primero el de menor coste (*uniform cost search* o *cheapest first strategy*), mientras que la mayoría la consideran un caso especial de la búsqueda primero el mejor; por su parte, Pearl (1984) diferencia entre si sólo se busca una solución factible (estrategia de búsqueda primero el mejor) o si se busca una solución óptima (algoritmo que Pearl denomina búsqueda primero el mejor*); Mompín et al. (1993) definen el procedimiento de búsqueda primero los mejores (*best few*), como una nueva técnica que consiste en explorar un conjunto de alternativas en paralelo; además, se han definido técnicas que alternan búsquedas en profundidad con búsquedas primero el mejor (Ibaraki 1988). Cortés et al. (1993) llaman óptimo por niveles a la búsqueda primero el mejor, B&B a lo que tal vez se podría denominar ramificación sin acotación y B&B con subestimación a lo que la mayoría conoce como B&B; por otro lado, consideran que el A* es el procedimiento más general.

Según citan Kumar & Kanal (1983b), mientras que Pohl argumenta que los procedimientos de búsqueda heurística son muy diferentes de los *branch and bound*, Hall e Ibaraki dicen que muchas técnicas de búsqueda heurística en representaciones en espacio de estados, son esencialmente B&B. Según este mismo trabajo, Martelli & Montanari afirman que su algoritmo de búsqueda heurística es diferente del B&B, ya que "*(B&B) technique does not recognize the existence of identical subproblems*"; por otro lado, Kumar & Kanal (1983b) comentan que el B&B de Ibaraki sí reconoce la existencia de subproblemas idénticos. También exponen que mientras Nilsson describe el procedimiento A*, considera que la programación dinámica (PD) es esencialmente un procedimiento de búsqueda en anchura; por otro lado, dicen que Morin & Marsten permiten una computación de PD con cotas, y para ellos esto indica que Morin & Marsten no consideran que la PD sea necesariamente una búsqueda en anchura.

Las relaciones entre B&B y PD también han sido controvertidas. Ibaraki (1988) discute cómo un conjunto de procedimientos de PD pueden ser vistos en una estructura de B&B, aunque también comenta que los B&B pueden verse como un procedimiento de PD al que se le adiciona un test de acotado. Marsten & Morin (1978) y Dyer et al. (1995) describen un algoritmo híbrido entre la PD y el B&B, y lo consideran como un procedimiento de PD más un test de acotado, o como un B&B más una poda por

dominancias. Por su parte, Bautista et al. (1992, 1994) exponen un nuevo procedimiento híbrido, la programación dinámica acotada, que también aprovecha las características de ambas técnicas. Según Kumar & Kanal (1983b), el trabajo de Ibaraki parece implicar que, para problemas de optimización combinatoria, la PD es un esquema de resolución de problemas más general que el B&B, pero en Ibaraki (1988) se describe un B&B muy general que incorpora diversos test de podado: cota, dominancias, infactibilidades, etc.

De todas formas, también cabe comentar la existencia de algunos autores que parecen tener más clara la conexión entre algunos de estos procedimientos: "*A class of algorithms similar to A* is used in operations research under the name of branch-and-bound algorithms*", Barr & Feigenbaum (1981), p. 64.

1.2. Objetivos y alcance.

El panorama descrito en la introducción, muestra un escenario de los procedimientos de exploración enumerativos en árboles y grafos de búsqueda, poco estructurado, disperso e insuficientemente formalizado. Considerando que aunque existen técnicas específicas para resolver algunos problemas combinatorios concretos, la mayoría deben solucionarse con procedimientos generales de exploración del espacio de soluciones, la dispersión, confusión e incluso duplicidad que se ha dado en los últimos años entre los procedimientos desarrollados en el área de la investigación operativa y en el de la inteligencia artificial (e incluso entre los diseñados en el seno de cada una de ambas disciplinas), dificulta en gran medida la selección de un procedimiento para resolver uno de los diferentes problemas de optimización combinatoria existentes.

Por otro lado, actualmente se están incorporando nuevos procedimientos (*branch & reduce*, *branch & cut*, *branch & price*, *branch and cut and price*, etc.), que en el fondo se pueden reducir a esquemas conocidos, así como nuevas técnicas y herramientas (técnicas de consistencia y de propagación local de restricciones, y paralelismo), lo que puede acelerar el desarrollo de nuevos procedimientos, nuevas denominaciones o nuevos híbridos de los ya existentes, complicando todavía más la situación actual.

Partiendo del supuesto de representar el proceso de resolución de los problemas de optimización combinatoria mediante un grafo de estados, los objetivos y aportaciones de esta tesis son los siguientes:

- Recopilación, análisis y crítica de diferentes procedimientos y estrategias de resolución existentes, así como de las formulaciones generales de dichas técnicas expuestas en la literatura.

- Propuesta, formalización y desarrollo de un metalgoritmo de exploración de grafos que englobe a los diversos procedimientos de búsqueda (*branch and bound*, programación dinámica, A*, búsqueda en profundidad, búsqueda en anchura, IDA*, etc.), y que, además, incorpore la posibilidad de utilizar las nuevas herramientas que se están desarrollando en el campo de la inteligencia artificial (técnicas de consistencia y de propagación local de restricciones) y de la ingeniería informática (paralelismo).

- Diseño y proposición de nuevos procedimientos híbridos, resultado de la combinación de los ya existentes.

A continuación se describen las principales hipótesis de trabajo bajo las que se desarrolla la presente tesis:

- Los procedimientos de búsqueda analizados presentan suficientes características comunes, como para poder desarrollar un metalgoritmo de exploración de grafos que englobe las diversas técnicas de exploración expuestas. La formalización de dicho metalgoritmo es el objeto central de la tesis.

- Los valores de todos los datos de los ejemplares de los problemas son deterministas y estáticos: los datos se conocen de partida y no se trabaja con problemas de solución on-line en un entorno cambiante, como podrían ser los juegos y los sistemas autónomos en tiempo real.

- Únicamente se consideran procedimientos unidireccionales y no se abordan algoritmos bidireccionales como el *perimeter search* con A* (PS*) y con IDA* (IDPS*) (Dillenburg & Nelson 1994), el *depth first iterative deepening* (DFID) en búsqueda bidireccional (Korf 1985), el BIDA* (Manzini 1995), etc.

- Por otro lado, sólo se utilizan representaciones del espacio de estados a través de árboles o grafos OR; no son consideradas las representaciones mediante árboles o grafos AND/OR, ni, por tanto, los procedimientos enumerativos diseñados para su resolución.

La adopción de estas hipótesis de trabajo presenta las siguientes implicaciones. Por un lado no se resuelven problemas combinatorios con datos aleatorios (excepto que se trabaje con valores estimados), ni aquéllos que deben resolverse dentro de un entorno dinámico. Y, por otro, tampoco se formalizan los procedimientos bidireccionales ni los basados en representaciones AND/OR (cabe destacar, sin embargo, que en este caso no se restringe de ninguna manera los problemas combinatorios que es posible resolver).

2. PROBLEMAS DE OPTIMIZACIÓN COMBINATORIA.

2.1. Definición.

Lawler (1996) define la combinatoria como la rama de la matemática que trata de ordenar objetos usualmente finitos en número y sujetos a varias restricciones. Por otro lado, Gass & Harris (1996) definen la optimización como el proceso de búsqueda del mejor valor que puede llegar a ser obtenido (en programación matemática, éste es el valor mínimo o máximo de la función objetivo sobre la región de soluciones factibles).

Ibaraki (1988) proporciona una definición formal de optimización combinatoria, que también es denominada optimización discreta, programación combinatoria o programación discreta.

Ibaraki define un problema de optimización como:

$$P: \text{MIN (MAX)} \; f(x)$$
$$x \in S,$$

donde X es el espacio de soluciones y $S \subset X$ es la región factible dentro del espacio X (en el conocido problema de la mochila, X está formado por los diferentes conjuntos de objetos posibles y S por los conjuntos de objetos cuyo peso total no supera cierto valor). Como se ha descrito, S es el conjunto de soluciones factibles, y, por otro lado, la función $f: S \rightarrow R$ (o a veces N) es la función objetivo.

Una solución $x \in S$ es óptima, si no existe otra solución $y \in S$ tal que $f(y) < f(x)$ si se minimiza o $f(y) > f(x)$ si se maximiza.

Un problema de optimización es de optimización combinatoria, si X y S son combinatorios o discretos, es decir, si son conjuntos de un número finito de elementos o de infinitos elementos numerables. Por tanto, se puede decir que un procedimiento de resolución de problemas de optimización combinatoria trata de encontrar una solución, que optimice (minimizando o maximizando) una función objetivo sobre un espacio combinatorio o discreto.

La definición proporcionada por Ibaraki (1988) no es suficientemente general, ya que uno de los problemas de optimización combinatoria más importantes, la programación lineal mixta, no se ajusta a dicha formulación debido a que los espacios de soluciones X y S no son numerables. Corominas (1996) propone una definición de problema de optimización combinatoria general, en la que se formaliza que una vez definido el valor

de las variables discretas, se dispone de un procedimiento para resolver óptimamente el resto del problema:

$$[\text{OPT}]\ z = f(x, y)$$
$$(x, y) \in S,$$

donde x representa a un vector de variables discretas, y a un vector de variables continuas, y dada una x´ se puede obtener una y´ que proporciona la solución óptima del problema para x´.

A veces no se necesita la solución óptima o se presenta la obligación de alcanzar el óptimo de forma aproximada (por limitaciones en el tiempo de cálculo, etc.); además, se desea que la diferencia con la solución óptima no sea muy grande, pero eso sí, que esté acotada. Corominas & Companys (1977) proponen una nueva definición que recoge estos aspectos:

$$[\delta\text{-OPT}]\ z = f(x)$$
$$x \in S \subseteq X,$$

donde:

x es una solución,

S es el conjunto de soluciones factibles,

K es el conjunto de números totalmente ordenado (reales, enteros, ...),

f: S $\rightarrow$ K, es una correspondencia que a cada x $\in$ S le hace corresponder un número.

Existe K´ totalmente ordenado y existe g, tal que:

g: K x K´ $\rightarrow$ K (K pueden ser los enteros y K´ los reales, de forma que a cada pareja entero-real se le asigna un número entero),

es de la forma:

$$g(u,\eta) \geq u \qquad \forall\, u \in K, \qquad \forall\, \eta \in K',$$

y además:

$$u' \geq u \Rightarrow g(u',\eta) \geq g(u,\eta) \qquad \forall\, u, u' \in K, \qquad \forall\, \eta \in K',$$
$$\eta' \geq \eta \Rightarrow g(u,\eta') \geq g(u,\eta) \qquad \forall\, u \in K, \qquad \forall\, \eta, \eta' \in K';$$

es decir, g incrementa el valor de u en η o en un valor relacionado con η ($g(u,\eta)$ puede ser: $u + \eta$, $u \cdot (1+\eta)$, $|u \cdot (1+\eta)|$, etc.).

Una vez definido g, se puede definir δ-minimizar como encontrar $x^* \in S$ tal que $f(x^*) \leq g(f(x),\delta)$, $\forall\ x \in S$; es decir, buscar una solución que, en el caso de que $g(u,\eta) = u+\eta$, esté como mucho δ unidades lejos del óptimo (puede ser no óptima en, como máximo, δ unidades).

2.2. Ejemplos.

Existe una gran cantidad de problemas industriales que son problemas de optimización combinatoria: problemas de inversiones, equilibrado o diseño de líneas de producción o montaje, diseño de rutas de recogida de personas, basuras, etc., gestión de proyectos con recursos limitados, problemas de carga de camiones y contenedores, diseño de rutas de transporte y distribución, diseño de redes eléctricas y de telecomunicaciones, localización de servicios e instalaciones, secuenciación de diferentes modelos de coches en las líneas de montaje, corte de bobinas para la fabricación de neumáticos, programación de los movimientos de un robot o un puente grúa, organización de horarios y turnos de trabajo, diseño de un almacén, localización de productos en un almacén, etc.

Todos estos problemas pueden formalizarse de una forma más estructurada y abstracta en una serie de problemas de optimización combinatoria, algunos de los cuales enumeran y describen Corominas et al. (1984), Ibaraki (1988), Bjorndal et al. (1995), Corominas (1996), Gass & Harris (1996) y Wang & Coleman (1996), aunque existen muchos otros.

* Problemas de flujos: dado un grafo orientado, G = (V, A), con un vértice origen, α, un vértice destino, ω, y con tres números enteros asociados a cada arco (l_{ij}, u_{ij}, c_{ij}) (límites inferior y superior del flujo y coste unitario), se trata de hallar un flujo compatible de coste mínimo. Existen muchos problemas que son casos particulares de éste: problema del transporte, del transporte con capacidades, de afectación, de acoplamiento (*matching problem*), de flujo máximo, del camino mínimo entre un par de vértices, etc.

* Problema del cartero chino (*Chinese postman problem*: CPP): dado un grafo no orientado, G = (V, E), con unos valores c_{ij} asociados a cada arista, se trata de hallar un ciclo de coste mínimo que pase al menos una vez por cada arista. También se puede plantear en un grafo orientado.

* Problema del viajante de comercio (*travelling salesperson problem*: TSP): dado un grafo completo no orientado, G = (V, E), con unos valores c_{ij} asociados a cada arista, se trata de hallar un ciclo hamiltoniano (que pase exactamente una sola vez por cada vértice) de coste mínimo. También se puede plantear en un grafo orientado.

* Problema general de itinerarios (*general routing problem*: GRP): sea un grafo G = (V, A, E), es decir con arcos y aristas, con valores c_{ij} asociados a cada arco o arista, y sean $V' \subseteq V$, $A' \subseteq A$, $E' \subseteq E$; se trata de hallar un itinerario que contenga todos los elementos de V', A' y E', en el que todos los arcos de A' tengan la misma orientación y con un coste mínimo (este caso es el *single vehicle routing problem*).

* Problema de afectación generalizado (*generalized assignment problem*: GAP): dadas m máquinas (i = 1, 2, ..., m) y n trabajos indivisibles (j = 1, 2, ..., n), en el sentido de que cada trabajo se debe realizar íntegramente en una sola máquina, se trata de afectar los trabajos a las máquinas teniendo en cuenta que cada máquina tiene un tiempo disponible b_i, que realizar j en i representa un coste c_{ij} y consume un tiempo t_{ij}, y con coste total mínimo. Es decir, encontrar una asignación de los n trabajos a las m máquinas de mínimo coste, tal que todo trabajo se asigne a una máquina y se respeten sus restricciones de capacidad. El programa lineal binario asociado es el siguiente:

$$
\begin{aligned}
[\text{MIN}]\ z = {} & \sum_{1 \le i \le m,\ 1 \le j \le n} c_{ij} \cdot x_{ij} & \\
& \sum_{1 \le j \le n} t_{ij} \cdot x_{ij} \le b_i & \forall\, i \\
& \sum_{1 \le i \le m} x_{ij} = 1 & \forall\, j \\
& x_{ij} \in \{0, 1\}. &
\end{aligned}
$$

* Problemas de *scheduling*: problemas en los que se trata de hallar una secuencia de trabajos y a veces sus instantes de inicio y finalización (un programa), considerando que dichos trabajos necesitan unos recursos usualmente limitados, y optimizando algún criterio como el tiempo total, el coste total, etc.

* Problema de localización de plantas: sea un grafo G, orientado o no, en el que los vértices (o algunos vértices) representan centros de demanda y los arcos o aristas un sistema de comunicaciones; se trata de localizar uno o diversos puestos que proporcionen servicio a los centros de demanda, optimizando alguna función en la que intervienen usualmente las distancias (de todas formas existen problemas como el de cubrimiento, en los que las distancias no participan en la función objetivo al estar incluidas en las restricciones).

* Problema de cubrimiento (*set-covering problem*): problema de programación entera definido como sigue:

$$[\text{MIN}]\ z = c \cdot x$$
$$E \cdot x \geq e,$$

donde los componentes de E son o unos o ceros, los componentes del vector columna e son todos unos, y las variables están restringidas a ser uno o cero. La idea del problema consiste en encontrar, a mínimo coste, un conjunto de columnas de E, tales que los unos del vector e estén "cubiertos" por al menos uno de los unos de las columnas seleccionadas. Notar que la cobertura múltiple está permitida.

* Problema de partición (*set-partitioning problem*): problema de programación entera definido como sigue:

$$[\text{MIN}]\ z = c \cdot x$$
$$E \cdot x = e,$$

donde los componentes de E son o unos o ceros, los componentes del vector columna e son todos unos, y las variables están restringidas a ser uno o cero. Problema similar al de cubrimiento excepto en que la cobertura múltiple no está permitida.

* Problema de empaquetado (*set-packing problem*): problema de programación entera definido como sigue:

$$[\text{MIN}]\ z = c \cdot x$$
$$E \cdot x \leq e,$$

donde los componentes de E son o unos o ceros, los componentes del vector columna e son todos unos, y las variables están restringidas a ser uno o cero. La idea del problema consiste en encontrar, a mínimo coste, un conjunto de columnas de E, tales que los unos del vector e estén "cubiertos" por como máximo uno de los unos de las columnas seleccionadas.

* Problema de *bin-packing*: consiste en la determinación del número mínimo de cajas necesarias para empaquetar un conjunto dado de elementos. El problema clásico en una dimensión se define como sigue: dada una caja de capacidad C y una lista de elementos $L = (p_1, p_2, ..., p_n)$, donde p_i tiene tamaño $s(p_i)$ satisfaciendo $0 \leq s(p_i) \leq C$, determinar el menor número entero m tal que existe una partición $L = B_1 \cup B_2 \cup ... \cup B_m$ que

satisface $\sum s(p_i) \leq C$, donde $p_i \in B_j$, $1 \leq j \leq m$. El conjunto B_j usualmente se asocia al contenido de una caja de capacidad C.

* Coloreado de los vértices de un grafo: dado un grafo G, se trata de determinar el número mínimo de colores necesarios para pintar cada vértice del grafo de un color, de manera que dos vértices adyacentes no tengan igual color.

* Problema de la mochila (*knapsack problem*: KP): dados n elementos u objetos de valor $c_j > 0$ y peso $a_j > 0$, y un valor b (límite de peso total), se trata de hallar un conjunto formado con dichos elementos cuyo peso total no supere b, de valor total máximo:

$$[MAX]\ z = \sum_{1 \leq j \leq n} c_j \cdot x_j$$
$$\sum_{1 \leq j \leq n} a_j \cdot x_j \leq b$$
$$x_j \in \{0, 1\} \text{ o entero no negativo, } \forall\ j.$$

Este problema ha sido extensamente estudiado por su aplicación inmediata y porque surge como relajación de diferentes problemas de programación entera.

* Problema de programación entera mixta (*mixed integer programming problem*: MIP): problema de programación matemática en el que las restricciones y la función objetivo son lineales, pero varias variables están restringidas a tener valores enteros. Las variables enteras pueden ser binarias o tomar cualquier valor entero. Estos problemas tienen gran importancia dentro de los problemas de optimización combinatoria, ya que muchos de éstos pueden ser formulados como problemas de programación entera mixta.

2.3. Complejidad de los problemas de optimización combinatoria.

Existen algunos problemas de optimización combinatoria que disponen de algoritmos eficientes que los resuelven, en cambio otros no; esto es debido a que son de diferente complejidad. La teoría de la complejidad se presenta de forma excelente en Garey & Johnson (1979) y en Papadimitriou & Steiglitz (1982). Aquí únicamente se relata una breve introducción a varios conceptos generales, pero antes de comenzar a exponerlos conviene recordar algunas definiciones previas (Hall 1996):

- Problema: descripción abstracta (o estructura de datos para otros autores) asociada a una pregunta que requiere una respuesta (en el problema TSP y dado un grafo con sus costes asociados, ¿cuál es el ciclo hamiltoniano de menor coste?).

- Ejemplar de un problema: un ejemplar de un problema incluye la especificación exacta de los datos (el grafo contiene 6 vértices, el arco (1, 2) de coste 8, el (3, 4) de coste 10, etc.); por tanto, un problema puede presentar un número infinito de ejemplares.

- Algoritmo: un algoritmo para un problema es un conjunto de instrucciones que garantiza encontrar la solución de cualquier ejemplar en un número finito de pasos.

- Paso en un algoritmo: en un ordenador consiste en una de las siguientes operaciones: suma, resta, multiplicación, división en precisión finita o comparación de dos números; la valoración del número de pasos en un algoritmo se expresa como una función del tamaño del ejemplar correspondiente.

- Tamaño de un ejemplar: es el número de bits necesarios para codificarlo; para medirlo se requieren las dimensiones del ejemplar (en el TSP, el número de vértices y arcos del grafo), que se denominan dimensiones inherentes, y el número máximo de bits necesarios para codificar cualquier dato (los arcos y sus costes).

- Notación de la O grande: sean dos funciones f(t) y g(t) de un parámetro t no negativo, se dice que $f(t) = O(g(t))$ si existe una constante $c > 0$ tal que, para todo t suficientemente grande, $f(t) \leq c \cdot g(t)$; la función $c \cdot g(t)$ es una cota asintótica superior de f. Papadimitriou & Steiglitz (1982), Nemhauser & Wolsey (1988) y Scholl (1995) proporcionan definiciones equivalentes.

Hall (1996) define el termino complejidad computacional, señalando que tiene dos usos que se deben distinguir claramente:

a) Por un lado, se refiere a un algoritmo para resolver ejemplares de un problema. Para Papadimitriou & Steiglitz (1982), la forma más comúnmente aceptada de medir la ejecución de un algoritmo es mediante el tiempo necesario para alcanzar la respuesta final. Este tiempo puede variar en función del ordenador utilizado, y para evitarlo, se expresa en términos del número de pasos necesarios para ejecutar el algoritmo en un ordenador hipotético (ordenador que ejecuta instrucciones secuencialmente y como mucho una en cada instante de tiempo), y se asume que cada paso consume una unidad de tiempo independientemente del tamaño del ejemplar. Además, el número de pasos que necesita un algoritmo no es el mismo para todos los inputs; así, se consideran todos los inputs del mismo tamaño y se define la complejidad del algoritmo en función del tamaño de la entrada y para el caso peor.

b) Por otro lado, se refiere al problema en sí mismo. La teoría de la complejidad computacional clasifica los problemas de acuerdo a su inherente tratabilidad o intratabilidad (esto es, si son fáciles o difíciles de resolver); este esquema de

clasificación incluye las clases P, NP, NP-completo y NP-duro (que posteriormente se definen).

El número de operaciones de los algoritmos puede ser diferente para ejemplares de un mismo problema y del mismo tamaño. Para Ahuja et al. (1989) existen tres procedimientos básicos para medir la ejecución de un algoritmo:

1. Análisis empírico: mide el tiempo de computación de un algoritmo utilizando una distribución de ejemplares del problema. El principal objetivo del análisis empírico consiste en estimar cómo se comporta el algoritmo en la práctica.

2. Análisis del caso peor: tiende a buscar cotas superiores del número de pasos que un algoritmo puede necesitar en cualquier ejemplar de un problema, es, por tanto, la máxima complejidad de un ejemplar de tamaño n. Este tipo de análisis tiende a ser pesimista ya que el caso peor sólo se presenta en muy pocos ejemplares. Gass & Harris (1996) proporcionan una definición similar: para un algoritmo y su problema asociado, consiste en la determinación de una cota superior del número de pasos que el algoritmo puede necesitar para cualquier ejemplar del problema.

3. Análisis del caso promedio: consiste en estimar el número de pasos promedio que necesita un algoritmo, supuesta una distribución de probabilidad de los ejemplares de un problema. Se diferencia del análisis empírico en que el análisis del caso promedio proporciona rigurosas propiedades matemáticas de la ejecución, más que estimaciones estadísticas; es por tanto la media de la complejidad de todos los ejemplares de tamaño n, y, por otro lado, presenta el inconveniente de que usualmente es difícil conocer la distribución de probabilidad de la complejidad de los ejemplares.

En general, los problemas de optimización combinatoria se clasifican según la complejidad computacional del caso peor, ya que como exponen Nemhauser & Wolsey (1988):

a) se garantiza absolutamente el tiempo de ejecución,
b) es independiente de la distribución de probabilidad de los ejemplares,
c) y parece más fácil de medir para analizar.

De todas formas, y como argumentan estos mismos autores y Bjorndal et al. (1995), este análisis no siempre refleja la tratabilidad computacional real: en muchos casos existe una gran diferencia entre la complejidad del caso peor y el tiempo y/o espacio necesario promedio para resolver ejemplares del problema. Así, parece imprescindible diferenciar el punto de vista teórico y la realidad, ya que, en el entorno organizativo en el que se engloba dentro de un contexto industrial concreto, un algoritmo debería ser considerado

bueno o malo según la relación existente entre el tiempo que tarda en resolver el problema planteado y el tiempo disponible.

Papadimitriou & Steiglitz (1982), Ibaraki (1988), Nemhauser & Wolsey (1988), Bondy (1995), Scholl (1995), Gass & Harris (1996) y Hall (1996) definen de manera similar los conceptos de algoritmo de tiempo polinomial, exponencial, fuertemente polinomial y pseudopolinomial, aunque Bondy (1995) sólo lo hace en referencia a los grafos. Después de introducir dichos conceptos, y como aclaración de los mismos, se muestra la exposición que realizan sobre el tema Ahuja et al. (1989) tomando como referencia un problema en redes.

Para discutir si un problema es fácil o difícil, primero se debe precisar dónde está la frontera entre los problemas fáciles y los difíciles. Comúnmente se dice que los problemas fáciles, los que se resuelven eficaz y prácticamente, son aquellos que se solucionan con un algoritmo con complejidad del tiempo $O(n^k)$, donde n es el tamaño del ejemplar (el número de bits necesarios para codificarlo) y k es una constante. Dicho algoritmo es de orden polinomial y se llama algoritmo de tiempo polinomial.

Por otro lado, los problemas difíciles son aquellos para los que no se dispone de un algoritmo acotado polinomialmente en n para su resolución (por ejemplo: $O(n^{\log(n)})$, $O(k^n)$). Son los denominados algoritmos de tiempo exponencial y su complejidad está acotada inferiormente por una función exponencial en n. De todas maneras, se podría argumentar que un algoritmo $O(c \cdot n^k)$ es peor que uno $O(k^n)$ si las constantes c y k son grandes y n es pequeña. Algunos autores como Ibaraki (1988) opinan que, aunque lo anterior es cierto, las definiciones se enfocan a destacar el comportamiento asintótico de un algoritmo $O(k^n)$ en función de n; sin embargo, Papadimitriou & Steiglitz (1982) exponen que existe una gran controversia sobre este tema que pone en duda la tesis de que tiempo polinomial sea sinónimo de práctico.

Aunque la distinción más importante se presenta entre algoritmos de tiempo polinomial y exponencial, también se han definido otras clases de algoritmos.

Por un lado se definen los algoritmos de tiempo fuertemente polinomial, como aquellos cuyo tiempo de ejecución está acotado polinomialmente por una función que sólo depende de las dimensiones inherentes del problema (en el TSP, el tamaño del grafo asociado) y es independiente del tamaño de los datos numéricos del ejemplar (número de bits necesarios para codificarlo).

Y por otro lado existen los algoritmos de tiempo pseudopolinomial, que son aquellos que se ejecutan en tiempo polinomial de la dimensión del problema y de las magnitudes de los datos implicados, más que en logaritmo en base dos de sus magnitudes; tales

algoritmos son técnicamente funciones exponenciales del tamaño de los datos de entrada, y por tanto, no se consideran polinomiales. Como dicen Garey & Johnson (1979), un algoritmo de tiempo pseudopolinomial mostrará comportamiento exponencial, sólo cuando se enfrente con ejemplares que contengan números exponencialmente grandes.

A continuación, se transcribe la exposición que realizan Ahuja et al. (1989) sobre el tema tomando como referencia un problema en redes. Las definiciones siguientes se muestran para aclarar los términos anteriores, y sobretodo el concepto de algoritmo de tiempo fuertemente polinomial y el de algoritmo de tiempo pseudopolinomial.

El tiempo de ejecución de un algoritmo en redes se acota en función de varios parámetros básicos del problema: el número de vértices (n), el número de arcos (m) y las cotas superiores del coste de los coeficientes y de las capacidades de los arcos (C y U, que se asume que toman valores enteros), y se determina contando el número de pasos realizados en la ejecución. Se asume que tanto C como U están polinomialmente acotados en n para varias constantes k, $C = O(n^k)$ y $U = O(n^k)$. Esta hipótesis, conocida como la hipótesis de similitud, es muy razonable en la práctica (por ejemplo, si los costes están restringidos a ser menores que $100 \cdot n^3$, se pueden tener costes de hasta 100.000.000.000 para redes con 1.000 vértices).

Se dice que un algoritmo es de tiempo polinomial, si su tiempo de ejecución está acotado por una función polinomial de la longitud de la entrada (número de bits necesarios para representarla), que para un problema de redes es una función de orden polinomial de n, m, $\log_2 C$ y $\log_2 U$ (por ejemplo: $O((n + m) \cdot (\log_2 n + \log_2 C + \log_2 U))$, $O(n^2 \cdot m)$, $O(n \cdot \log_2 n)$, etc.). Consecuentemente, se dice que un algoritmo de redes es de tiempo polinomial, si su tiempo de ejecución está acotado por una función polinomial de n, m, $\log_2 C$ y $\log_2 U$.

Se dice que un algoritmo es de tiempo fuertemente polinomial, si su tiempo de ejecución está acotado por una función polinomial solamente de n y m, pero no de $\log_2 C$ o $\log_2 U$. El interés de los algoritmos fuertemente polinomiales es principalmente teórico; en particular, si se evoca a la hipótesis de similitud, todo algoritmo de tiempo polinomial es de tiempo fuertemente polinomial, ya que $\log_2 C \approx O(\log_2 n)$ y $\log_2 U \approx O(\log_2 n)$.

Se dice que un algoritmo es de tiempo exponencial, si su tiempo de ejecución crece como una función que no puede ser polinomialmente acotada, por ejemplo $O(n \cdot C)$, $O(2^n)$, $O(n!)$ y $O(n^{\log n})$. (Obsérvese que $n \cdot C$ no puede estar acotado por una función polinomial de n y de $\log_2 C$).

Se dice que un algoritmo es de tiempo pseudopolinomial, si su tiempo de ejecución está polinomialmente acotado por n, m, C y U -por ejemplo: $O(m + n \cdot C)$ y $O(m \cdot C)$-. Los algoritmos pseudopolinomiales constituyen la subclase más importantes dentro de los algoritmos exponenciales. Para los problemas que satisfacen la hipótesis de similitud, los algoritmos de tiempo pseudopolinomial son algoritmos de tiempo polinomial, pero los algoritmos no son atractivos si C y U son de grado polinomial grande en n.

Para Ahuja et al. (1989), existen dos importantes razones para preferir los algoritmos polinomiales a los exponenciales:

- cualquier algoritmo polinomial es asintóticamente superior a cualquier algoritmo exponencial, incluso en casos extremos: $n^{1.000}$ es menor que $n^{0,1 \cdot \log n}$ si n es suficientemente grande (en este ejemplo n debe ser mayor que $2^{10.000}$, pero como se expone a continuación estos casos no son frecuentes),

- muchas experiencias prácticas han mostrado que, como regla, los algoritmos de tiempo polinomial se ejecutan mejor que los algoritmos de tiempo exponencial; además, en la práctica los algoritmos polinomiales son de grado pequeño.

Probar que un problema es fácil, es simple: es suficiente encontrar un algoritmo con complejidad de tiempo polinomial que lo resuelva; sin embargo para probar que un problema es difícil, no es suficiente con decir que no se ha encontrado ningún algoritmo con complejidad de tiempo polinomial, se debería probar que no es posible concebir ningún algoritmo que lo resuelva en tiempo polinomial. Según Bondy (1995), actualmente existen muchos problemas para los que todavía no se ha encontrado un algoritmo de tiempo polinomial, y podrían no existir.

Como comentan diferentes autores (Papadimitriou & Steiglitz 1982, Ibaraki 1988, Nemhauser & Wolsey 1989, Bondy 1995, Scholl 1995, Hall 1996, etc.), un problema de optimización combinatoria tiene un problema de reconocimiento asociado (también llamado de decisión o factibilidad), que es un problema cuya pregunta a solucionar requiere la respuesta "sí" o "no" (en el TSP, ¿el grafo contiene un ciclo hamiltoniano de longitud menor o igual que k?). Este problema no es más difícil de resolver que el problema de optimización asociado, y además, los resultados negativos probados para los problemas de reconocimiento son aplicables a los problemas de optimización.

Por otro lado, el problema de reconocimiento y el problema original de optimización son equivalentes, en el sentido de que el algoritmo de reconocimiento se puede encajar en una búsqueda binaria para resolver el problema de optimización, con un número polinomial de llamadas a éste (un procedimiento de bisección podría ser el más efectivo). Consecuentemente, si un problema de reconocimiento requiere $O(g(n))$ el

problema de optimización requiere O(K · g(n)), siendo K el número máximo de problemas de reconocimiento que se deben resolver para solucionar el problema de optimización asociado; de manera similar, si el problema de optimización puede ser resuelto en tiempo O(g(n)), el problema de reconocimiento se resuelve en el mismo orden de tiempo O(g(n)) -Ibaraki (1988)-. Por tanto, ambos problemas son equivalentes en el sentido del orden de magnitud necesario para su resolución.

Papadimitriou & Steiglitz (1982), Ibaraki (1988), Nemhauser & Wolsey (1988, 1989), Bondy (1995), Scholl (1995) y Hall (1996) proporcionan definiciones similares de los términos: problema P, NP, NP-completo, NP-duro, fuertemente NP-completo y débilmente NP-completo. A continuación se introducen dichos conceptos y se remite a Garey & Johnson (1979) para un estudio más profundo.

En el momento de trabajar con complejidad de problemas, conviene distinguir claramente entre computación determinista y no determinista. La computación determinista es la que se realiza en un ordenador hipotético, entendido éste como el ordenador que ejecuta instrucciones secuencialmente y como mucho una en cada instante de tiempo. Mientras, la computación no determinista se refiere a un concepto similar al de computación paralela pero sin restringir el número de procesos que se ejecutan a la vez, además, cada camino de computación es independiente de los resultados de los otros (se obtiene el mismo efecto que si se realizaran computaciones deterministas para resolver cada uno un problema de reconocimiento, trabajando en paralelo y sin interaccionar entre ellas).

Se dice que un problema de reconocimiento y de optimización es de clase P (tiempo polinomial), si en una computación determinista existe un algoritmo de tiempo polinomial que lo resuelve. La clase P comprende aquellos problemas para los que existen algoritmos eficientes y que usualmente son considerados fáciles.

Se dice que un problema de reconocimiento es de clase NP (no determinista polinomial), si dispone de un algoritmo que en una computación no determinista responde sí en tiempo polinomial, si el problema tiene respuesta sí. Dicho de otra manera, los problemas NP se caracterizan por poder ser resueltos por enumeración, y por obtener una respuesta afirmativa, si existe, en tiempo polinomial de una computación determinista.

Por definición, P es una subclase de NP ($P \subseteq NP$).

Sean dos problemas de reconocimiento A y B, tales que, todo ejemplar p de A de tamaño n puede ser transformado en un ejemplar q de B en tiempo determinista

polinomial, y donde p y q satisfacen la condición que p tiene solución afirmativa si y sólo si la tiene q. Entonces se dice que A es reducible (polinomialmente) a B.

En este caso, cualquier ejemplar de A puede ser resuelto transformándolo en un ejemplar de B y aplicando un algoritmo para B. Ibaraki (1988) sugiere la relación: (complejidad de A) ≤ (complejidad de transformación de A a B) + (complejidad de B), y que la complejidad de A no es mayor que la de B si la complejidad de transformación es despreciable (lo que ocurre, cuando la complejidad de la transformación es de tiempo polinomial y la de B no lo es). Otra forma más compacta de expresar la relación de complejidad anterior, tal vez sería: (complejidad de A) ≤ MAX{ (complejidad de transformación de A a B), (complejidad de B) }.

Un problema de reconocimiento B se denomina NP-duro, si cualquier problema A de clase NP es reducible a B. Si un problema NP-duro pertenece a los NP, se llama NP-completo (término introducido por Cook en 1971). Es decir, un problema es NP-completo cuando es NP y todos los demás problemas NP pueden reducirse a él. Si un problema es NP-duro, es al menos tan difícil de resolver como cualquiera de los problemas de reconocimiento NP-completos. Por otro lado, el término NP-duro se puede extender a los problemas de optimización combinatoria, de forma que un problema de optimización es NP-duro si su versión de reconocimiento es NP-completo (esto es debido a que resolver el problema de optimización es al menos tan duro como resolver el problema de reconocimiento).

De las definiciones anteriores se desprende que los problemas NP-completos son la intersección de las clases de problemas NP y NP-duros. Por definición, los problemas NP-completos son la clase que contiene los problemas más difíciles dentro de los problemas NP. Por otro lado, estos problemas presentan varias propiedades:

a) Dos problemas A y B NP-completos satisfacen que A es reducible a B y que B es reducible a A.
b) Si al menos un problema NP-completo fuera resoluble en tiempo polinomial en una computación determinista, entonces todo problema NP sería resoluble en tiempo polinomial.

Si lo enunciado en la propiedad b se cumple, entonces P = NP; actualmente se cree y se asume que P ≠ NP, sin embargo, no se ha podido probar ni lo uno ni lo otro y ésta es la cuestión fundamental todavía no resuelta en la ciencia de la computación teórica.

Se debe tener presente que el concepto NP-completo está basado en la complejidad del caso peor, y en varios problemas ocurre que, aun existiendo ejemplares de muy alta complejidad, éstos pueden ser eficientemente resueltos en el sentido de la complejidad

media (por ejemplo, el problema de la mochila). De todas formas, y aun desde el punto de vista de la complejidad media, los problemas NP-completos son más difíciles que los problemas NP: el tamaño máximo de los problemas NP-completos que pueden ser prácticamente tratados es mucho menor que los problemas NP, aun con más sofisticados algoritmos. Por tanto, el concepto de NP-completo también es utilizado para medir la complejidad desde un punto de vista práctico (Ibaraki 1988).

Aunque todavía no se ha descubierto un algoritmo en tiempo polinomial para un problema NP-completo, algunos disponen de algoritmos pseudopolinomiales que los resuelven. Se dice que un problema es fuertemente NP-completo, si no puede tener un algoritmo pseudopolinomial que lo resuelva; un problema se dice que es débilmente NP-completo, si puede disponer de algoritmos pseudopolinomiales que lo resuelvan (por ejemplo, el problema de la mochila 0-1). Ibaraki (1988) comenta que empíricamente los problemas débilmente NP-completos tienden a ser más fáciles de manejar mediante procedimientos enumerativos.

Como expone Ibaraki (1988), en la década de los 80 la teoría de la complejidad computacional ha revelado la existencia de una jerarquía infinita que contiene a las clases P y NP, entre sus miembros. Sean $f_1(n)$ y $f_2(n)$ funciones crecientes tales que $f_2(n)$ es asintóticamente más grande que $f_1(n)$ (por ejemplo: $f_1(n) = 1.000 \cdot n$ y $f_2(n) = 2^n$). Parece obvio que la clase de problemas resolubles en tiempo determinista $f_1(n)$, está contenida en la clase de problemas resolubles en tiempo determinista $f_2(n)$. En otras palabras, una secuencia de funciones:

$$\log n, n, n^2, n^3, \ldots, 2^n, 2^{n^2}, \ldots, 2^{2^n}, \ldots, 2^{2^{2^n}}, \ldots$$

define una secuencia infinita de clases de complejidad.

Como matiza Ibaraki (1988), la clase P viene dada por $P = \lim_{k\to\infty}$ clase (n^k); por otro lado parece que $NP \subset \lim_{k\to\infty}$ clase (2^{n^k}), sin embargo, la localización exacta de NP en la secuencia anterior todavía no se conoce exactamente y tampoco se puede descartar totalmente que P = NP.

Según Hall (1996), también se han definido clases para algoritmos paralelos y para algoritmos aleatorios. De todas formas, cabe recordar que la pregunta más trascendental todavía no resuelta es: ¿P = NP?

A continuación se ofrece una lista de diversos problemas P y NP-completos, citados por Even (1979), Papadimitriou & Steiglitz (1982), Corominas et al. (1984) y Nemhauser & Wolsey (1988)[1].

Son problemas P, la resolución de ecuaciones lineales (*solving linear equations*), la programación lineal (*the linear programming problem*), la conexidad del grafo (*graph connectedness*), el camino en un grafo orientado (*path in a digraph*), etc., y las versiones de reconocimiento de los problemas de camino de mínimo peso con datos no negativos (*the minimum-weight path problem with nonnegative data*), camino de mínimo peso (*the minimum-weight path problem*), transporte (*the transportation problem*), árbol parcial mínimo (*minimum spanning tree*), máximo acoplamiento (*maximum matching*), etc.

Son problemas NP-completos, el problema de camino hamiltoniano (*the directed hamilton path problem*), circuito hamiltoniano (*the directed hamilton circuit problem*), cadena hamiltoniana (*the hamilton path problem*), ciclo hamiltoniano (*the hamilton circuit problem*), condición de satisfactibilidad (*satisfiability condition*), 3-coloración (*the 3-coloration problem*), etc., y las versiones de reconocimiento de los problemas de afectación generalizado (*the generalized assignment problem*), programación de secuencias en procesadores en paralelo (*multiprocessor scheduling*), viajante de comercio (the *travelling salesperson problem*), empaquetado (*the set packing problem*), mochila 0-1 (*the 0-1 knapsack problem*), 2-partición (*the 2-partition problem*), cubrimiento de vértices mínimo (*the minimum vertex cover problem*), número cromático (*the chromatic number*), corte máximo (*the maximum cut problem*), etc.

2.4. Representación de los problemas de optimización combinatoria.

El objetivo de la exposición siguiente consiste en identificar y especificar la forma en la que se representan los problemas de optimización combinatoria a lo largo del presente trabajo. Con tal fin se realiza un desarrollo argumental que, aunque a primera vista puede resultar algo abstracto, se considera de gran utilidad en el momento de argumentar la representación que se ha seleccionado: en forma de grafo o árbol de estados OR.

Para Cortés et al. (1993) solucionar un problema es partir de la salida y, recorriendo un camino, andar hasta la llegada; es una forma de construir caminos y andar por ellos. La solución de un problema es el camino, el espacio de soluciones posibles es el problema y la representación del espacio es la representación del problema. El espacio está configurado por sus posibles resultados, por los lugares de partida y por los diferentes

[1] Ibaraki (1988) ofrece una lista más amplia, clasificándolos en problemas de grafos, redes, scheduling y otros.

caminos que se pueden trazar y recorrer. Como se puede comprobar, esta definición encaja a la perfección con una representación de estados en forma de árbol o grafo OR.

Según Salkin (1975), Mompín et al. (1987), Ibaraki (1988), Bjorndal et al. (1995), Grötschel & Lovász (1995), etc., la optimización combinatoria estudia problemas caracterizados por tener un número finito de soluciones factibles (caminos para Cortés et al. 1993), por tanto, se podría pensar en hallar la solución óptima por enumeración. Esta opción es en la práctica usualmente imposible ya que se produce la temida explosión combinatoria, que como ya se ha comentado para el problema de líneas de montaje y Gass & Harris (1996) definen, es el fenómeno asociado a los problemas de optimización cuya dificultad computacional aumenta exponencialmente con el tamaño del problema.

Una vez está claro que para problemas de dimensiones interesantes no es posible enumerar todas las soluciones factibles y seleccionar la mejor, se debe pensar en cómo resolver los problemas de optimización combinatoria. Para Cortés et al. (1993) y Rich & Knight (1994), el proceso de construcción de sistemas capaces de resolver problemas, y en particular problemas de optimización combinatoria, consta de los siguientes pasos:

1. Definir y describir el problema con la mayor precisión posible: estados iniciales, finales e intermedios, y operaciones de transformación entre estados.
2. Analizar y evaluar el problema, a través de características que pueden ser importantes: número de soluciones requeridas, óptimas o no, descomposición del problema en subproblemas, ...
3. Identificar, aislar y representar el conocimiento necesario para resolver el problema: de los estados y de los operadores posibles.
4. Escoger la mejor técnica de solución de problemas y aplicársela.

La definición y descripción precisa del problema implica el diseño de una representación del mismo que lo modelice de forma que sea tratable sistemáticamente. Winston (1994a) presenta una enumeración de procedimientos generales de resolución de problemas, que puede ayudar a definir una correspondencia entre representaciones y procedimientos de resolución:

a) Método de generación y prueba: consiste en generar soluciones factibles y evaluarlas, aceptándolas o rechazándolas (como se ha comentado este procedimiento es habitualmente impracticable).

b) Método de análisis de medios y metas: basándose en el espacio de estados y conociendo el estado inicial, el estado meta y la diferencia entre ambos, el objetivo

consiste en reducir las diferencias aplicando los operadores posibles. Para ello la idea clave es explorar un árbol o grafo OR.

c) Método de reducción del problema: procedimiento que parte de conocer las metas y convertirlas en submetas más fáciles de alcanzar (por ejemplo, $\int(f(x)+g(x))dx$ se puede sustituir por $\int f(x)dx$ por un lado y $\int g(x)dx$ por otro, y sumar el resultado). Para ello la idea clave es explorar un árbol o grafo de metas, un árbol o grafo AND/OR.

Como especifica Nilsson (1971), la división anterior no es estricta. El método de reducción de problemas consiste en hacer un análisis del problema original para encontrar un conjunto de subproblemas, tales que, la resolución de varios subconjuntos de subproblemas implique la resolución del problema original; cada conjunto de subproblemas puede ser abordado por métodos de espacio de estados (denominados por Winston (1994a) de análisis de medios y metas) o puede ser analizado para producir nuevos subproblemas. Por tanto, estrictamente hablando la aproximación de espacio de estados puede considerarse como un caso de reducción de problemas. En la presente tesis únicamente se consideran aproximaciones en el espacio de estados, descartándose la división del problema original en problemas independientes de menor tamaño o realizándose dicha división antes de utilizar el procedimiento de resolución enumerativo.

Tanto para la representación en espacio de estados como la de reducción de problemas, se suelen utilizar grafos y árboles, obteniéndose los denominados OR si se utiliza una representación mediante el espacio de estados, y los AND/OR si se trabaja con representaciones de reducción de problemas. Como señalan Mompín et al. (1987) y Shirai & Tsujii (1987), el espacio de estados se puede representar gráficamente por medio de un grafo, donde cada vértice representa un estado del sistema y los enlaces entre ellos representan la acción de un operador para cambiar de un estado a otro; cuando los estados duplicados se representan por un sólo vértice, ya no se habla de árboles sino de grafos.

Al trabajar con espacios de estados (como se hace de aquí en adelante) y para disponer de una representación completa, se debe especificar claramente tres cosas (Nilsson 1971):

- la descripción de los estados y en particular la del estado inicial,
- el conjunto de operadores y su efecto en los estados descritos,
- las propiedades de un estado objetivo. En este punto existe cierta controversia ya que, mientras algunos investigadores en inteligencia artificial consideran disponer del estado objetivo (sobretodo en juegos), en investigación operativa usualmente

no se conoce, aunque sí alguna propiedad que lo describe: propiedades que debe o que no debe cumplir.

Una vez se diseña una representación del problema en espacio de estados, el problema de encontrar un estado que cumpla una condición de objetivo puede ser formulado como el problema de encontrar un vértice en el grafo asociado cuyo estado asociado cumpla la condición de objetivo. Como exponen Nilsson (1971), Shirai & Tsujii (1987) y Bautista et al. (1992), si un problema puede ser representado en el espacio de estados, solucionarlo se reduce a un proceso de búsqueda de un camino entre dos vértices específicos de dicho espacio representado por un grafo. Mishkoff (1988) denomina al proceso de examen de las posibles soluciones alternativas, búsqueda; al conjunto de posibles caminos de exploración, espacio de búsqueda; y a la forma de representar gráficamente el espacio de búsqueda, árbol de búsqueda.

Estos mismos autores junto a Companys (1989a), aunque es ampliamente aceptado, consideran que si se asocian unos costes a los arcos del grafo usualmente no sólo se busca el camino que lleva del vértice inicial al vértice objetivo, sino que de entre todos los posibles se busca el camino óptimo, el de menor coste asociado (de todas formas existen excepciones: Pastor (1994) describe una aplicación industrial en la que más que optimizar simplemente se buscan soluciones factibles). Por otro lado, al ser la optimalidad muy costosa en tiempo y espacio, ésta tiende a debilitarse, y, como comentan diferentes autores (Cortés et al. 1993, Riera et al. 1995 y Corominas 1996), el esfuerzo de muchos investigadores se orienta hacia el diseño y análisis de algoritmos aproximados que proporcionen soluciones factibles.

Otro aspecto a destacar es el hecho de que hay que distinguir entre el grafo/árbol posible de búsqueda (llamado espacio de estados o espacio de búsqueda) y el grafo/árbol construido en el procedimiento de búsqueda (llamado grafo/árbol de búsqueda); uno es el grafo/árbol implícito y el otro el grafo/árbol explícito. Claramente el espacio de búsqueda puede ser infinito o, aun finito, muy extenso; por tanto, y debido a que un grafo puede ser especificado explícita o implícitamente, el problema de buscar soluciones se convierte en el de hacer explícita la mínima parte posible del grafo implícito que contiene el estado deseado.

La dimensión del problema no es el único factor que condiciona su resolución, el diseño de su representación tiene una gran influencia en el esfuerzo de búsqueda necesario para resolverlo: es fundamental seleccionar la formulación más fuerte, más compacta, y obviamente se prefieren las representaciones con espacios de estados pequeños. El proceso requerido para el diseño de la representación es poco conocido; según Mompín et al. (1987) y Nemhauser & Wolsey (1988), es un arte más que una metodología formal, ya que no existen teorías y normas generales que ayuden a elegir correctamente, y así

depende de la experiencia que permite aprovechar las estructuras y características propias del problema (simplificar nociones como las simetrías, etc.). Winston (1994a) expone el principio de la representación según el cual, una vez que el problema se describe mediante una buena representación está casi resuelto, y describe un conjunto de características de las buenas representaciones. Por su lado, Shirai & Tsujii (1987) argumentan que la representación adecuada de un problema simplifica su compresión.

Actualmente, la comunidad científica internacional tiene muy presente la importancia de la representación de los problemas; así, según Hoffman & Padberg (1996), se están realizando muchos esfuerzos en la reformulación de problemas de programación entera, considerando a veces ventajoso el hecho de incrementar el número de variables, el de restricciones o ambos a la vez -Alonso & Escudero (1995, 1997, 1998) resuelven un problema de planificación del tráfico aéreo mediante dos formulaciones equivalentes: la segunda necesita un mayor número de variables y restricciones, pero muchas de éstas definen facetas de la región factible; como consecuencia, es frecuente que en la solución continua relajada del problema se obtengan muchas variables con valor entero, y, por consiguiente, el proceso de resolución es mucho menor y más rápido-. Además y como comentan Johnson et al. (1997), es posible controlar el tamaño del grafo explícito proporcionando una buena formulación inicial.

La mayoría de investigadores consideran que antes de solucionar un problema se pueden aplicar sencillos test que permiten reformularlo compactando la solución, al transformar, añadir y/o eliminar variables y/o restricciones -Johnson et al. (1997) proponen la disgregación de restricciones y la reducción de coeficientes-; estas técnicas son conocidas usualmente como procedimientos de preproceso y se tratan en mayor profundidad en el apartado 3.8.1.

Debido a los adelantos en el desarrollo de ordenadores multiprocesadores capaces de implementar en paralelo algoritmos de optimización, éste es otro aspecto a tener en cuenta en el momento de seleccionar una formulación (Meyer 1989). Desde el punto de vista de la modelización, es importante formular el problema de forma que pueda ser descompuesto en zonas de exploración casi independientes que se adapten a la arquitectura del ordenador donde será resuelto. Los aspectos claves son la granularidad de la computación, es decir, el tamaño de las piezas que son ejecutadas en paralelo, y, por otro lado, la cantidad de información que se intercambia entre los procesadores.

3. PROCEDIMIENTOS DE RESOLUCIÓN ENUMERATIVOS.

Como ya se ha comentado, la gran dificultad que presentan los problemas combinatorios reside precisamente en la combinatoria existente en la obtención de soluciones, que hace que el número de soluciones, aunque finito, sea demasiado elevado como para poder evaluarlas todas y quedarse con la mejor. A pesar de la existencia de procedimientos específicos que resuelven algunos problemas combinatorios concretos, la mayoría de ellos deben solucionarse con técnicas generales de exploración del espacio de soluciones factibles. A este tipo de problemas es posible asociarles un grafo OR cuyos vértices corresponden, o a conjuntos de soluciones, o a estados que muestran la evolución y construcción de las soluciones en curso. Basándose en esta idea existen diversos procedimientos de exploración del grafo que se puede asociar al problema, aunque todos van explorando, o en general enumerando, diferentes estados del grafo asociado; debido a esta característica común se les suele denominar procedimientos de resolución enumerativos.

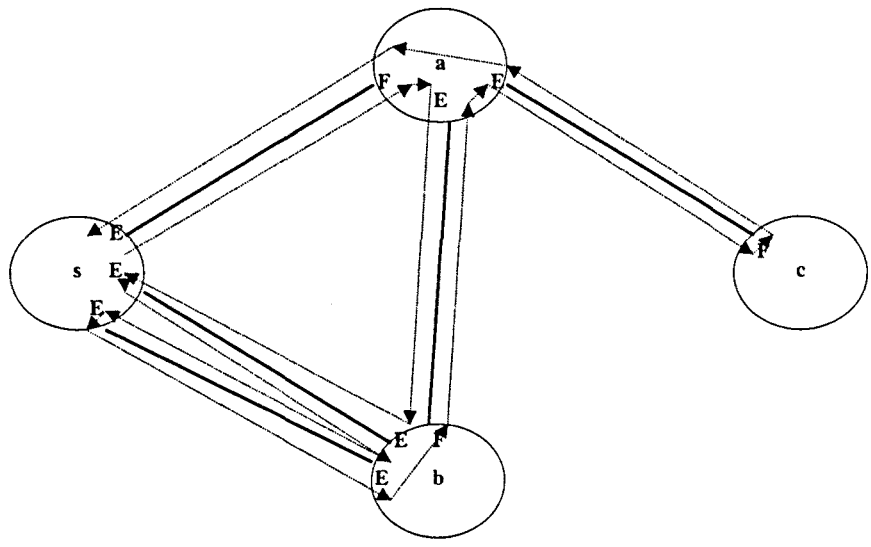
Según señala Balcázar (1993), el esquema general en el que se basan todos estos procedimientos enumerativos es conocido mediante la denominación genérica "esquema divide y vencerás". Su descripción intuitiva es simple: los datos se descomponen en partes aproximadamente iguales, se resuelve recursivamente el mismo problema sobre estas partes, y se combinan las soluciones parciales para obtener la solución del problema global (Nilsson 1971, Nemhauser & Wolsey 1989, Balcázar 1993, Brunat 1996, etc.). En contrapartida a esta aparente sencillez, tanto la forma de descomponer los datos como la de combinar los resultados pueden ser relativamente complejas. Por otro lado, este esquema se basa en explorar un árbol o grafo AND/OR, pero como se expone en las hipótesis de trabajo, en esta tesis únicamente se consideran representaciones mediante árboles o grafos OR, descartándose la descomposición del problema original en problemas independientes de menor tamaño, o realizándose dicha división antes de utilizar el procedimiento de resolución enumerativo considerado.

Desde hace más de 30 años, en la literatura proveniente del área de la investigación operativa existen referencias que abordan tanto los problemas de optimización combinatoria, como los procedimientos enumerativos que se han ido desarrollando para su resolución. De todas formas, la mayoría de estas referencias exponen procedimientos de forma aislada y son escasas aquéllas que presentan una formulación común que engloba a más de un procedimiento. Además, cabe destacar que algunos de estos procedimientos no son tan recientes. Por ejemplo, desde el siglo XIX se conoce un antecesor de lo que hoy se denomina búsqueda en profundidad (*depth first search*) como una técnica para recorrer laberintos: como cita Even (1979), en 1.882 E. Lucas, en *Récreations Mathématiques*, relata el trabajo de Trémaux como sigue.

Disponiendo de un grafo conexo y comenzando en uno de los vértices, se desea caminar a través de todos los arcos, de vértice a vértice, de forma que se visiten todos los vértices, se recorran todos los arcos y se finalice en el mismo vértice donde se comenzó. Claramente se necesita ir marcando los lugares por donde se va pasando, y para ello se utilizan dos tipos de marcas: F para la salida del arco por el que se visita por primera vez el vértice y E para cualquier entrada en un arco cuando se usa para abandonar el vértice; por tanto, un arco tiene una entrada y una salida, que se marcan por separado. Además ninguna marca es cambiada o eliminada.

Sea s el vértice inicial y v el vértice actual, el algoritmo de Trémaux es el siguiente:

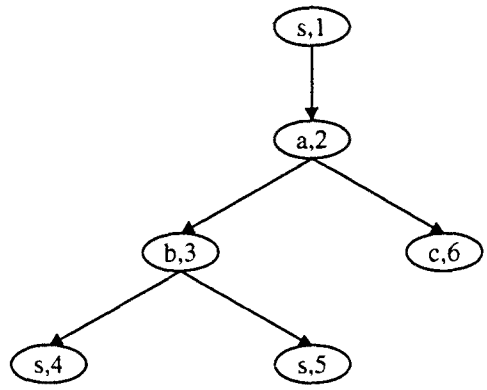
Fase 1. v ← s.
Fase 2. Si todos los arcos están marcados, ir a la fase 4.
Fase 3. Elegir un arco no marcado, marcar su entrada con E y atravesarlo hasta el vértice u. Si u tiene algún arco marcado (si no se visita por primera vez), entonces marcar la salida del arco por el que se ha entrado en u con E, atravesar el arco hasta volver a v e ir a la fase 2. Si u no tiene ningún arco marcado (si es la primera vez que se visita), entonces marcar la salida del arco por el que se ha entrado en u con F, v ← u e ir a la fase 2.
Fase 4. Si no hay arcos marcados con F, fin (se ha vuelto a s y se ha recorrido el grafo completamente).
Fase 5. Usar el arco marcado con F, atravesarlo hasta el vértice u, v ← u e ir a la fase 2.



Aplicación del algoritmo de Trémaux, de Even (1979).

El procedimiento de Trémaux constituye un antecesor de lo que hoy se conoce como búsqueda en profundidad, ya que siempre explora el vértice más profundo haciendo *backtracking* si se llega a un vértice terminal. En el ejemplo anterior, la exploración realizada se podría representar mediante un árbol de búsqueda, en el que las letras

asociadas a cada vértice indican el vértice del grafo explorado, y los números el orden de la exploración:



Representación de la exploración realizada por el algoritmo de Trémaux, mediante un árbol de búsqueda.

Los procedimientos de exploración denominados de *branch and bound* (que como matizan Corominas et al. (1984) no es un algoritmo sino una familia muy numerosa), han sido una de las técnicas mejor estudiadas realizándose numerosas formulaciones cada vez más generales. Como comenta Müller-Merbach (1997), tienen una historia de 37 años y comienzan con Land & Doig (An Automatic Method of Solving Discrete Programming Problems. *Econometrica*, vol. 28, 1960, pp. 497-520), quienes sugieren el principio del árbol de búsqueda para programación entera; el mismo principio es aplicado al TSP por Little et al. (An Algorithm for the Traveling Salesman Problem. *Operations Research*, vol. 11, 1963, pp. 972-989), quienes sugieren el término *branch and bound*.

En 1988, Ibaraki realiza una extensa y brillante exposición acerca de los problemas de optimización combinatoria y su dificultad, los algoritmos de *branch and bound* y algunas aplicaciones en diversos problemas de optimización combinatoria, comenta el uso de éstos y de nuevos procedimientos enumerativos de exploración de grafos en el área de la inteligencia artificial, y la posibilidad de diseñar algoritmos heurísticos utilizando como base los algoritmos *branch and bound* descritos –Ibaraki (1988)–. De todas formas, aunque se puede considerar que Ibaraki es uno de los primeros autores que intenta diseñar un marco común para varios de los procedimientos existentes, no logra formular un modelo general para todos ellos. Por otro lado, en la última década, y tanto en investigación operativa como en inteligencia artificial, han surgido nuevos procedimientos enumerativos (como por ejemplo la *bounded dynamic programming*) y se han publicado diferentes trabajos en los que se utilizan nuevas técnicas y herramientas (técnicas de consistencia y paralelización), aunque también aplicadas a procedimientos singulares.

Es necesario destacar que al utilizar técnicas de exploración en árboles de búsqueda, la mayoría de autores provenientes del área de la investigación operativa se han planteado desde siempre el objetivo de optimizar, mientras que entre los provenientes del área de la inteligencia artificial el objetivo prioritario ha sido obtener soluciones factibles. Actualmente parece que, como mínimo en cuanto a problemas de optimización combinatoria, se coincide en el objetivo de optimizar. Conviene realizar la aclaración anterior, ya que en el uso de los diferentes procedimientos de exploración a veces se pasa del primero al segundo objetivo y viceversa, lo que puede dar lugar a confusiones.

En el presente apartado, se enumeran diversas clasificaciones de los procedimientos de resolución enumerativos (3.1.), se comentan las funciones de evaluación de vértices que presentan diversos autores (3.2.) y se definen las diferentes técnicas identificadas en la literatura referenciada (3.3. a 3.9.).

3.1. Clasificación de los procedimientos de resolución enumerativos.

Los procedimientos de resolución enumerativos admiten varias clasificaciones en función del objetivo o característica principal por la cual se realiza dicha clasificación. Diferentes autores, entre ellos Nilsson (1971), Barr & Feigenbaum (1981), Pearl (1984), Mompín et al. (1987), Companys (1989a), Cortés et al. (1993), Ginsberg (1993), Rich & Knight (1994) y Winston (1994a), realizan clasificaciones semejantes aunque con ciertas particularidades según la exposición concreta de cada autor.

En el presente texto se exponen cuatro clasificaciones diferentes. Si la característica a considerar es la forma de dirigir la exploración utilizando la información que se dispone para la misma, se distingue entre procedimientos ciegos (o sin información) y guiados (o con información heurística); si lo importante es la optimalidad de la solución, se presentan los procedimientos heurísticos y los exactos; si lo que se desea destacar es la factibilidad y calidad de las soluciones obtenidas, entonces existen procedimientos para la obtención de una o todas las soluciones factibles o para demostrar la no existencia de ninguna solución factible, o bien para obtener una solución subóptima, o bien para obtener una o todas las soluciones óptimas; en cambio, si el elemento de principal importancia es la conservación o no de los estados equivalentes según el coste de seguir las trayectorias para alcanzarlos, se presentan los procedimientos con o sin agrupación de estados equivalentes.

En la práctica, la clasificación de estas técnicas resulta ser complicada y confusa, ya que por un lado las clasificaciones presentadas resultan ser demasiado generalistas, y, por otro, los diferentes procedimientos desarrollados tienen muchas características comunes que no los especifican y diferencian claramente. Para exponer los diferentes

procedimientos referenciados, en el apartado 3.1.5. se realiza una enumeración de diferentes conjuntos de técnicas, que no se corresponde con ninguna de las clasificaciones presentadas y que tampoco aspira a ser una nueva clasificación. Dicha enumeración únicamente pretende agrupar estos procedimientos en la medida de lo posible y en referencia a alguna característica común.

3.1.1. Ciegos / guiados.

Atendiendo a la manera de dirigir la búsqueda utilizando la información que se dispone para dicha misión, los procedimientos de resolución enumerativos se clasifican en ciegos y guiados:

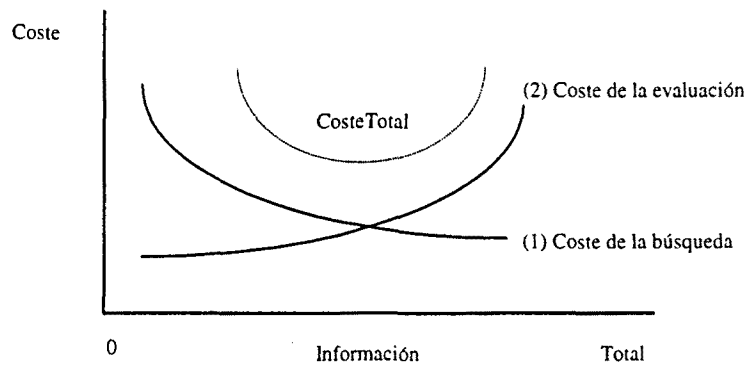
- Procedimientos ciegos (o sin información): la búsqueda con procedimientos ciegos se corresponde aproximadamente con la generación y estudio sistemático de los estados del espacio de búsqueda, hasta encontrar una solución que cumple la condición de objetivo. En estos procedimientos, la elección del vértice a explorar sólo depende de su posición en el árbol de búsqueda y no de otras características, y por tanto, el orden en el que se explora para encontrar la solución sólo se basa en una estrategia de selección uniforme y estática; por otro lado, se consideran impracticables para problemas no triviales al explorar inútilmente una gran región del espacio de soluciones factibles y producirse la temida explosión combinatoria. Las diferentes técnicas difieren en el orden en el que se exploran los vértices, y entre éstas se encuentran: la generación de todas las soluciones, la búsqueda en anchura, la búsqueda en profundidad con *backtracking*, la búsqueda no determinista (según Winston 1994a, interesante cuando no se tiene la certeza de qué búsqueda es mejor si en profundidad o en anchura), el *iterative deepening*, el *iterative broadening*, y otros procedimientos que no son objeto de estudio del presente trabajo como la búsqueda en anchura y en profundidad para grafos AND/OR.

- Procedimientos guiados (o con información heurística): la búsqueda con procedimientos guiados genera y explora estados del espacio de búsqueda aprovechando una cierta información sobre el problema específico, llamada información heurística, para (en muchas ocasiones) restringir drásticamente la exploración, y hasta encontrar una solución que cumple la condición de objetivo. Los instantes en los que se puede aplicar la información heurística son al seleccionar el próximo vértice a explorar (usualmente se toma "el más prometedor" según una función de evaluación o indicador), al elegir el sucesor o sucesores a generar en el curso de la expansión de un vértice (pudiendo generar una expansión parcial) y al decidir qué vértices deben ser descartados del árbol de búsqueda (garantizando la optimalidad de la solución en algunas ocasiones y no pudiéndola garantizar en otras). Entre estos procedimientos destacan la búsqueda primero el mejor, la búsqueda en

anchura limitada, la búsqueda en profundidad-m, el algoritmo A, el algoritmo A* y el algoritmo IDA*.

Companys (1989a) y Winston (1994a) opinan que la eficiencia de la búsqueda aumenta si existe una forma de ordenar los vértices a explorar (en particular mediante información heurística acerca del problema), de modo que los más prometedores se exploren primero. Por su parte, Rich (1990) señala que la información heurística no es necesaria para problemas triviales, ya que se pueden resolver por enumeración exhaustiva, pero que en entornos más complejos esta información juega un rol crucial en la resolución de problemas de forma factible.

Es importante no olvidar que para un problema dado puede haber más de una función de evaluación y que es difícil demostrar cuál se comporta más adecuadamente; además, la elección de la función de evaluación determina críticamente los resultados de la exploración. Por otro lado, se debe tener en cuenta que el tiempo invertido en evaluar los vértices para seleccionar el que se va a explorar, debe ser recompensado con una reducción del tamaño del espacio de búsqueda: en general existe un compromiso entre el coste de evaluación de una función heurística y el ahorro de tiempo de búsqueda que proporciona dicha función. Mompín et al. (1987) muestran este compromiso en la siguiente figura, de la que deducen que el coste total mínimo se encuentra en sistemas no totalmente informados y con un cierto componente de búsqueda:



Curvas de coste en una exploración arborescente, de Mompín et al. (1987).

3.1.2. Heurísticos / exactos.

En función de la posibilidad de asegurar o no la optimalidad de la solución o soluciones obtenidas, se distinguen los siguientes procedimientos:

- Procedimientos exactos: la búsqueda con procedimientos exactos se corresponde con la generación y estudio de estados del espacio de búsqueda, hasta encontrar una o varias soluciones para las que se puede asegurar su optimalidad.

- Procedimientos heurísticos: procedimientos en los que se relaja la condición de optimalidad de la solución o soluciones a obtener. En numerosas ocasiones, se intenta reducir el esfuerzo de búsqueda sacrificando la garantía de encontrar soluciones óptimas; según Nilsson (1971) y Barr & Feigenbaum (1981), en la práctica interesa minimizar una combinación promedio de la calidad de la solución y del coste de la búsqueda necesaria para obtenerla. Además hay que tener presente que, como expone Mishkoff (1988), en general los procedimientos heurísticos no garantizan encontrar una solución factible a problemas específicos aunque éstas existan.

La mayoría de procedimientos desarrollados en la literatura permiten diseñar una o diversas versiones exactas o heurísticas del mismo, en función de, por ejemplo, la condición de finalización de la búsqueda: las técnicas de búsqueda *branch and bound*, de programación dinámica y el algoritmo A*, usualmente son consideradas técnicas exactas, aunque como exponen varios autores se pueden derivar diversos procedimientos heurísticos a partir de éstas. Concretamente, Nilsson (1971, 1980), Corominas et al. (1984), Companys (1989a) y Corominas (1996), presentan la posibilidad de convertir procedimientos exactos en algoritmos heurísticos, prescindiendo de algunas de las condiciones que lo definen y realizando la búsqueda en un área restringida:

a) Limitar el tiempo máximo de ejecución del procedimiento, con lo que se puede llegar a obtener ninguna solución factible, la mejor solución hallada (solución preferible) y una cota del error, o incluso la solución óptima y la confirmación de que lo es.

b) Eliminar vértices mediante reglas "sensatas", como por ejemplo:

- en cada paso, prescindir de los vértices peores entre los sucesores de un vértice o entre todos los generados, o guardar un número concreto de vértices sucesores.
- generar vértices pero sólo guardar el mejor o los mejores.
- en caso de minimizar, eliminar los vértices tales que $z_i + \varepsilon \geq \bar{Z}$, siendo z_i el valor de una cota inferior de dicho vértice, $\varepsilon > 0$ una cota superior del error máximo que se permite cometer (que puede ser función de diversos parámetros, por ejemplo de la posición del vértice en la arborescencia y/o del tiempo que todavía se dispone para resolver el problema) y $\bar{Z}$ el valor de la solución preferible.

c) Limitar la memoria o ejecutar el procedimiento hasta que se excede la memoria disponible.

d) Autolimitar la memoria: expulsar los vértices de, por ejemplo, peor valor de una cota o indicador.

Ibaraki (1988) expone tres procedimientos heurísticos derivados del *branch and bound*, en los que utiliza varias de las ideas expuestas anteriormente: método de la ε-asignación, método del T-corte y método del M-corte; además comenta que éstos se pueden combinar para obtener nuevos procedimientos heurísticos híbridos (técnicas descritas con detalle en el apartado 3.9.1.).

Los diferentes procesos de selección del vértice a partir del cual ramificar en cada nivel, también permiten diseñar múltiples algoritmos heurísticos: por ejemplo, se pueden diseñar diferentes heurísticas a partir de una dada, tomando como criterio para elegir el vértice a explorar: el valor de una cota superior, el valor de una cota inferior, el valor de un indicador, etc. (Álvarez et al. 1988).

Algunos autores como Ibaraki (1988) y Rich & Knight (1994) proporcionan varios argumentos para justificar el uso de procedimientos heurísticos frente a exactos:

> - sin ellos se está desesperadamente enredado en una explosión combinatoria: muchos problemas de optimización combinatoria son muy difíciles de resolver exactamente e incluso con algoritmos muy sofisticados el tamaño de los problemas resolubles es muy limitado,
> - para propósitos prácticos normalmente no se necesita una solución óptima, con frecuencia una buena aproximación es suficiente,
> - si bien las soluciones que se logran con una heurística pueden no ser muy buenas en los peores casos, estos peores casos raramente se presentan en el mundo real,
> - intentar comprender por qué funciona una heurística, o por qué no lo hace, normalmente sirve para profundizar en la comprensión del problema;

además se puede añadir que un procedimiento heurístico sirve para inicializar un proceso de eliminación de estados "no objetivo" en un algoritmo exacto: dada la solución preferible (que como mínimo al inicio coincide con la solución heurística), se eliminan aquellos estados para los que se puede asegurar, mediante una cota, que entre sus descendientes no existe ninguna solución factible mejor que la solución preferible.

3.1.3. Obtención de una solución factible / una solución subóptima / una o todas las soluciones óptimas.

Atendiendo a la factibilidad y calidad de las soluciones buscadas, se presenta la siguiente clasificación:

- Procedimientos para la obtención de una o todas las soluciones factibles, o para demostrar la no existencia de ninguna solución: la búsqueda se basa en la generación y estudio de estados del espacio de búsqueda, hasta encontrar una o todas las

soluciones que cumplen las condiciones de factibilidad, o hasta poder asegurar que no existe ninguna solución que cumple dichas condiciones.

- Procedimientos para la obtención de una solución subóptima: procedimientos que se basan en generar y estudiar estados del espacio de búsqueda, hasta encontrar una solución "de cierta calidad". Usualmente la calidad de una solución se mide en relación a una cota de la solución óptima, lo que implica que, para que sea realmente significativa, la cota también debe ser "de calidad"; en un contexto industrial, la calidad de una solución se puede medir en relación a un valor que se desea obtener y que es considerado alcanzable.

- Procedimientos para la obtención de una o todas las soluciones óptimas: la búsqueda se basa en la generación y estudio de estados del espacio de búsqueda, hasta encontrar una o todas las soluciones para las que es posible asegurar su optimalidad.

3.1.4. Agrupando estados equivalentes / sin agrupar estados equivalentes.

Winston (1994a) es uno de los pocos autores que presenta una clasificación de las técnicas de resolución enumerativas, considerando como elemento de principal importancia, la agrupación o no de estados equivalentes según el coste de alcanzar dichos estados; así, se distinguen dos clases de procedimientos:

- Procedimientos sin agrupación de estados equivalentes: la búsqueda se basa en la generación y estudio de estados del espacio de búsqueda hasta encontrar uno o varios estados considerados objetivo, manteniendo los estados equivalentes (también denominados duplicados o redundantes) y las diversas trayectorias exploradas para alcanzarlos.

- Procedimientos con agrupación de estados equivalentes: procedimientos en los que, en el momento de la generación y estudio de estados del espacio de búsqueda, se van agrupando los estados equivalentes y sólo se conserva la mejor de entre las diferentes trayectorias exploradas para alcanzarlos. Son técnicas de este tipo la programación dinámica, el *branch and bound* en combinación con la programación dinámica, etc., aunque en la práctica se diseñan procedimientos que permiten incorporar esta característica a elección del usuario, haciendo muy difícil una clasificación exhaustiva.

3.1.5. Conjuntos de procedimientos referenciados.

Como ya se ha comentado, a continuación se realiza una breve descripción de los diferentes conjuntos de procedimientos en los que se agrupan las técnicas referenciadas.

Esta pseudo-clasificación se basa en citar en un mismo conjunto aquellas técnicas que presentan ciertas características semejantes entre sí, y únicamente se realiza para presentar los procedimientos enumerativos de una forma más compacta que no una simple enumeración y descripción. Además hay que tener presente que algunos de los procedimientos que se citan podrían ser expuestos en varios de dichos conjuntos de técnicas.

- Procedimientos de búsqueda basados en la "fuerza bruta": aquéllos que para la generación y exploración de estados del espacio de búsqueda no utilizan ninguna técnica de podado de dicho espacio, pudiendo llegar a generar, en el caso extremo y más probablemente que con otros procedimientos, todos los estados. Entre éstos se encuentran: la obtención de todas las soluciones, la búsqueda aleatoria, la búsqueda en profundidad, la búsqueda en anchura, la búsqueda primero el de menor coste, la búsqueda primero el mejor, etc.

- Procedimientos de búsqueda basados en programación dinámica: la característica principal que presentan estas técnicas de búsqueda y que las diferencian del resto, consiste en la agrupación de estados equivalentes y en la conservación de la mejor trayectoria hasta alcanzarlos según una función de evaluación objetivo.

- Procedimientos de búsqueda basados en la poda de vértices mediante cotas: técnicas de exploración que identifican vértices que no contienen soluciones óptimas y los eliminan de la enumeración; para ello, se resuelve un problema relajado del original, cuya solución óptima proporciona una cota del valor de la solución del problema original. Entre los procedimientos basados en la poda de vértices mediante cotas, más conocidos como de ramificación y acotación, cabe destacar, entre otros, el procedimiento *branch and bound*, el *branch and prune*, el *depth-first / breadth-first / best-first / ... branch and bound*, el *branch and cut*, el *branch and price* y el *branch and cut and price*.

- Procedimientos de búsqueda usualmente descritos en el área de la inteligencia artificial: técnicas de exploración expuestas básicamente en el área de la inteligencia artificial y que forman parte de los procedimientos enumerativos de búsqueda en espacios de estados: *dependency directed backtracking, iterative deepening, iterative broadening*, algoritmos A y A*, algoritmo IDA*, algoritmos de potencia heurística, búsqueda en profundidad-m, búsqueda en anchura limitada, etc.

- Procedimientos de búsqueda híbridos: aquéllos que se basan en utilizar, de forma simultánea, las estrategias y/o características diferenciadoras de diversos procedimientos "puros": incorporan varias técnicas que podrían ser consideradas

"puras". Cabe destacar la programación dinámica acotada y los procedimientos *branch and bound* con técnicas de exploración de entornos.

- Procedimientos de búsqueda basados en técnicas de reducción: procedimientos en los que la exploración se basa en el uso sistemático de técnicas que reducen el espacio de estados. Se diferencia entre las técnicas de reducción procedentes del área de la investigación operativa (las técnicas de preproceso) y las diseñadas en el área de la inteligencia artificial (las técnicas de consistencia).

- Procedimientos heurísticos: técnicas que usualmente no son capaces de asegurar la optimalidad de la solución o soluciones obtenidas, e incluso su factibilidad para problemas específicos. Existen procedimientos heurísticos basados en el *branch and bound* (como el método de la ε-asignación, del T-corte y del M-corte, entre otros), en el algoritmo A*, en la programación dinámica, y otros como las estrategias de escalada, la búsqueda en haz de amplitud n, etc.

En los apartados 3.3. a 3.9. se describen los procedimientos enumerativos que se engloban en cada uno de estos conjuntos de técnicas, exponiendo las definiciones que se presentan en la literatura referenciada.

Después de consultar dichos apartados, el lector puede tener la sensación de que se ha descrito un panorama poco estructurado y poco compacto, donde no hay casi nada claro y donde unos autores definen y utilizan unos elementos o procedimientos de una forma y otros de otra, que, aun semejante, no es igual. Por otro lado, parece que no se tienen muy claros los conceptos o que han sido utilizados entremezclados y tal vez de una manera poco formal. El estado del arte que aquí se describe es el entorno real al que se enfrenta cualquier investigador que trabaje en el campo de la resolución de problemas de optimización combinatoria mediante árboles de búsqueda OR. Por este motivo, y a pesar de incorporar ya un gran esfuerzo de síntesis, se considera oportuno presentarlo de esta forma tan poco estructurada y formalizada.

3.2. Funciones de evaluación.

A lo largo del texto, en numerosas ocasiones se ha hecho referencia a los mejores y peores vértices o estados del espacio de búsqueda. Para dirigir la exploración se han definido diversas estrategias de búsqueda, que tienen en común el disponer de un procedimiento para seleccionar el próximo vértice a explorar entre todos los candidatos. Como ya se ha comentado, para evaluar dichos vértices, ordenarlos y poder decir cuál es el considerado como mejor o peor, usualmente se utiliza una función de evaluación (también llamada indicador) que, a veces, incorpora información heurística relativa al

problema. La información que pueden incorporar las funciones de evaluación puede ser muy variada: la probabilidad de que un vértice se encuentre en el mejor camino, medidas o cotas de la distancia a un vértice objetivo, medidas aditivas de coste, etc., pero como comenta Nilsson (1971), habitualmente se les suele asociar una estimación del camino mínimo desde el vértice raíz a un vértice objetivo, pasando por el vértice evaluado.

La función de evaluación de los vértices es de vital importancia en la definición y especificación de los diferentes procedimientos de búsqueda, ya que en función de la concreción de ésta se pueden definir varios de los procedimientos existentes. En la literatura se han expuesto diversas funciones de evaluación y selección de vértices, con diferentes grados de generalidad. A continuación se detallan algunas de las funciones más generales, aplazando la especificación de las funciones más particulares a la definición del procedimiento en el que son utilizadas (apartados 3.3. a 3.9.).

Diversos autores (Nilsson (1971, 1980), Barr & Feigenbaum 1981, Pearl (1983, 1984), Mompín et al. 1987, Ibaraki 1988, Companys 1989a, Korf 1990, Cortés et al. 1993, Ginsberg 1993, Kusiak et al. 1993, Rich & Knight 1994, Greenberg 1996, etc.) exponen funciones de evaluación para procedimientos enumerativos, cada vez más generales. Sea un problema de minimización; una de las formas más habitualmente utilizadas de caracterizar la bondad de un vértice en vistas a su distancia a un vértice considerado como objetivo (que puede ser un estado que proporcione una solución factible, una solución subóptima o una solución óptima), es mediante la siguiente función de evaluación:

$$f^*(n) = g^*(n) + h^*(n),$$

donde:

- n es el vértice considerado,
- $g^*(n)$ es el coste del camino de mínimo coste (en número de arcos o como una función aditiva de los costes asociados a dichos arcos) desde el vértice raíz s hasta el citado vértice n, y está definido para todos los vértices accesibles desde s,
- $h^*(n)$ es el coste del camino de mínimo coste (de nuevo en número de arcos o como una función aditiva de los costes asociados a dichos arcos) desde el vértice n hasta un estado objetivo,
- $f^*(n)$ es el valor que adopta la función de evaluación y se corresponde con el valor del camino de mínimo coste desde el vértice raíz s hasta un vértice objetivo, condicionado a pasar por el vértice n (por tanto $f^*(s) = h^*(s)$).

Como se puede comprobar, $f^*(n)$ es un evaluador perfecto que nos indica siempre el camino de mínimo coste desde el vértice raíz hasta un vértice considerado objetivo; por tanto, la selección del vértice de mínimo valor de $f^*(n)$ conduce directamente a la solución óptima. Lamentablemente, es habitual desconocer $f^*(n)$ y se dispone únicamente de f(n), una estimación de dicho valor. La precisión de f(n) respecto a $f^*(n)$ depende en su totalidad de la calidad del estimador de $g^*(n)$, g(n), y del estimador de $h^*(n)$, h(n). Se define f(n) como sigue:

$$f(n) = g(n) + h(n),$$

donde:

- n es el vértice considerado,
- g(n) es una estimación de $g^*(n)$,
- h(n) es una estimación de $h^*(n)$,
- f(n) es una estimación heurística de $f^*(n)$.

Si el espacio de estados es una arborescencia, el coste desde el vértice raíz s hasta el vértice n por el camino más económico es plenamente calculable, y por tanto $g(n) = g^*(n)$ es un estimador perfecto; de todas formas y si esto no ocurre, g(n) puede estimarse atribuyéndosele el coste del camino de menor coste de s a n en el grafo ya construido, entonces $g(n) \geq g^*(n)$ y g(n) sólo puede equivocarse en el sentido de sobrestimar el coste mínimo.

El estimador h(n) se basa en información heurística acerca del problema y permite encajar un conocimiento heurístico directamente en el proceso formal de búsqueda, por este motivo algunos autores (Barr & Feigenbaum 1981, Mompín et al. 1987, etc.) lo denominan habitualmente información o función heurística. En caso de minimizar es conveniente que el estimador h(n) nunca sobrestime a $h^*(n)$, ya que entonces se pueden eliminar estados de los que se conoce su coste asociado con la seguridad de no eliminar ninguna solución mejor que la solución preferible; se dice que h(n) es admisible (o que h(n) constituye una cota de $h^*(n)$) si $h(n) \leq h^*(n)$ $\forall$ n. Aunque la propiedad de admisibilidad de h(n) es altamente recomendable, es posible diseñar estimadores (y por tanto procedimientos) cuya principal diferencia sea el cumplimiento o no de dicha propiedad: Nilsson (1980), Mompín et al. (1987) y Companys (1989a), consideran que la única diferencia entre el algoritmo A* y el algoritmo A, estriba en que h(n) no pueda o pueda sobrestimar a $h^*(n)$, es decir en la admisibilidad o no de h(n).

Para Pearl (1984), el efecto de g(n) consiste en proporcionar a la búsqueda un componente de exploración en anchura; por otro lado, considerando únicamente h(n) se realiza una búsqueda donde todas las soluciones factibles son igualmente deseables, ya

que se seleccionan los vértices sin considerar el coste acumulado hasta alcanzarlos. Intentando generalizar la función de evaluación para permitir ajustar el balance entre ambas tendencias, Barr & Feigenbaum (1981) exponen que en 1969 Pohl define el *heuristic path algorithm* (HPA), que es un procedimiento de búsqueda enumerativo que incorpora la siguiente función de evaluación:

$$f(n) = (1 - \omega) \cdot g(n) + \omega \cdot h(n), \text{ donde } \omega \in [0, 1].$$

Fijando $\omega = 1$ se obtiene la función de evaluación $f(n) = h(n)$, haciendo $\omega = 0$ la función de evaluación resultante es $f(n) = g(n)$ y fijando $\omega = 0,5$ se consigue la función de evaluación descrita inicialmente $f(n) = g(n) + h(n)$. Variando ω entre 0 y 1 se pueden encontrar infinitas variantes de un mismo procedimiento que incorpore esta función de evaluación. Según Pearl (1984), en cuanto al número de vértices expandidos, algunos experimentos parecen indicar que se obtienen los mejores resultados con $\omega = 0,5$. Ibaraki (1988), basándose en su propia experiencia computacional, matiza que en general $\omega = 0,5$ va mejor que $\omega = 1$ y éstos mejor que $\omega = 0$, aunque también especifica que si h(n) es un buen estimador ω debe tender a 1 y si contiene errores $\omega < 1$.

Para valorar la influencia de la información heurística h(n) en la búsqueda, algunos autores como Nilsson (1971, 1980) y Companys (1989a) comentan el concepto de poder o potencia heurística de un algoritmo de exploración que incorpora la función de evaluación descrita por Pohl. Así, la selección del valor de ω es esencial para calibrar la potencia heurística de un algoritmo:

- con valores pequeños de ω predomina la exploración en anchura y se intenta garantizar que ninguna parte del grafo queda inexplorada para siempre,

- con valores grandes de ω predomina la componente heurística y se busca encontrar rápidamente un camino hasta un vértice objetivo, aunque no sea óptimo.

Según Companys (1989a), la eficiencia de la exploración aumenta en muchos casos si el valor de ω varía con la profundidad del vértice: a poca profundidad se utiliza un valor de ω grande, y a grandes profundidades se toman ω menores para explorar más en anchura e intentar encontrar algún camino hasta un estado objetivo. Barr & Feigenbaum (1981) relatan un nuevo algoritmo propuesto por Pohl, llamado de pesos dinámicos (*dynamic weighting*), en el se que utiliza una generalización de la función de evaluación definida para el HPA:

$$f(n) = g(n) + \omega(n) \cdot h(n),$$

donde la función $\omega(n)$ puede ser mayor o igual a 1 y varía con la profundidad del vértice n, reduciendo su valor a medida que la profundidad aumenta.

Barr & Feigenbaum (1981) también describen el procedimiento heurístico propuesto por Harris, llamado búsqueda con ancho de banda (*bandwidth search*), que se basa en asumir que no es posible encontrar ninguna buena función f(n) que satisfaga la condición de admisibilidad; así introduce la condición de ancho de banda, que requiere una función que, en todos los vértices no objetivo, cumpla $h(n) \leq h^*(n) + \varepsilon$, y como consecuencia define un algoritmo ε-admisible y encuentra soluciones ε-óptimas.

Como es lógico y comentan diversos autores (Nilsson 1971, Barr & Feigenbaum 1981, Hartnell 1986, Kusiak et al. 1993, etc.), la eficiencia de los procedimientos enumerativos depende de la selección de la función de evaluación f(n), que determina críticamente el resultado de la búsqueda y debe depender del objetivo perseguido: obtener soluciones óptimas, buenas soluciones o soluciones factibles. En la práctica, aunque puede parecer un proceso sencillo, el diseño de una buena función de evaluación puede llegar a ser muy complejo, tal y como señala Hartnell (1986). Por otro lado y debido a que la información tiene un coste, Nilsson (1980) recuerda que se debe llegar a un compromiso entre el coste de obtener f(n) y el coste de la computación, tal y como ya se ha comentado en el apartado 3.1.1. para los procedimientos guiados.

A falta de especificar otras características de los procedimientos enumerativos de búsqueda en grafos OR, como por ejemplo el agrupar o no los estados equivalentes o el podar o no vértices, la concreción de la función f(n) permite diferenciar entre sí varios de los procedimiento descritos en la literatura referenciada: por ejemplo, según Greenberg (1996) un caso especial es la búsqueda en profundidad, donde $h(n) = 0$, g(n) se define como la profundidad del vértice n y donde se selecciona el vértice de mayor f(n) (o para no perder el criterio de optimización utilizado -la minimización-, definiendo g(n) como menos la profundidad del vértice n y seleccionando el vértice de menor f(n)); otro caso particular que comenta Greenberg (1996) es el *branch and bound* para programación entera, donde $g(n) + h(n)$ es el valor objetivo de la relajación lineal en el vértice; como exponen Mompín et al. (1987) y Companys (1989a), si no se dispone de información heurística, esto es si $h(n) = 0$, el procedimiento resultante pertenece a los algoritmos de búsqueda en anchura, si se define g(n) como la profundidad del vértice y se selecciona el vértice de menor f(n); si sólo interesa alcanzar un vértice objetivo sea de la forma que sea, Rich & Knight (1994) proponen definir $g(n) = 0$, así se elige el vértice que parece más cercano al objetivo, y, según Barr & Feigenbaum (1981), se minimiza el esfuerzo de exploración a expensas de no buscar una solución de mínimo coste (idea que según estos autores fue utilizada por primera vez por Doran y Michie en 1966 en su algoritmo *graph traverser*); de todas formas y como exponen Nilsson (1971) y Pearl (1984), aunque no sea esencial encontrar un camino de coste mínimo, g(n) se debe

incluir en f(n) ya que es mejor trabajar con g(n) que si ella. Vistas las numerosas posibilidades existentes, parece evidente que el desarrollo de eficientes y efectivas funciones de evaluación es una tarea difícil, llegando a constituir, según Shih et al. (1992), un arte más que una ciencia.

Hartnell (1986) expone la posibilidad de introducir en los procedimientos de búsqueda la selección de los vértices en función del cumplimiento o no de una jerarquía de criterios: primero se selecciona para expandir el vértice que cumple el primer criterio, si hay empate o no existe ninguno, el que cumple el segundo criterio y así sucesivamente. De todas formas, si los criterios son cuantificables se presenta la misma situación que si se dispone de más de una función heurística: se puede evaluar lo prometedor que es un vértice ponderando los valores de dichas funciones y luego seleccionando el de mejor valor ponderado. Este tipo de procedimientos permiten introducir una mayor flexibilidad, al incorporar dichos criterios cuantificables a la función de evaluación mediante coeficientes de ponderación suficientemente diferentes. Si de todas maneras persiste el empate, Hartnell (1986) propone desempatar al azar. Parece conveniente diseñar los coeficientes utilizados para ponderar las diversas funciones heurísticas y/o criterios, dinámicos y en función de las características de dichos criterios, de la profundidad del vértice, etc.

En un primer análisis se puede comprobar, cómo la mayoría de estrategias de selección que se proponen en la literatura únicamente son función del vértice considerado y no tienen en cuenta ni el estado de evolución de la exploración ni las condiciones de entorno en las que se resuelve el problema. Recopilando dichas estrategias de selección y teniendo presente varias características no consideradas, en el apartado 7.4.4. se define la siguiente función general de evaluación y selección de vértices:

$$f(n, t, a_t, e) = \varphi[g(n), h(n), c(n), l_1(n), ..., l_s(n), r(n); t; \bar{Z}, h_t^*, v_t, va_t, m_t, R_t, F_t; T, V, VA_t, M_t, TG_k; \alpha(n, t, a_t, e), \beta(n, t, a_t, e), \delta(n, t, a_t, e), \gamma_1(n, t, a_t, e), ..., \gamma_s(n, t, a_t, e), \tau(n, t, a_t, e)],$$

que es función de n (el vértice considerado), t (el tiempo de ejecución hasta el momento), a_t (el estado de evolución de la arborescencia en el instante t) y e (las condiciones de entorno en las que se resuelve el problema).

3.3. Procedimientos de búsqueda basados en la "fuerza bruta".

Los procedimientos de búsqueda basados en la "fuerza bruta" son aquéllos que para la generación y exploración de estados del espacio de búsqueda, y hasta encontrar una o diversas soluciones que cumplan la condición de objetivo, no utilizan ninguna técnica

de reducción de dicho espacio, pudiendo llegar a generar en el caso extremo y más probablemente que con otros procedimientos, todos los estados. Según Cortés et al. (1993), el espacio de estados ha de ser explorado de forma progresiva y sistemática: se ha de establecer un orden para explorar los vértices factibles y un procedimiento para retroceder cuando se descubre un camino sin salida (un vértice terminal no objetivo). Las diferentes técnicas que se pueden denominar como de "fuerza bruta" se diferencian en el orden en el que se explora el espacio de estados hasta alcanzar un estado objetivo, y difieren entre sí en cuanto a optimalidad, coste computacional, etc. Entre éstas se encuentran tanto procedimientos que no utilizan ningún tipo de información heurística acerca del problema (obtención de todas las soluciones, búsqueda aleatoria, búsqueda en profundidad, búsqueda en anchura, etc.), como procedimientos que sí utilizan información heurística (búsqueda primero el de menor coste, búsqueda primero el mejor, etc.). Como comenta Companys (1989a), estos procedimientos no suelen ser muy interesantes, ya que usualmente resultan ser impracticables para problemas de dimensiones industriales al explorar inútilmente grandes regiones del espacio de soluciones factibles. Su gran ventaja es el hecho de ser universalmente aplicables, pero a costa de ser potencialmente ineficientes (Bruynooghe & Venken 1990).

Los procedimientos que se describen a continuación tienen la característica común de utilizar la "fuerza bruta" para encontrar la solución o soluciones buscadas. Estos procedimientos han sido descritos por diferentes autores, algunos como algoritmos por sí mismos y otros como estrategias para seleccionar el siguiente vértice a expandir dentro de un procedimiento de búsqueda. A veces las definiciones que exponen no son lo suficientemente precisas para saber si se refieren a una estrategia de selección o a un algoritmo concreto, y si es así, qué otras características incorpora el algoritmo de búsqueda. Así por ejemplo, existe el algoritmo *depth first search* que es un algoritmo de ramificación que selecciona como vértice a explorar el creado más recientemente. y también puede existir un algoritmo *depth first search* como el anterior, al que además se le incorpora un procedimiento de agrupación de vértices equivalentes. Otro ejemplo se encuentra en Cortés et al. (1993): después de exponer una serie de procedimientos de búsqueda, comenta que todos se pueden referir a la búsqueda de una solución factible, subóptima u óptima, aunque también a la búsqueda de varias o todas las soluciones, y que la única diferencia la constituye las soluciones que se van guardando.

3.3.1. Procedimiento del Museo Británico (*British Museum*).

Cortés et al. (1993) y Winston (1994a) definen de forma semejante el procedimiento llamado del Museo Británico (*British Museum*), como la técnica que consiste en encontrar todas las soluciones factibles y seleccionar la de mejor función de evaluación. Por tanto es una estrategia de búsqueda de la solución óptima, aunque también se puede utilizar para hallar todas las soluciones óptimas, o una o varias soluciones factibles. Para

generar todas las soluciones factibles se puede recurrir a cualquier procedimiento que genere todos los estados del árbol de búsqueda, y en particular usualmente se cita, debido a su sencillez, la búsqueda en profundidad o en anchura (técnicas que se describen en los apartados 3.3.3. y 3.3.4.). Como es previsible, al explorar todos los estados se produce la temida explosión combinatoria que hace que el procedimiento sea muy ineficiente.

Por su parte, Rich & Knight (1994) definen el procedimiento, al que también denominan generación y prueba *(generate-and-text)*, como una técnica que consiste en generar una posible solución y verificar si es una solución objetivo: si lo es el procedimiento finaliza y si no lo es se generan más soluciones. Si se generan soluciones de forma sistemática y la solución existe, el procedimiento es capaz de encontrarla. Como se puede comprobar es una forma exhaustiva de búsqueda en el espacio de estados del problema, y para estos autores la forma más sencilla de implementación es mediante una búsqueda en profundidad con *backtracking*.

3.3.2. Búsqueda aleatoria.

La búsqueda aleatoria ha sido definida por Shirai & Tsujii (1987) (quienes la denominan búsqueda por prueba y error), por Rich & Knight (1994) y por Winston (1994a) (quien la llama búsqueda no determinista), y consiste en un procedimiento que selecciona al azar el próximo vértice a expandir, es, por tanto, la generación y exploración arbitraria y aleatoria de estados hasta alcanzar un vértice objetivo.

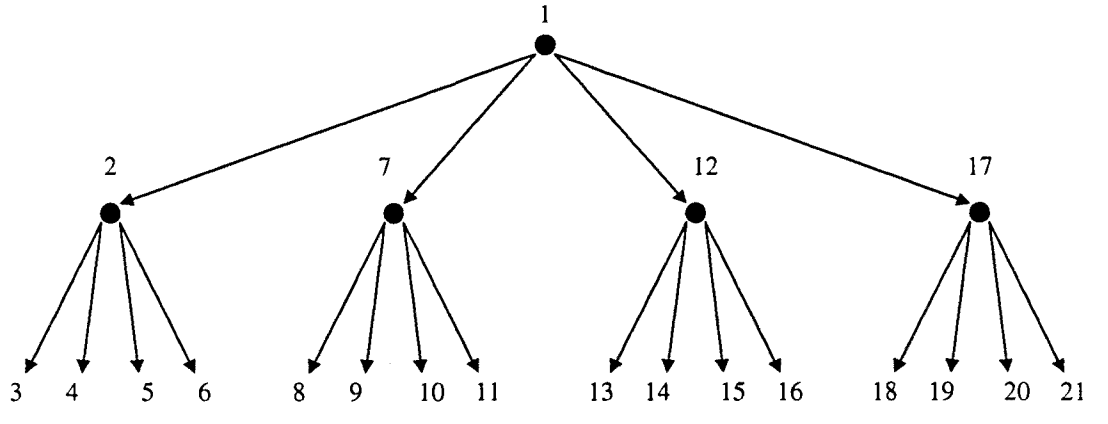
Rich & Knight (1994) comentan que este procedimiento es igual que un algoritmo del Museo Británico en el que la selección de vértices, y finalmente la generación de soluciones, se realiza de forma aleatoria. Por otro lado, para Winston (1994a) es interesante cuando no se tiene la certeza de qué búsqueda es mejor: la búsqueda en profundidad o en anchura.

3.3.3. Búsqueda en profundidad (*depth first search* o DFS).

Entre otros, Nilsson (1971), Ibaraki (1976b, 1988), Even (1979), Barr & Feigenbaum (1981), Pearl (1984), Mompín et al. (1987), Shirai & Tsujii (1987), Mishkoff (1988), Companys (1989a), Cormen et al. (1990), Korf (1990), Cortés et al. (1993), Ginsberg (1993), Liao (1994), Rich & Knight (1994), Winston (1994a) y Brunat (1996), definen el algoritmo de búsqueda en profundidad o búsqueda primero en profundidad (*depth first* o *depth first search* o DFS), también llamado *linear search* o *single branch search* por Ibaraki (1976b), estrategia de búsqueda LIFO por Pearl (1984) e Ibaraki (1988), búsqueda hacia el fondo por Mompín et al. (1987), búsqueda vertical por Shirai & Tsujii (1987), o estrategia en profundidad por Cortés et al. (1993).

La búsqueda en profundidad consiste en explorar siempre un vértice hijo del vértice expandido más recientemente (aunque no se especifica cómo se selecciona dicho vértice hijo), y si éste no tiene sucesores, se realiza un proceso de *backtracking* reinicializando la búsqueda en el vértice antecesor más cercano que posea vértices hijos sin explorar (aunque tampoco se especifica cómo se selecciona el nuevo vértice hijo a explorar si existen varios candidatos); por tanto, primero se expande el vértice generado más recientemente (que coincide con el vértice más profundo en el árbol de búsqueda) y que todavía no ha sido explorado. El procedimiento finaliza cuando se ha generado un vértice objetivo o cuando se ha explorado sin éxito todo el árbol de búsqueda.

El gráfico siguiente muestra un ejemplo de cómo actúa la búsqueda en profundidad, en la generación y exploración de los estados de un espacio de búsqueda. Los números asociados a cada estado especifican el número de orden en el que se explora el vértice referenciado:



Exploración del espacio de estados en una búsqueda en profundidad, de Ginsberg (1993).

Algunos autores (Barr & Feigenbaum 1981, Mompín et al. 1987, Companys 1989a, etc.) proponen poner un límite a la profundidad alcanzada por la exploración, llamado profundidad de corte, ya que algunos espacios de búsqueda pueden tener una profundidad infinita; así, una vez se alcanza dicha profundidad, se retrocede y se continúa por el vértice más cercano.

Para Winston (1994a) este procedimiento constituye una buena idea cuando se tiene la confianza de que todas las trayectorias parciales llegan a callejones sin salida (vértices terminales no objetivo) o conducen a un vértice objetivo, después de un número razonable de etapas. Mompín et al. (1987), Ibaraki (1988) y Cortés et al. (1993) opinan que es un algoritmo ventajoso y eficaz cuando existen muchas soluciones aceptables, y que puede proporcionar rápidamente soluciones factibles aunque rara vez encuentra la solución óptima.

Entre las posibles cualidades que presenta este procedimiento destacan dos (Ibaraki 1988, Korf 1990, Liao 1994 y Rich & Knight 1994):

- es el más económico desde el punto de vista de la cantidad de memoria que se utiliza, aunque experimentalmente Liao (1994) ha encontrado que se tarda mucho más que con otros procedimientos, como por ejemplo el *best bound search* (descrito en el apartado 3.3.6.),
- si se tiene suerte puede encontrar una solución sin tener que explorar gran parte del espacio de estados.

Entre los inconvenientes se deben comentar los siguientes:

- experimentalmente se ha encontrado que suele ser más largo, en cuanto a tiempo de computación y número de vértices explorados, que otros procedimientos como por ejemplo el *best bound search* (descrito en el apartado 3.3.6.),
- rara vez encuentra la solución óptima,
- usualmente necesita una profundidad de corte para evitar entrar en bucles infinitos.

Varios autores comentan que un vértice se puede evaluar mediante la distancia, en número de pasos, desde éste al vértice inicial, y que entonces se selecciona para expandir aquel vértice factible de mayor distancia.

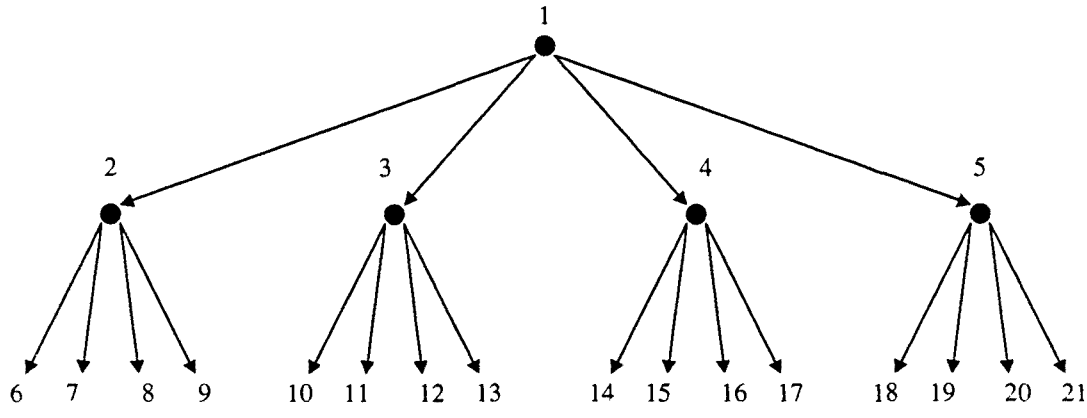
3.3.4. Búsqueda en anchura (*breadth first search* o BFS).

Diversos autores (Nilsson 1971, Even 1979, Barr & Feigenbaum 1981, Pearl 1984, Mompín et al. 1987, Shirai & Tsujii 1987, Ibaraki 1988, Mishkoff 1988, Companys 1989a, Cormen et al. 1990, Korf 1990, Cortés et al. 1993, Ginsberg 1993, Rich & Knight 1994, Winston 1994a, Brunat 1996, etc.) han definido el algoritmo de búsqueda en anchura o búsqueda en amplitud o búsqueda primero en anchura (*breadth first* o *breadth first search* o BFS), también llamado estrategia de búsqueda FIFO por Pearl (1984) e Ibaraki (1988), búsqueda a lo ancho por Mompín et al. (1987), búsqueda horizontal por Shirai & Tsujii (1987), o estrategia en amplitud por Cortés et al. (1993).

La búsqueda en anchura consiste en explorar siempre un vértice del nivel d (de profundidad d) antes que cualquiera del nivel d + 1; para ello, se expanden los vértices en el orden en el que han sido generados (que coincide con la regla de examinar primero los vértices más superficiales); de todas formas, no se especifica el orden en el que se generan los vértices sucesores de uno dado, y por tanto, no se especifica el orden de exploración de los vértices de un mismo nivel d. El procedimiento finaliza cuando se ha

generado un vértice objetivo o cuando se ha explorado sin éxito todo el espacio de búsqueda.

El gráfico siguiente muestra un ejemplo de cómo actúa la búsqueda en anchura, en la generación y exploración de los estados de un espacio de búsqueda. Los números asociados a cada estado especifican el número de orden en el que se explora el vértice referenciado:



Exploración del espacio de estados en una búsqueda en anchura, de Ginsberg (1993).

Para Winston (1994a), este procedimiento constituye una buena idea cuando se tiene la certeza de que el factor de ramificación (número medio de vértices descendientes de un vértice no terminal) es pequeño. Por otro lado, Mompín et al. (1987) comentan que esta técnica se adapta a problemas con pocos caminos de solución.

Entre las posibles cualidades que se presentan de este procedimiento (Barr & Feigenbaum 1981, Cortés et al. 1993 y Rich & Knight 1994), destacan las siguientes:

- no se queda atrapado explorando callejones sin salida,
- si existe solución se garantiza encontrarla, y si existen varias y sólo se busca una, esta estrategia encuentra la de mínimo número de pasos (niveles).

Entre los inconvenientes se debe comentar que si el factor de ramificación es alto, la ocupación de memoria y el tiempo de computación son muy altos (Ibaraki 1988, Korf 1990, Cortés et al. 1993 y Ginsberg 1993).

Korf (1990) señala que un vértice se puede evaluar mediante la distancia, en número de pasos, desde éste al vértice inicial, y que entonces se selecciona para expandir aquel vértice factible de menor distancia. Desde este punto de vista y como comentan Cortés et al. (1993), este algoritmo es semejante a la búsqueda en profundidad excepto en la estrategia de ordenar para explorar los vértices generados: en la búsqueda en

profundidad los vértices acabados de generar se colocan al inicio de una lista ordenada, y en la búsqueda en anchura se añaden al final de dicha lista.

3.3.5. Búsqueda primero el de menor coste (*uniform cost search* o *cheapest first strategy*).

La búsqueda primero el de menor coste (*uniform cost search, uniform cost method, uniform cost procedure* o *cheapest first strategy*), definida por Nilsson (1971), Barr & Feigenbaum (1981) y Pearl (1984) entre otros, consiste en un procedimiento de búsqueda en un árbol, en el que los arcos tienen asociados unos costes no negativos, que se basa en seleccionar los vértices para expandir en orden no decreciente de g(n) (coste mínimo del camino desde el vértice raíz al vértice n); es, por tanto, un algoritmo de búsqueda en anchura asociando a los arcos un coste no negativo y buscando el camino más corto según este coste. Así, y según Nilsson (1971) y Barr & Feigenbaum (1981), si los arcos del árbol de búsqueda tienen igual coste asociado, el algoritmo primero el de menor coste es el algoritmo de búsqueda en anchura; o más en general, la búsqueda en anchura siempre se puede considerar un caso particular de la búsqueda primero el de menor coste.

3.3.6. Búsqueda primero el de mejor cota (*best bound search*).

La búsqueda primero el de mejor cota (*best bound search*), citada por Ibaraki (1976b, 1988) (quien también la denomina *minimum value search*), Liao (1994), etc., consiste en un procedimiento de exploración del árbol de búsqueda en el que se selecciona para expandir el vértice de mejor cota global, que en la terminología utilizada es el de mejor valor de $g(n) + h(n)$, donde $g(n) = g^*(n)$ y $h(n)$ nunca sobrestima a $h^*(n)$ y por tanto satisface la propiedad de admisibilidad. Así, y como ocurre en todos los procedimientos anteriores, el movimiento de avance se realiza seleccionando para expandir el vértice que parece más prometedor de entre todo el conjunto de vértices abiertos (vértices generados pero no explorados) que se tiene hasta ese momento.

Liao (1994) comenta que con la regla *best bound search* el número de vértices tiende a incrementarse a medida que el programa se ejecuta, y que se requiere tanta cantidad de memoria central para recordar todos los vértices del árbol, que limita el tamaño de los problemas resolubles con esta regla. De todas formas, también expone que dicha regla es todavía la estrategia de selección de vértices más usualmente utilizada.

En este momento cobra mas sentido, si cabe, el comentario realizado en la introducción del apartado 3.3., referente a lo poco precisas que son definiciones expuestas en la literatura para saber si se está refiriendo a un algoritmo concreto o a una regla de selección del siguiente vértice a expandir dentro de un procedimiento general de

búsqueda. Además, las denominaciones de los procedimientos existentes tampoco parece ser uniformes y únicas, ya que, por ejemplo, Ibaraki (1988) también denomina a este procedimiento búsqueda primero el mejor (*best first search*) cuando para la mayoría esta denominación corresponde a un procedimiento diferente, al que, por su parte, Ibaraki denomina búsqueda heurística (*heuristic search*).

3.3.7. Búsqueda primero el mejor (*best first search* o BF).

Varios autores, Barr & Feigenbaum (1981), Pearl (1984), Mompín et al. (1987), Ibaraki (1988), Korf (1990), Shapiro (1990), Cortés et al. (1993), Ginsberg (1993), Rich & Knight (1994), Winston (1994a), etc., han definido el algoritmo de búsqueda primero el mejor (*best first* o *best first search* o BF), también llamado *ordered state-space search* por Barr & Feigenbaum (1981), búsqueda heurística (*heuristic search*) por Ibaraki (1988), búsqueda ordenada por Shapiro (1990), u óptimo por niveles por Cortés et al. (1993).

El procedimiento de exploración consiste en una búsqueda en el espacio de estados, en la que el movimiento de avance se realiza seleccionando para expandir el vértice que parece más prometedor de entre todo el conjunto de vértices abiertos que se tiene hasta ese instante, y donde se utiliza para evaluar lo prometedor de un vértice una función de evaluación f(n) de la que se selecciona el valor mínimo. Como aclara Winston (1994a), el movimiento de avance se realiza sin importar donde está el vértice seleccionado en el árbol parcialmente desarrollado, únicamente se tiene en cuenta el valor de f(n).

Para Mompín et al. (1987) este tipo de estrategias pueden ser consideradas estrategias generales de búsqueda heurística. Esta afirmación es apoyada por la exposición de Barr & Feigenbaum (1981) y Korf (1993), en la que destacan que la búsqueda en profundidad (*depth first search*), la búsqueda en anchura (*breadth first search*) y la búsqueda primero el de menor coste (*uniform cost search*), son casos especiales de la búsqueda primero el mejor (*best first search*):

a) Para la búsqueda en profundidad (*depth first search*) se selecciona f(n) como menos el valor de la profundidad del vértice n.
b) Para la búsqueda en anchura (*breadth first search*) se toma f(n) como la profundidad del vértice n.
c) Para la búsqueda primero el de menor coste (*uniform cost search*) se elige f(n) como el coste del camino del vértice inicial al vértice n.

Rich & Knight (1994) comentan que en ocasiones es interesante realizar la búsqueda de forma que los estados equivalentes no se exploren. Este comentario pone de manifiesto que, al menos, existen dos procedimientos de búsqueda primero el mejor que se

diferencian en el hecho de agrupar o no los estados equivalentes, o que existen dos tipos de procedimientos (aquéllos que agrupan los estados equivalentes y aquéllos que no los agrupan) que utilizan una estrategia de exploración del árbol de estados del tipo primero el mejor. Korf (1993) estudia esta posibilidad como un método para aumentar la eficiencia, por lo que puede deducir que, para este autor, el procedimiento de búsqueda primero el mejor original no incluye la agrupación de estados equivalentes.

Según sea la función de evaluación existen diferentes estrategias primero el mejor, que, como expone Pearl (1984), tienen en común que el espacio de búsqueda es un grafo de estados, el vértice seleccionado para la expansión es el de menor valor de f(n), y si existen dos caminos hasta el mismo vértice se descarta el de peor valor de f(n); de todas formas, Pearl (1984) también realiza un comentario parecido al anterior, exponiendo variantes del algoritmo primero el mejor según se realice o no la agrupación de vértices equivalentes y se manejen o no dichos vértices.

El procedimiento de búsqueda primero el mejor (*best first search*) presenta una serie de limitaciones, de las cuales Korf (1993) destaca las dos siguientes: por un lado, la gran cantidad de memoria necesaria para almacenar la parte de la arborescencia ya explorada, y, por otro, el elevado tiempo que se precisa para generar nuevos vértices e insertarlos de forma ordenada en la lista de vértices activos.

3.3.8. Búsqueda primero el mejor* (*best first search** o BF*).

Para Pearl (1984), la estrategia de búsqueda primero el mejor (*best first search* o BF) termina cuando encuentra una solución factible; si se busca una solución óptima se debe cambiar la condición de finalización del procedimiento, y, al algoritmo resultante, Pearl lo denomina búsqueda primero el mejor* (*best first search** o BF*).

3.3.9. Búsqueda primero los mejores (*best few*).

Mompín et al. (1987) definen el procedimiento de búsqueda primero los mejores (*best few*), como una técnica que consiste en explorar un conjunto reducido de alternativas en paralelo, elegidas de acuerdo con la función heurística f(n).

Aunque Mompín et al. (1987) no especifican qué quieren decir con explorar alternativas en paralelo, existen dos posibles interpretaciones igualmente válidas como procedimientos de búsqueda: por un lado, se pueden referir a que en cada etapa del procedimiento se explora el conjunto de los mejores vértices según el valor de la función de evaluación f(n); y por otro, se pueden referir a explorar mediante procesadores en paralelo dicho conjunto de vértices.

3.3.10. *Parallel depth first search* (PDFS).

Técnica de búsqueda descrita por Ibaraki (1988) que consiste en considerar en cada iteración los p vértices más profundos, y en caso de empate los de mejor valor de f(n), y seleccionar para expandir aquél de mejor indicador f(n). Ibaraki comenta que si p = 1 este procedimiento equivale a la búsqueda en profundidad. Por tanto, parece ser que Ibaraki define más específicamente que otros autores la búsqueda en profundidad, ya que según se puede deducir, a igual profundidad selecciona para explorar aquel vértice de mejor valor de f(n).

3.3.11. *Floating down search* (FDS).

Procedimiento descrito por Ibaraki (1988) que consiste en dividir los vértices activos (vértices generados pero todavía no explorados) en dos grupos: el conjunto S, formado por los vértices con igual o mejor indicador f(n) que sus padres, y el conjunto L, compuesto por los vértices con peor indicador que sus vértices padres; y, a continuación, seleccionar el mejor vértice de S y aplicarle una búsqueda en profundidad sin *backtracking*, o, si S = ∅, el mejor de L; y así sucesivamente.

3.3.12. *Jump backtracking.*

Técnica descrita por Ibaraki (1988) que consiste en alternar la búsqueda en profundidad con la búsqueda primero el mejor: se aplica una búsqueda en profundidad sin *backtracking* a un vértice y, cuando ésta acaba, se elige el siguiente a expandir mediante la búsqueda primero el mejor.

3.3.13. Procedimientos de búsqueda usualmente descritos en el área de la inteligencia artificial.

En el área de la inteligencia artificial se han descrito un conjunto de técnicas de búsqueda en espacios de estados, muchas de las cuales se pueden considerar procedimientos de búsqueda basados en la "fuerza bruta". En el apartado 3.6. se definen varias de estas técnicas: *iterative deepening* (3.6.2.), *iterative broadening* (3.6.6.), los algoritmos Z y Z* (3.6.7.), los algoritmos A y A* (3.6.8.), el algoritmo IDA* (3.6.9.), los algoritmos de potencia heurística (3.6.13.), los algoritmos B y B' (3.6.16.), etc.

3.4. Procedimientos de búsqueda basados en programación dinámica (*dynamic programming* o DP).

En pocas palabras y a modo de síntesis, la característica más importante que de cara a los procedimientos de resolución enumerativos presentan las técnicas de búsqueda basadas en programación dinámica determinista finita (con un número finito de estados y de etapas), consiste en la agrupación de estados equivalentes y la conservación de la mejor trayectoria hasta alcanzar dicho estado según una función de evaluación objetivo. De todas formas, cabe destacar que la programación dinámica (*dynamic programming* o DP), procedimiento que comienza a ser formalizado en 1957 por Bellman, es una técnica mucho más amplia y que un estudio riguroso de la misma requeriría uno o diversos textos *adhoc*.

Para exponer las bases y el funcionamiento de la programación dinámica, se sigue a grandes rasgos el hilo argumental utilizado por Kaufmann & Cruon (1967), completado con diversos matices introducidos por otros autores que también han estudiado este tema: Morin (1978), Prawda (1981), Companys (1982), Kumar & Kanal (1983b), Pearl (1984), Ibaraki (1988), Cormen et al. (1990), Hillier & Lieberman (1991), Taha (1991), Bautista et al. (1992), Cortés et al. (1993), Winston (1994b), Dyer et al. (1995), White (1996), etc.

Como ya se ha comentado, la forma que se utiliza en el presente texto para representar problemas de optimización combinatoria es mediante un grafo o árbol de estados OR. En terminología propia de programación dinámica, una decisión provoca la transición de un estado a otro y una secuencia de decisiones constituye una política (una trayectoria) que determina una secuencia de estados. Toda política tiene un sólo estado inicial y un sólo estado final, y es factible si conduce del inicial al final. Por otro lado, cada política aplicada a un estado tiene un coste, y una política factible de mínimo coste se llama política (trayectoria) óptima.

Entonces parece claro que resolver un problema de optimización combinatoria puede verse como el problema de encontrar políticas óptimas en una representación en espacio de estados, es decir, hallar el camino extremo en una representación en espacio de estados.

Según Kaufmann & Cruon (1967), para los problemas que se pueden representar en un espacio de estados polietápico (aunque como se matiza posteriormente sólo es necesario que sea un espacio de estados sin circuitos) se puede aplicar el principio del Óptimo (o de Bellman). Este principio se basa en la siguiente propiedad: considérese una trayectoria óptima entre dos puntos A y B, bajo ciertas condiciones, y en particular si se ha elegido adecuadamente el espacio en el cual se representan las evoluciones del sistema, toda porción MN de esta trayectoria es óptima entre todas las trayectorias

posibles de M a N. Por tanto, el estado del sistema se debe caracterizar por completo: en cuanto a la evolución futura del sistema, no debe haber lugar a distinguir entre dos transformaciones diferentes entre los periodos 1 a n, mientras éstas conduzcan a una mismo estado x_n.

Principio del Óptimo (o de Bellman): toda subpolítica $w_{ij}(x_i, x_j)$ extraída de una política óptima $w_{s,n}^*(x_s, x_n)$ es óptima de x_i a x_j (Kaufmann & Cruon 1967); o dicho de otra forma, si el camino óptimo del vértice x_s al x_n pasa por los vértices x_i y x_j (x_s, ..., x_i, ..., x_l, ..., x_j, ..., x_n), la parte del camino entre x_i y x_j (x_i, ..., x_l, ..., x_j,) es el camino óptimo desde el vértice x_i al vértice x_j (Companys 1982).

La consecuencia de este principio parece evidente: entre el conjunto de políticas que contienen a una subpolítica $w_{s,k}(x_s, x_k)$ dada (óptima o no), la mejor es aquélla que se obtiene al completar la subpolítica dada por una subpolítica óptima $w_{k,n}^*(x_k, x_n)$, y, por tanto, si existen varias trayectorias para ir del estado x_i al estado x_j sólo es necesario conservar la mejor de todas, que será la que forme parte de la trayectoria óptima del estado x_s al x_n y que pase por x_i y x_j.

De todas formas este principio no serviría de nada si no fuese por el hecho de que las trayectorias pueden construirse mediante concatenación de trayectorias más cortas, y que cuanto más corta es una trayectoria, más fácil es saber si es óptima o no entre sus extremos. De aquí surge la necesidad de trabajar con una representación del problema mediante un espacio de estados polietápico, aunque en realidad sólo es necesario que dicho espacio no tenga circuitos, permitiéndose que algunos estados de una etapa no conduzcan a la siguiente sino a una etapa posterior en más de un solo nivel.

El principio de Bellman constituye la base del método de la programación dinámica y permite, considerando sucesivamente 1, 2, 3, ..., n periodos o etapas, construir progresivamente una política óptima resolviendo un problema al que se le hace corresponder una representación en un grafo sin circuitos, con un coste asociado a cada arco y mediante la unificación de los estados equivalentes. Así, y como comenta Ibaraki (1988), si únicamente se busca un camino óptimo desde el vértice inicial a un vértice objetivo, en cada estado sólo es necesario guardar el camino de mínimo coste desde el estado inicial hasta él mismo, pudiéndose eliminar todos los demás. Para este mismo autor, la programación dinámica es el nombre general de los procedimientos que mejoran su eficiencia explotando esta propiedad.

En programación dinámica determinista el estado en la siguiente etapa está completamente determinado por el estado y la decisión de la etapa actual; en el caso probabilístico (que en este texto no se utiliza ni estudia) existe una distribución de probabilidad para el valor posible del siguiente estado (Hillier & Lieberman 1991).

La programación dinámica comúnmente resuelve el problema en etapas, en cada una de las cuales interviene exactamente una variable de optimización (Taha 1991). Los cálculos en las diferentes etapas se enlazan a través de cálculos recursivos de manera que se genera una solución óptima factible a todo el problema, y, según Companys (1982), dicha ecuación funcional recurrente puede ser progresiva (dado un vértice se decide a donde se va) o regresiva (dado un vértice se decide de donde se viene). Dicha relación recursiva identifica la política óptima para la etapa n, dada la política óptima para la etapa (n + 1) (Hillier & Lieberman 1991). Como ya se ha comentado, la teoría unificadora fundamental en la programación dinámica es el principio de Bellman, que dice cómo se puede resolver un problema adecuadamente descompuesto en etapas a través del uso de cálculos recursivos. Por tanto, se puede definir la programación dinámica como lo hace Ibaraki (1988): principio computacional que obtiene políticas óptimas resolviendo ecuaciones funcionales resultantes del principio de Bellman. Sin embargo, según Taha (1991), aunque ofrece una estructura bien definida para resolver el problema en etapas, es "vago" en cuanto a la forma en que se debe optimizar cada etapa.

En este momento se puede exponer la definición que realiza Pearl (1984) de programación dinámica: formulación recursiva de una búsqueda en anchura que, para espacios de búsqueda con estructuras regulares, puede dar expresiones analíticas para el coste o la política óptima.

Companys (1982), Hillier & Lieberman (1991) y Winston (1994b) coinciden a grandes rasgos al exponer las características comunes que se presentan en la mayor parte de las aplicaciones resueltas mediante programación dinámica; éstas son:

1) En lugar de enfocar directamente el problema, optimizando su función económica globalmente, se divide en N etapas, en cada una de las cuales se adopta una decisión que consiste en dar a una variable uno de sus posibles valores.

2) A cada etapa están asociados unos estados. La decisión tomada en cualquier etapa ocasiona la transición del estado de la etapa actual a otro estado de la etapa siguiente. A las transiciones (y a los estados) están asociados valores económicos cuya agrupación constituye la función económica global del problema.

3) Dado el estado en curso, la política o secuencia de decisiones óptima para las etapas que faltan es independiente de las decisiones adoptadas en las etapas ya consideradas. A esta idea se la conoce como principio de Bellman.

4) Se puede establecer una ecuación de recurrencia que permita obtener la política óptima a partir de cada estado con N etapas pendientes, supuesta la política óptima a partir de cada estado con N - 1 etapas.

5) El problema es trivial para N = 0 ó N = 1, lo que permite inicializar la secuencia.

6) Cuando el número de etapas no está acotado, se puede introducir, a partir de la ecuación de recurrencia descrita en 4), su equivalente en el límite (generalmente una ecuación funcional), que se resuelve mediante procedimientos iterativos (de todas formas, en este texto sólo se considera un número de etapas finito).

Según Hillier & Lieberman (1991), una forma de clasificar los problemas de programación dinámica determinista es por la forma de la función objetivo; se puede hacer otra clasificación en términos de la naturaleza del conjunto de estados, que pueden estar representados por variables continuas o discretas. Directa o indirectamente, otros autores (Prawda 1981, Companys 1982, etc.) proponen clasificaciones más amplias y completas de los programas dinámicos; a continuación se relata la clasificación expuesta por Prawda en 1981:

- en cuanto a la forma cómo se combinan las eficiencias parciales del sistema, éstas pueden añadirse en forma de suma, multiplicarse, o bien se puede calcular el máximo o mínimo de las mismas,
- en cuanto a la optimización total de las eficiencias, se puede maximizar o minimizar,
- en cuanto a la eficiencia en sí, ésta puede ser una función discreta o continua,
- en cuanto al sistema que se está optimizando, éste puede tener restricciones o no tenerlas,
- en cuanto al número de etapas, éste puede ser finito o infinito,
- en cuanto a la certeza de la función de eficiencia, ésta puede ser determinista o estocástica,
- en cuanto a la manera de resolver el problema, ésta puede ser en dirección de la entrada a la salida o de la salida a la entrada.

Para Kaufmann & Cruon (1967), la programación dinámica proporciona un método de enumeración particularmente ventajoso si se evita la repetición de estados iguales; pero para Cortés et al. (1993), aunque presenta la ventaja de eliminar trayectorias redundantes y agrupar estados equivalentes, el coste computacional de la detección de vértices comunes a más de un camino puede ser muy alto. Según Winston (1994b), respecto a la enumeración explícita, el número de cálculos que hace la programación dinámica es mucho menor. De todas maneras y como también comenta Ibaraki (1988), para problemas de dimensiones industriales el espacio de estados se hace tan grande que se necesita demasiado tiempo para resolver el problema exclusivamente con programación dinámica. Por otro lado, Hillier & Lieberman (1991) comentan que como todo problema acotado de programación entera pura tiene sólo un número finito de soluciones factibles, resulta natural considerar el uso de algún tipo de procedimiento de enumeración, y que

la programación dinámica proporciona un procedimiento de este tipo aunque no es especialmente eficiente para la mayor parte de este tipo de problemas.

Siendo clave la idea de detectar y agrupar estados equivalentes, existen formulaciones de algunos problemas mediante programación dinámica en las que la detección de dichos estados es más fácil que en otras formulaciones y en otros problemas. Este hecho explica que tradicionalmente se hayan tratado unos problemas mediante programación dinámica (por ejemplo, el problema de la mochila) y otros no. Otro aspecto que influye en la utilidad de los procedimientos de programación dinámica, además de la fácil detección de estados equivalentes, es la frecuencia con la que se presentan estos estados. Un problema para el que funciona extraordinariamente bien la programación dinámica es el problema de la mochila, caracterizando un estado con la capacidad de la mochila ya utilizada; en un ejemplar con 50 objetos y una capacidad máxima de 1.000, la programación dinámica genera una arborescencia con una anchura máxima de 1.001 estados; en cambio, si este problema se resuelve mediante una exploración exhaustiva del árbol formado, se puede alcanzar una anchura máxima de 2^{50} estados. En este problema, además de presentarse muy frecuentemente estados equivalentes, su detección es también muy sencilla.

3.5. Procedimientos de búsqueda basados en la poda de vértices mediante cotas.

Como ya se ha comentado en diversas ocasiones, se puede llegar a invertir mucho tiempo en la resolución de problemas de optimización combinatoria mediante la enumeración explícita: esta técnica no es eficiente. Los algoritmos de resolución deben identificar qué soluciones no son óptimas para eliminarlas de la enumeración, y, así, surgen los procedimientos enumerativos basados en la poda de vértices mediante cotas.

En la resolución de problemas de optimización combinatoria mediante este tipo de procedimientos, la mayor suposición algorítmica realizada es que existe otro problema de optimización, la relajación o el problema relajado, cuya solución óptima proporciona una cota inferior, en caso de minimizar, del valor de la solución del problema original. También se asume que es posible el uso de programas de optimización standard, o al menos más sencillos, para resolver el problema relajado.

Entre los procedimientos de búsqueda basados en la poda de vértices mediante cotas, más conocidos como de ramificación (o de separación) y acotación, cabe destacar el procedimiento *branch and bound*, el *branch and search*, el *branch and relax*, el *branch and reduce*, el *branch and prune*, el *depth-first / breadth-first / best-first / ... branch and bound*, el *branch and bound* con estrategia primero el de mejor cota local (*locally best bound rule*), el *branch and cut*, el *branch and price* y el *branch and cut and price*.

Como introducen Balas et al. (1996), con el objetivo de aprovechar al máximo la formulación y la estructura combinatoria de los problemas de optimización combinatoria, la línea más usualmente seguida por los investigadores en los últimos años ha sido considerar clases de problemas con estructuras especiales e inventar procedimientos que obtengan el máximo provecho en cada caso. Así se están diseñando y probando multitud de procedimientos de cálculo de cotas superiores, procedimientos de acotación, estrategias de ramificación, etc., que intentan aprovechar las estructuras, dominancias y simetrías propias de cada problema.

3.5.1. *Branch and bound.*

El procedimiento de búsqueda en espacios de estados que se detalla a continuación, la técnica *branch and bound*, se puede considerar el procedimiento de búsqueda más general expuesto hasta este punto, que utiliza la ramificación (*branch*) y la acotación (*bound*) en la exploración del espacio de soluciones factibles. Cabe destacar que este procedimiento constituye la base sobre la que se sustentan gran parte de las demás técnicas consideradas, y, por tanto, definiendo el *branch and bound* se están definiendo muchas características de otros procedimientos. Como es ampliamente conocido, no se trata de un algoritmo, consiste en una forma general de enfocar la optimización de problemas de optimización combinatoria, y, de esta forma, se puede comenzar a hablar de un meta-algoritmo.

De manera muy amplia y como comentan diversos autores (Bruin et al. 1988, Nemhauser & Wolsey 1989, Kumar 1990, Roca 1992, Jünger et al. 1994, Grötschel & Lovász 1995, Gass & Harris 1996, Hoffman & Padberg 1996, Sierksma 1996, etc.), se puede definir el *branch and bound* como un procedimiento de resolución de problemas de optimización, que consiste en la partición (ramificación o separación) sucesiva del conjunto de soluciones posibles en subconjuntos más pequeños y en la resolución del problema sobre cada subconjunto. Los problemas resultantes son llamados subproblemas o vértices en el árbol de enumeración y la idea consiste en que la solución óptima del problema es la mejor entre las soluciones óptimas de los subproblemas.

El procedimiento parcial descrito en el párrafo anterior constituye una técnica de ramificación (separación) y exploración del árbol de búsqueda casi siempre muy ineficaz debido al peligro que se produzca la temida explosión combinatoria. Para reducir el número de subproblemas a resolver, en el *branch and bound* habitualmente primero se calcula una solución factible inicial mediante heurísticas, que pasa a constituir la solución preferible, para, posteriormente, calcular cotas del valor de la solución óptima de los subproblemas definidos en los vértices mediante la resolución de unos problemas relajados (que son más fáciles de solucionar): si el valor de la cota de la solución óptima de un subproblema no es mejor que el valor de la función objetivo de la

solución preferible, el subproblema es descartado (podado). El procedimiento finaliza cuando todas las cotas calculadas para los subproblemas no son mejores que el valor de la solución preferible.

La utilidad del *branch and bound* (técnica que debe su nombre al uso de cotas superiores e inferiores para podar) deriva en que, en general, muchas de las soluciones son podadas y sólo se necesita enumerar y explorar una pequeña fracción del espacio de soluciones factibles (Pearl 1984, Ibaraki 1988, Kumar 1990 y Hoffman & Padberg 1996).

Una vez se ha definido a grandes rasgos el funcionamiento básico de los procedimientos *branch and bound*, a continuación se procede a su descripción detallada para un problema de minimización, tomando como hilo conductor, aunque no exclusivamente, la exposición que realiza Toshihide Ibaraki en Ibaraki (1988). De todas formas cabe destacar que son muchos los autores que han definido y trabajado los algoritmos de búsqueda *branch and bound*: Corominas & Companys (1977), Corominas et al. (1984), Mompín et al. (1987), Ibaraki (1988), Companys (1989a), Nemhauser & Wolsey (1989), Cortés et al. (1993), Ginsberg (1993), Jünger et al. (1994), Franco & López (1995), Gangonells & Gómez (1995), Sierksma (1996), etc., y que también son referenciados en este texto.

3.5.1.1. Formalización.

Como ya se ha introducido, para poder utilizar un algoritmo *branch and bound* en la resolución de un problema de optimización combinatoria, el problema original, denominado P_0 según la terminología utilizada en Ibaraki (1988), se debe poder separar (en el sentido definido en el apartado 3.5.1.3.) en un conjunto de problemas parciales más pequeños, denominados P_i, cuya resolución equivalga a resolver el problema original P_0; y esta idea debe aplicarse iterativamente hasta que la separación resulte imposible (así por ejemplo, en el problema de la mochila 0-1 al fijar una variable a 0 ó a 1 se obtienen dos problemas parciales cuya resolución equivale a resolver el problema original).

Si sólo se separa el árbol, se está realizando una enumeración total del espacio de soluciones factibles; para construir un procedimiento que únicamente enumere y examine una pequeña porción del árbol de búsqueda, siendo podado todo el resto, se deben utilizar las siguientes propiedades:

a) Si se llega a saber que un problema parcial P_i no contiene ninguna solución mejor que la solución preferible, no se separará ni se explorará.

b) Si la solución óptima de un problema parcial P_i se puede obtener por otros medios, no es necesario seguir separando dicho subproblema P_i.

c) Si ninguna de las soluciones contenidas en un subproblema P_i cumple las propiedades de obligado cumplimiento para toda o alguna solución óptima del problema (en el sentido definido en el apartado 3.5.1.4.), P_i no se separará ni se explorará.

Para implementar las dos primeras operaciones de podado, Ibaraki (1988) comenta que existen dos métodos básicos (descritos a continuación como los expone dicho autor, aunque con ciertos matices): por un lado presenta el podado por cotas y por otro generaliza el concepto incluyendo el podado por dominancias.

1. Test de cota inferior[2]. Sean $f(P_i)$ y $g(P_i)$ los valores óptimos de los problemas P_i y $\overline{P}_i$ respectivamente, donde $\overline{P}_i$ es el problema relajado de P_i.

Se describe un problema parcial P_i como el siguiente problema de optimización:

$$P_i: [MIN]\ (MAX)\ f(x)$$
$$x \in S_i,$$

donde: S_i es el conjunto de soluciones factibles.
X_i es el conjunto de soluciones.
$S_i \subseteq X_i$, $X_i \subset X$ y $S_i = S \cap X_i$.

Cabe destacar que, como se puede comprobar aunque no es explícitamente expuesto en Ibaraki (1988), las separaciones se realizan en el conjunto de soluciones X.

Por otro lado, se describe un problema relajado $\overline{P}_i$ de un problema parcial P_i como el siguiente problema de optimización:

$$\overline{P}_i: [MIN]\ (MAX)\ g(x)$$
$$x \in \overline{S}_i,$$

donde: $\overline{S}_i$ es el conjunto de soluciones factibles del problema relajado $\overline{P}_i$.
X_i es el conjunto de soluciones.
$g_i(x) \leq f(x), \forall\ x \in \overline{S}_i$.
$S_i \subseteq \overline{S}_i \subseteq X_i$, $X_i \subset X$.

[2] En el apartado 3.5.1.6. se expone con mayor precisión este test, así como los procedimientos de relajación utilizados.

El valor óptimo del problema relajado $\overline{P}_i$ constituye una cota inferior de $f(P_i)$ -por ejemplo, en un programa entero se puede prescindir de la restricción de integridad de las variables y obtener un programa lineal cuya solución óptima constituye una cota inferior de la solución óptima del programa entero-. Además, se cumplen las propiedades siguientes (considerando desde este puntoo que la expresión $g(P_i)$ se refiere a $g_i(x)$):

1. Si la solución óptima de $\overline{P}_i$ cumple todas las restricciones de P_i -en el ejemplo propuesto, todas las variables son enteras-, entonces también es la solución óptima de P_i.
2. Si $\overline{P}_i$ no tiene solución factible, entonces P_i también es infactible.
3. Sea $\overline{Z}$ el valor de la mejor solución conocida (solución preferible) de P_0; si $g(P_i) \geq \overline{Z}$, entonces P_i seguro que no es mejor que $\overline{Z}$, y, por tanto, se puede podar el problema parcial (subproblema) P_i.

2. Test de dominancia[3]. Supóngase que los subproblemas se obtienen fijando las variables a sus posibles valores. Sean P_k y P_h dos problemas parciales obtenidos de P_0 al fijar las mismas variables a sus posibles valores, por tanto, P_k y P_h tienen el mismo conjunto de variables libres. Si cualquier solución factible en P_h también es factible en P_k, y el valor de la función objetivo de P_k no es mayor que el valor de la función objetivo de P_h ($f(P_k) \leq f(P_h)$), entonces se dice que P_k domina a P_h. De esta manera, si existe un problema parcial P_k que domina a P_h y que ya ha sido generado, entonces P_h puede ser podado.

A primera vista puede parecer que la definición que se expone en Ibaraki (1988) del concepto de test de dominancia está incompleta, ya que no se comenta que, además, se deben dar las mismas condiciones de contorno. La definición que se realiza en párrafo anterior es correcta ya que las condiciones de contorno están implícitas en la frase "... Si cualquier ...". Lo realmente difícil es poder comprobar dichas relaciones de dominancia; de todas formas, cabe destacar que para poder asegurar que un problema parcial P_k domina a otro P_h, basta con demostrar que el valor de la solución óptima de P_k no es mayor que el valor de la solución óptima de P_h; pero es probable que para demostrar una cosa se necesite la otra.

Existen otros autores, por ejemplo Bruin et al. (1988), quienes además consideran con entidad propia el test de factibilidad, de forma que un subproblema puede ser eliminado si se puede probar que no contiene ninguna solución factible. De todas maneras, esta forma de podar se puede considerar implícitamente tanto en el test de cota inferior como el momento de generar dichos subproblemas.

[3] En el apartado 3.5.1.7. se expone con mayor precisión el test de dominancia.

Según Ibaraki (1988), la descripción del cuerpo de un algoritmo *branch and bound* consiste en especificar claramente todos los componentes que lo forman (el operador de ramificación, el procedimiento de cálculo de una cota inferior (g), el cálculo de una cota superior (u), las relaciones de dominancia (D) y la función de selección del próximo vértice a explorar) y las relaciones existentes entre ellos en el proceso de exploración del espacio de soluciones factibles.

Durante la computación del algoritmo se generan y exploran vértices (que se corresponden con problemas parciales o subproblemas) que pueden ser activos o no activos. Continuando con la nomenclatura expuesta en Ibaraki (1988), se debe definir:

A: conjunto de vértices activos.
N: conjunto de vértices generados hasta el momento.
O: mejor solución obtenida hasta el momento: solución preferible.
$\overline{Z}$: valor de la solución preferible.
s(A): función de selección del próximo vértice a explorar.
$g(P_i)$: cota inferior del vértice P_i.
$u(P_i)$: cota superior del vértice P_i.
P_jDP_i: relación de dominancia entre los vértices P_j y P_i, por la que el vértice P_j domina al vértice P_i.
G: conjunto de vértices P_i para los que se ha calculado $g(P_i)$.

Una vez definida la nomenclatura básica, a continuación se expone un algoritmo *branch and bound* para la obtención de la primera solución óptima:

Fase A1. Iniciación.

A = {P_0}; N = {P_0}; $\overline{Z} = \infty$; O = $\varnothing$.

Fase A2. Selección.

Si A = $\varnothing$ ir a A9;

si no, seleccionar P_i = s(A) e ir a A3.

Fase A3. Cota superior.

Calcular u(P_i).

Si u(P_i) < $\overline{Z}$, $\overline{Z}$ = u(P_i) y O = {x} (donde x es una solución factible de P_i tal que u(P_i) = f(x)). Ir a A4.

Fase A4. Test tratamiento.

Si $P_i \in$ G ir a A8;

si no ir a A5.

Fase A5. Test cota inferior.

Calcular g(P_i).

Si g(P_i) $\geq \overline{Z}$ ir a A8;

si no ir a A6.

Fase A6. Test dominancia.

Si existe P_k ($\neq P_i$) $\in$ N tal que $P_k D P_i$ ir a A8;

si no ir a A7.

Fase A7. Separación.

Separar P_i en P_{i1}, P_{i2}, ..., P_{ik} y hacer A = A $\cup$ {P_{i1}, P_{i2}, ..., P_{ik}} - {P_i} y N = N $\cup$ {P_{i1}, P_{i2}, ..., P_{ik}}. Volver a A2.

Fase A8. Finalizar P_i.

Hacer A = A - {P_i}. Volver a A2.

Fase A9. Fin.

Si $\overline{Z} < \infty$ el mejor valor hallado es $\overline{Z}$ (que es igual a f(P_0)) y O guarda la solución óptima de P_0;

si no f(P_0) = ∞ y P_0 es infactible.

Para obtener todas las soluciones óptimas, el único cambio a introducir es que en el test de cota inferior y en el de dominancia sólo se deben eliminar los vértices peores y no los que son de igual valor.

A continuación se incluye el diagrama de flujo del algoritmo *branch and bound* que se propone en Ibaraki (1988), para la obtención de una única solución óptima en un problema de minimización.

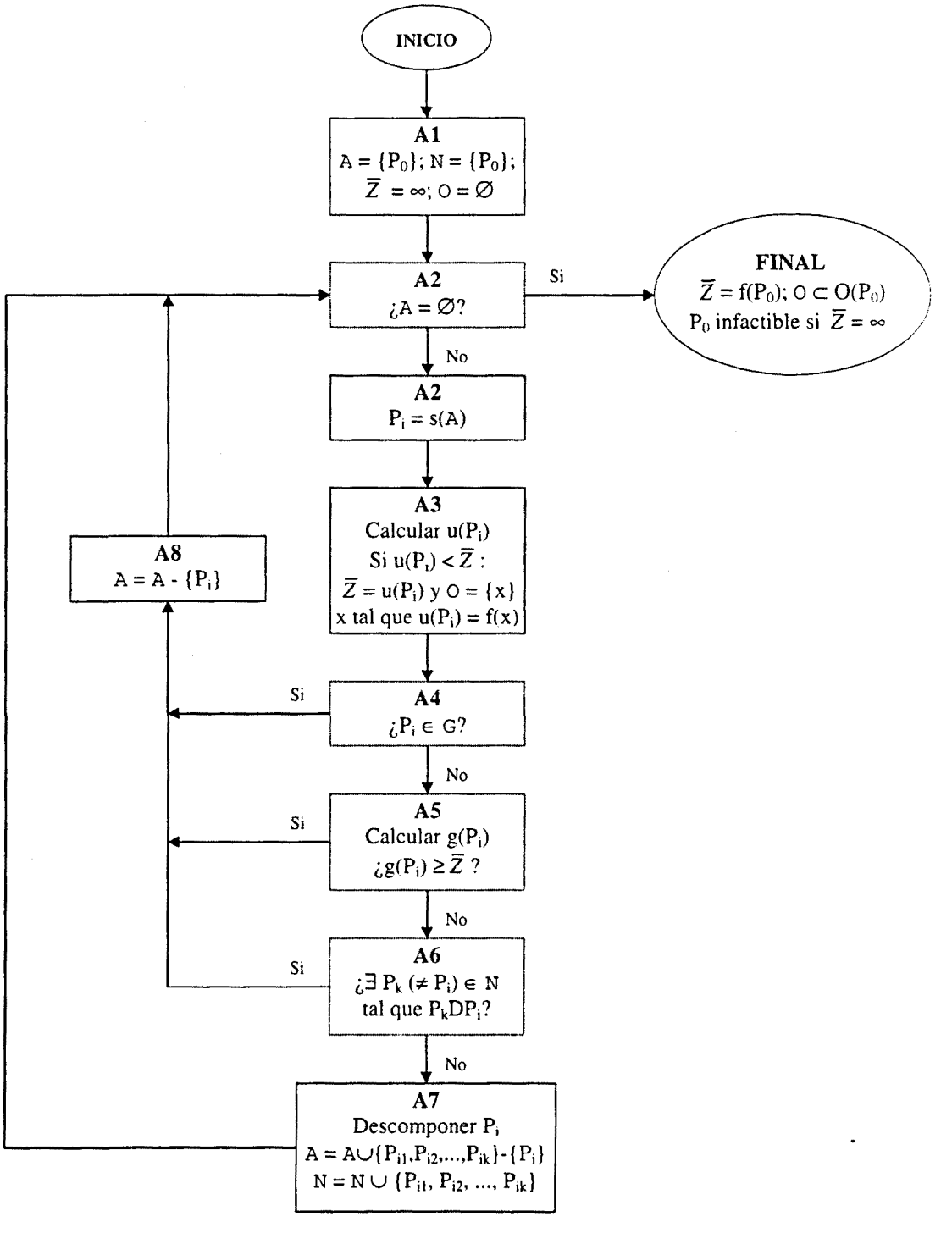


Diagrama de flujo del algoritmo *branch and bound* para la obtención de una única solución óptima en un problema de minimización, de Ibaraki (1988).

El algoritmo *branch and bound* propuesto permite diseñar multitud de variantes en función del orden en el que se ejecuten las diferentes fases, así como en la concreción de éstas. En cuanto al orden de ejecución, en Ibaraki (1988) se proponen diversas variaciones:

- Si el test de dominancia es potente y su cálculo es corto, se puede realizar la fase A6 antes de las fases A4 y A5.
- Calcular $g(P_i)$ y $u(P_i)$ después de la fase A7 y para todos los vértices creados, en vez de hacerlo después de seleccionar el vértice a explorar; de esta manera se facilita el proceso de selección y acotado.
- Aplicar el test de cota inferior a todos los vértices activos, cuando éstos son generados.
- Trabajar con varias cotas inferiores y, por ejemplo, aplicar una sencilla en el momento de la selección (fase A2) y una cota más sofisticada en el test de cota inferior (fase A5).
- Verificar relaciones de dominancia sólo con un subconjunto de los vértices generados; según Ibaraki (1988), esta táctica resulta ser más rápida pero menos efectiva.

Por otro lado, también se pueden diseñar diversas variantes en función de la concreción de las diferentes fases (Corominas et al. 1984 y Cortés et al. 1993). En los apartados 3.5.1.3. al 3.5.1.8. se exponen diferentes desarrollos expuestos en la literatura. A modo de ejemplo se puede comentar que la definición de la fase de ramificación (fase A7) es únicamente una posibilidad: también es posible no generar todos los problemas parciales sino sólo algunos, con lo que, por un lado no se puede eliminar el vértice precedente hasta que se hayan generado todos sus sucesores, y por otro, se debe registrar que descendientes han sido generados.

Cabe destacar que en el algoritmo propuesto no se ha detallado ninguna fase de preproceso que ajuste y reduzca la formulación. Además se propone calcular cotas superiores en cada uno de los vértices generados y no sólo calcular una primera cota superior inicial únicamente en el vértice P_0, como ocurre en gran parte de la literatura.

Companys (1989a) presenta un conjunto de características de los procedimientos *branch and bound*, entre las que cabe destacar las dos siguientes: por un lado, la definición de los vértices objetivo es dinámica y se modifica a lo largo de la exploración: podemos alcanzar un vértice objetivo y no saberlo hasta mucho más tarde; y, por otro, un vértice puede pasar de estar activo a estar cerrado en virtud de la información obtenida a lo largo de la exploración en ramas totalmente ajenas a él.

Como ya se ha introducido en el apartado 2.4. al comentar la representación de los problemas de optimización combinatoria, en el momento de diseñar un algoritmo *branch and bound* es muy importante distinguir entre el árbol posible de búsqueda y el árbol construido en el proceso de exploración: uno es el árbol implícito y el otro el árbol explícito. Como el espacio de búsqueda puede ser infinito o, aun finito, muy extenso, el problema de buscar soluciones se convierte en el de hacer explícita la mínima parte posible del grafo implícito que contiene un estado deseado. Como matizan Mompín et al. (1987) y Companys (1989a), el procedimiento genera un grafo G y un árbol T con los mismos vértices que G, pero con la diferencia que en T cada vértice tiene un único

antecesor; G contiene todos los posibles caminos desde el vértice raíz s hasta cualquier vértice n, T, en cambio, sólo contiene el camino de menor coste o preferente desde s hasta n (si se agrupan los estados equivalentes).

Antes de pasar a describir con detalle las diferentes fases del algoritmo, a continuación se exponen dos algoritmos *branch and bound* típicos en el área de la investigación operativa.

Por un lado está el *algoritmo de Land & Doig* para programación lineal entera, algoritmo que como comentan Corominas et al. (1984) se puede considerar como un precursor del procedimiento *branch and bound*; así, y como señalan Jünger & Thienel (1998), esta técnica llegó a ser el método seleccionado para implantar el procedimiento de resolución *branch and bound* en el software comercial en los años 60 y 70 (según Sierksma 1996, actualmente los procedimientos *branch and bound* continúan siendo el método que más usualmente se utiliza en la práctica para resolver programas lineales enteros y mixtos). El algoritmo de Land & Doig consta de las siguientes fases:

Fase 0. Acotación.

Se resuelve el problema prescindiendo de las condiciones de integridad de las variables.

Fase 1. Separación.

Si todas las variables que han de ser enteras lo son, fin y el óptimo es la mejor solución entera encontrada;

si no, se selecciona una de las variables que debe ser entera y no lo es (x_i), y se construyen los dos programas lineales resultantes de incorporar al programa lineal anterior cada una de las restricciones siguientes:

$$x_i \leq e_i$$

$$x_i \geq e_i + 1$$

donde e_i es la parte entera de x_i.

Fase 2. Acotación.

Se resuelven los dos programas lineales.

Fase 3. Selección.

Entre todos los programas lineales todavía no ramificados, se selecciona aquél para el cual la función económica toma el mejor valor (el vértice de mejor cota) y se vuelve a la fase 1.

Por otro lado se describe el *algoritmo de Balas* para la resolución de programas lineales binarios puros. Esta técnica realiza una búsqueda en profundidad con *backtracking*; para seleccionar la variable a separar, define un índice de imposibilidad de obtener una solución factible para cada variable candidata y selecciona la variable de menor índice (como comentan Corominas et al. 1984, también se puede tener en cuenta únicamente la

función objetivo y seleccionar la variable libre de menor coeficiente en valor absoluto en dicha función objetivo); para aumentar la eficiencia del algoritmo y evitar separaciones inútiles, en cada vértice se pueden hacer tests de eliminación de variables: con ellos se puede reducir el número y complejidad de las restricciones, además de fijar el valor de alguna variable (dichos tests, que en este texto se definen como técnicas de reducción, se presentan en el apartado 3.8.).

3.5.1.2. Particularidades de la función objetivo.

En referencia a la función objetivo, una característica a tener muy presente en este tipo de procedimientos de búsqueda es que ésta no ha de cumplir ninguna condición especial, no es necesario conocer su forma exacta y basta con poder acotarla y calcular su valor en un punto cualquiera (Corominas 1974).

3.5.1.3. Estrategias de ramificación / separación.

Uno de los elementos más importantes de los procedimientos *branch and bound* lo constituye la estrategia de separación o ramificación (*branch*), que especifica cómo separar un problema dado en un conjunto de problemas parciales más pequeños, los subproblemas P_i, de forma que su resolución equivalga a la resolución del problema original P_0. En realidad, la división se realiza sobre el conjunto de soluciones posibles y se resuelve el problema sobre cada uno de los subconjuntos generados.

Según comenta Ibaraki (1988), un mismo problema puede tener diferentes operaciones y estructuras de separación; además, la elección de la estrategia a utilizar influye grandemente en la eficacia del algoritmo *branch and bound* diseñado.

Recuperando la terminología utilizada en la formalización del procedimiento *branch and bound* (apartado 3.5.1.1.), se describe un problema parcial P_i como el siguiente problema de optimización:

$$P_i: [MIN]\ (MAX)\ f(x)$$
$$x \in S_i,$$

donde: S_i es el conjunto de soluciones factibles.
X_i es el conjunto de soluciones.
$S_i \subseteq X_i$, $X_i \subset X$ y $S_i = S \cap X_i$.

Como exponen Ibaraki (1988), Gendron & Crainic (1994) y Franco & López (1995), las operaciones de separación pueden verse como la descomposición o división del

conjunto de soluciones X_i en un número finito de subconjuntos, X_{i1}, X_{i2}, ..., X_{ik}, tales que:

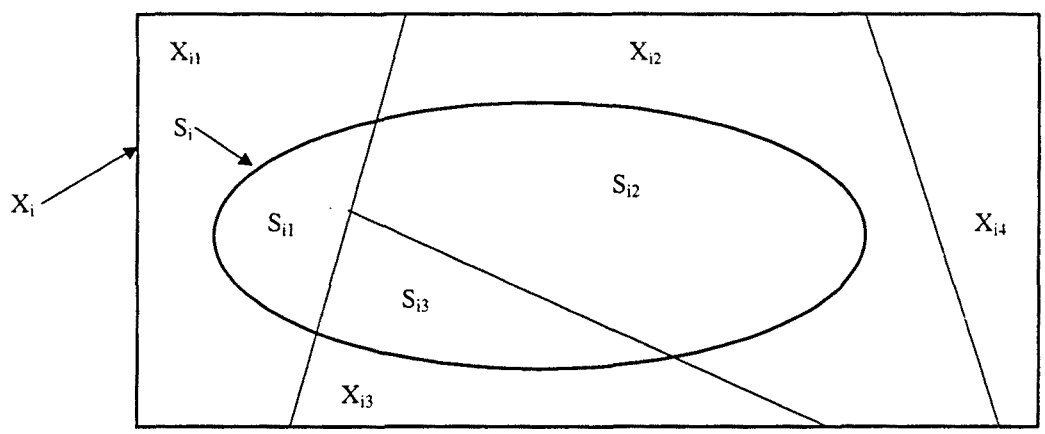
$$X_{ij} \subset X_i, \qquad j = 1, 2, ..., k$$
$$\cup_{j=1}^{k} X_{ij} \supset S_i$$

Una vez realizada la separación, se pueden definir los siguientes k problemas parciales P_{ij}, con j = 1, 2, ..., k:

$$P_{ij}: [MIN] (MAX) \ f(x)$$
$$x \in S_{ij},$$

donde $S_{ij} = S_i \cap X_{ij}$.

Con el objetivo de aclarar conceptos, Ibaraki ilustra la separación anterior en la figura siguiente:

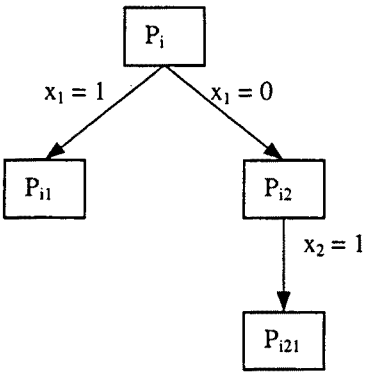


Operación de separación aplicada al problema parcial P_i
(X_i es separado en X_{ij} (j=1,2,3,4), quienes entonces determinan S_{ij}), de Ibaraki (1988).

Según se expone en Ibaraki (1988), Nemhauser & Wolsey (1989) y Gendron & Crainic (1994), en muchos casos la división de X_i en los subconjuntos X_{i1}, X_{i2}, ..., X_{ik} proporciona una partición de X_i, si, además de cumplir $X_i = \cup_{j=1}^{k} X_{ij}$, todos los X_{ij} son mutuamente disjuntos: $X_{ij} \cap X_{is} = \varnothing$ para j, s = 1,...,k, j ≠ s (propiedad que distingue una división de una partición); pero en otros casos no. A menudo también es posible identificar un conjunto Y_i, como resultado de estudiar el problema parcial P_i, tal que ninguna solución $x \in Y_i$ puede ser la solución óptima de P_0; en este caso la condición $\cup_{j=1}^{k} X_{ij} \supset S_i$ queda como sigue $\cup_{j=1}^{k} X_{ij} \supset S_i - Y_i$.

Por otro lado, y aunque no es comentado en Ibaraki (1988), a veces se puede considerar que $X_i \neq \cup_{j=1}^{k} X_{ij}$: por ejemplo, el fijar una variable a un valor concreto puede implicar

el fijar otras variables a otros valores, eliminado otros posibles y, por tanto, regiones del espacio de soluciones; de esta manera, si se realiza esta doble ramificación la unión de los subconjuntos puede no ser X_i. Sea un problema con las variables binarias x_1, x_2 y la restricción $x_1 + x_2 \geq 1$:



Ejemplo de la relación $X_i \neq \cup_{j=1}^{k} X_{ij}$.

en este caso $P_i = P_{i1} \cup P_{i2}$, pero si se realiza la siguiente ramificación, que por otro lado es obligada, se obtiene que $P_i \neq P_{i1} \cup P_{i21}$.

Como se desprende de las exposiciones anteriores, y se comenta en Ibaraki (1988), los operadores de ramificación (o de expansión según Companys 1989a y Cortés et al. 1993), además de dividir el espacio de búsqueda en subespacios, también permiten eliminar algunos de éstos al estudiarlos y comprobar que no es posible que contengan la solución óptima de P_0.

Recordando que se realiza la exposición para un caso de minimización, cabe remarcar que la resolución de P_i puede ser equivalente a la resolución de P_{i1}, P_{i2}, ..., P_{ik}, de forma que $f(P_{ij}) \geq f(P_i)$; de esta manera $f(P_i) \geq f(P_0)$ y las soluciones de los subproblemas P_{ij} constituyen cotas superiores del problema global y de los subproblemas todavía no explorados, por tanto pueden ser utilizadas para podar el árbol de búsqueda.

En Ibaraki (1988) también se exponen algunos ejemplos de procedimientos de ramificación aplicados a problemas muy conocidos:

a) En el problema de la mochila 0-1, fijando el valor de una variable a 0 ó a 1 se obtienen dos problemas parciales.

b) Los procedimientos de separación para la resolución de problemas de programación entera utilizando técnicas de ramificación y acotación, suelen ser de dos tipos:

- fijar valores para la variable x_i, desde su cota inferior L_i hasta su cota superior U_i;
- restringir el valor de la variable x_i a: $\leq \lfloor \bar{x}_i \rfloor$ ó $\geq \lfloor \bar{x}_i \rfloor + 1$, donde $\lfloor \bar{x}_i \rfloor$ es el resultado de resolver el correspondiente programa lineal.

c) En problemas de secuenciación u ordenación, se suelen fijar las variables que pueden ir en la primera posición, en la segunda, en la tercera, ..., etc.

Como se puede observar, en Ibaraki (1988) únicamente se expone la posibilidad de una partición del vértice seleccionado. De todas maneras existe un conjunto de autores (Corominas et al. 1984, Mompín et al. 1987, Companys 1989a, etc.) que, además de la partición, proponen que, por un lado, el procedimiento de separación no sea necesariamente una partición: sea una división, y, por otro, que no se generen a la vez todos los sucesores de un vértice, lo que puede llegar a ser computacionalmente menos costoso (Mompín et al. 1987).

3.5.1.4. Propiedades de la solución óptima.

Un concepto que no es introducido por Ibaraki (1988), aunque sí por otros autores (Kubiak et al. 1997, Tadei et al. 1998, etc.), es la importancia de obtener propiedades que debe cumplir toda solución óptima o tales que existe al menos una solución óptima que las cumple. Como exponen estos autores, este hecho reduce considerablemente el número de soluciones a enumerar en un esquema *branch and bound*: reduce el tamaño del árbol de búsqueda a hacer explícito. De esta forma, una solución total o parcial es factible si cumple dichas propiedades; si no, se poda o, aún mejor, no se genera en el proceso de separación.

Tadei et al. (1998) resuelven un problema de *flow-shop* de n piezas y dos máquinas, con diferentes instantes de disponibilidad de las piezas. Estos autores exponen dos nuevas condiciones que debe cumplir una solución óptima: si se cumplen, respectivamente permiten eliminar o fijar una pieza del inicio de una parte de la secuencia, como integrante de una solución óptima.

3.5.1.5. Cálculo de la cota superior.

En un problema de minimización, el disponer de una solución factible es imprescindible para poder aplicar el test de acotado inferior e identificar, de esta forma, subproblemas/vértices que no contienen soluciones óptimas para eliminarlas/os de la enumeración.

En la literatura publicada en el área de la investigación operativa, la mayoría de textos proponen como primer paso en cualquier algoritmo *branch and bound* el cálculo de una

solución inicial factible de P_0 mediante procedimientos heurísticos, $u(P_0)$, solución que es una cota superior de $f(P_0)$ y que pasa a constituir la solución preferible $\overline{Z}$.

Entre otros autores, Ibaraki (1988), Almiñana & Pastor (1995), Nagar et al. (1996), etc., proponen calcular cotas superiores en cada uno de los vértices P_i generados, $u(P_i)$, y no únicamente en el vértice P_0. Realmente, la cota superior de P_i es la mejor (menor en el caso considerado) de todas las $u(P_i)$ halladas mediante diferentes procedimientos heurísticos y así $u(P_i)$ se utiliza como el valor de la mejor solución obtenida hasta el momento en el vértice P_i.

Según Ibaraki (1988), Nagar et al. (1996) y Johnson et al. (1997), si se dispone de una buena cota superior -para lo cual, y como exponen Grötschel & Lovász (1995), se necesitan heurísticas de calidad- el podado de vértices mediante el test de acotado inferior puede llegar a ser muy potente. Además, y como es lógico, conviene conocer una solución factible cuanto antes para comenzar a podar desde el inicio (Corominas et al. 1984 y Gendron & Crainic 1994).

Los procedimientos heurísticos utilizados para obtener soluciones factibles pueden ser de cualquier tipo, así es posible utilizar algoritmos heurísticos derivados de procedimientos basados en el propio *branch and bound* o en técnicas generales de exploración del espacio de soluciones factibles. Concretamente, en *The 13th Biennial European Conference on Artificial Intelligence*, Schaerf (1998) comenta que la búsqueda local constituye un paradigma emergente para la búsqueda combinatoria, que recientemente ha demostrado ser muy eficiente para un gran número de problemas de *scheduling*; se basa en la idea simple de navegar en el espacio de búsqueda pasando iterativamente de una solución a otra de su vecindario (compuesto por las soluciones que pueden ser obtenidas aplicando un cambio local a ésta), y se propone la combinación de técnicas de búsqueda local como el recocido simulado, la búsqueda tabú y varias clases de *hill-climbing*, con búsquedas con *backtracking*. En el área de la investigación operativa estas técnicas no son tan recientes: en Fischetti et al. (1995), para resolver un modelo de programación lineal entera se utiliza un algoritmo *branch and cut* (procedimiento descrito en el apartado 3.5.8.) en el que para obtener buenas soluciones factibles proponen aplicar procedimientos de "refinado" a soluciones factibles ya disponibles (posiblemente obtenidas mediante heurísticas); por su parte, y con el objetivo de aumentar la eficacia del *branch and bound* que proponen, Guillén et al. (1995) desarrollan un procedimiento de generación de soluciones factibles a un coste computacional muy bajo, que consiste en un algoritmo heurístico que implementa la idea de que soluciones cercanas a la óptima deben tener una estructura similar a la óptima (¿optimización local?).

Por otro lado, Johnson et al. (1997) proponen una forma de realizar el menor trabajo posible si se utiliza la programación lineal como procedimiento de acotado inferior.

Como el resolver un programa lineal usualmente requiere mucho esfuerzo computacional, se deberían utilizar los resultados del programa lineal tanto como sea posible y, en este caso, para obtener una solución factible. La técnica consiste en el redondeo sucesivo de las variables fraccionales hasta que se encuentra una solución o hasta que se detecta infactibilidad; es obvio que el orden en el que se redondea tiene gran impacto en la posibilidad de encontrar una solución entera, pero para algunos problemas de optimización combinatoria específicos, a veces es posible, basándose en su estructura particular, obtener una ordenación que garantiza una solución entera.

Según Ragsdale & Shapiro (1996), existe un concepto erróneo, comúnmente mantenido en el área de la investigación operativa, que considera vital el uso de una buena solución inicial para resolver programas matemáticos rápidamente, utilizando procedimientos *branch and bound*. Este concepto erróneo consiste en que la eficiencia computacional del *branch and bound* mejora, o al menos no empeora, al especificar una buena cota inicial del valor de la función objetivo. Esta creencia se refleja en el manual de usuario de diversos paquetes comerciales y en diversos artículos de investigación; sin embargo, aunque no es injusta, no es totalmente cierta: el uso de una mejor solución inicial puede conducir a veces a una búsqueda *branch and bound* más larga (Ragsdale & Shapiro 1996). Sea un problema de minimización en el que $\overline{Z}$ es el valor de la solución preferible. Se puede esperar que si se resuelve un problema con una $\overline{Z}$ inicial = $+\infty$ y con una $\overline{Z}$ mejor, la segunda resolución no necesite más vértices que la primera; sin embargo, como exponen Ragsdale & Shapiro (1996), se han descubierto casos en los que esto no ocurre y en los que en el segundo caso se necesitan muchos más vértices que utilizando $\overline{Z} = +\infty$. Para entender la razón de esta anomalía, se puede considerar el impacto que tiene la mejor solución en diferentes tipos de estrategias de selección del próximo vértice a ramificar utilizadas por algunos paquetes informáticos, como son la "mejor proyección", "pseudo-coste" y "error relativo" (todas ellas expuestas en el apartado 3.5.1.8.). Todas estas estrategias seleccionan vértices basándose en una estimación de la mejor solución entera factible que se podría obtener de cada subproblema; estas estimaciones dependen directamente del mejor valor inicial o actual, y por tanto, el uso de una solución inicial puede alterar el orden en el que son seleccionados los vértices en un *branch and bound*, desviando la búsqueda del camino que guía a la solución óptima y provocando la necesidad de un mayor número de vértices.

Respecto al trabajo de Ragsdale & Shapiro (1996) conviene hacer una precisión importante para evitar errores de concepto. El uso de una buena solución inicial para resolver programas matemáticos es muy importante de cara a realizar, lo antes posible, un eficiente proceso de podado. Otro aspecto consiste en el uso que se realiza de dicha información, y cómo el valor $\overline{Z}$ es incorporado en las diferentes estrategias de selección del próximo vértice a ramificar.

3.5.1.6. Procedimientos de acotación / relajación.

Otra de las fases cruciales que es necesario especificar claramente en el diseño de un algoritmo *branch and bound*, es el procedimiento de cálculo de la cota g (*bound*) -inferior en caso de minimización y también denominado procedimiento de relajación-, que especifica una forma de excluir un subproblema de la exploración: cómo se identifican, para eliminarlos, los problemas parciales P_i que no contienen soluciones óptimas.

Continuando con la terminología utilizada en Ibaraki (1988), se denomina cota inferior del valor óptimo de un subproblema P_i ($f(P_i)$), al valor $g(P_i)$ que cumple $g(P_i) \leq f(P_i)$ para $P_i \in \wp$ (donde $\wp$ es el conjunto de vértices de la arborescencia).

Como ya se ha comentado, para un problema parcial P_i se define una relajación $\overline{P}_i$ como el siguiente problema de optimización:

$$\overline{P}_i\text{: [MIN] (MAX) } g(x)$$
$$x \in \overline{S}_i,$$

donde: $\overline{S}_i$ es el conjunto de soluciones factibles del problema relajado $\overline{P}_i$.
X_i es el conjunto de soluciones.
$g_i(x) \leq f(x), \forall\, x \in \overline{S}_i$.
$S_i \subseteq \overline{S}_i \subseteq X_i,\ X_i \subset X$.

Por tanto, $g(P_i)$ es el resultado de resolver la relajación $\overline{P}_i$ del subproblema P_i: el espacio de soluciones factibles se amplía, y, en caso de minimizar, el valor óptimo para $\overline{P}_i$ nunca excede al valor de $f(P_i)$. Cabe recordar que la expresión $g(P_i)$ se refiere a $g_i(x)$.

Como ya se ha expuesto en la formalización inicial, se cumplen las siguientes propiedades:

1. Si $\overline{P}_i$ es infactible P_i también y se asume por convenio que $g(P_i) = f(P_i) = \infty$.
2. Asumiendo que el valor de la función objetivo g(x) es igual a f(x), si una solución óptima de $\overline{P}_i$ es factible en P_i, también es solución óptima de P_i; si $g(x) \neq f(x)$ entonces esto no es cierto.
3. Sea $\overline{Z}$ el valor de la solución preferible de P_0, si $g(P_i) \geq \overline{Z}$, entonces P_i se puede podar.

De esta manera y como señalan Bjorndal et al. (1995), el procedimiento de acotado se basa en la resolución exacta del problema relajado. De nuevo cabe destacar que para la

resolución de la relajación $\overline{P}_i$, a veces es posible el uso de programas de optimización standard, o al menos más sencillos.

A continuación se comentan diferentes procedimientos de acotación expuestos en la literatura:

a) Programación lineal.

En la resolución de programas lineales mixtos se obtiene una cota inferior típica, denominada por Ibaraki (1988) $g_L(P_i)$, al resolver el programa lineal resultante de relajar la condición de integridad de las variables del correspondiente programa matemático.

A lo largo de la historia, el procedimiento de acotado basado en la resolución de los programas lineales resultantes de dicha relajación ha sido, parece ser, la técnica más utilizada (Marsten & Morin 1978, etc.). Incluso en nuestros días existen autores que habiendo experimentado con varias formas de obtener cotas inferiores, como por ejemplo la relajación lagrangiana, comentan que obtienen los mejores resultados con la relajación lineal (Dillenberger et al. 1994 en la resolución de un problema de planificación de la producción mediante programación lineal mixta y Johnson et al. 1997). De todas formas, esto no quiere decir que no existan problemas específicos para los que se obtienen mejores resultados con otros procedimientos de acotado.

Según Johnson et al. (1997), una gran parte del tiempo consumido por el *branch and bound* se invierte en resolver programas lineales; así, las mejoras aparecidas en los últimos años en la resolución de estos programas matemáticos, que han conseguido que en estos momentos la metodología existente sea muy eficiente, han repercutido notablemente en la mejora de eficiencia de los procedimientos *branch and bound.* Además, en estos momentos los códigos modernos de programas lineales tratan fácilmente la adición de filas (restricciones) y de columnas (variables), lo que es esencial para los nuevos procedimientos *branch and bound* denominados *branch and cut*, *branch and price* y *branch and cut and price*, técnicas descritas en los apartados 3.5.8., 3.5.9. y 3.5.10., respectivamente. Como exponen estos mismo autores, los métodos de punto interior no son muy utilizados en *branch and bound*, ya que es molesto reoptimizar un programa lineal con un procedimiento de punto interior cuando se han añadido filas o columnas.

A pesar de todo, y como el resolver un programa lineal usualmente requiere mucho esfuerzo computacional, Johnson et al. (1997) proponen utilizar los resultados de éste tanto como sea posible. Así, la solución del programa lineal se utiliza usualmente de tres maneras:

- el valor de la solución es utilizado como cota del vértice activo del árbol de búsqueda;
- la información asociada a los costes reducidos se utiliza para fijar variables;
- y la solución del programa lineal también su utiliza para guiar la selección de los vértices y las variables de ramificación.

b) Relajación lagrangiana.

El procedimiento de acotación basado en la relajación lagrangiana consiste en incorporar a la función objetivo algunas restricciones (según Corominas 1974 y Hoffman & Padberg 1996, usualmente las que dificultan los cálculos), dejando otras fuera, por ejemplo las de no negatividad. Sea un problema parcial P_i descrito como el siguiente problema de optimización:

$$P_i\text{: [MIN] } f(x)$$
$$A \cdot x \geq b$$
$$x \in X_i,$$

donde X_i es el conjunto de soluciones.

La relajación lagrangiana se define mediante el siguiente problema de optimización:

$$\overline{P}_i(\lambda)\text{:[MIN] } L(x; \lambda) = f(x) + \lambda^T \cdot (b - A \cdot x)$$
$$x \in X_i$$

Son varios los autores que comentan esta técnica de acotación (Corominas 1974, Ibaraki 1988, Dyer et al. 1995, Hoffman & Padberg 1996, Johnson et al. 1997, Marín 1997, etc.); entre éstos, en Ibaraki (1988) se señala que el incluir restricciones en la función objetivo proporciona mejores cotas que las que se obtienen ignorando totalmente dichas restricciones. Además, la mayoría considera que es muy importante disponer de buenos multiplicadores de lagrange, ya que contra mejores son éstos, las cotas que proporcionan son más ajustadas; de todas formas presenta el inconveniente de que hay que hallar los valores de los multiplicadores λ (Corominas 1974), para lo cual, y según Hoffman & Padberg (1996), se van resolviendo subproblemas hasta que se encuentran los valores óptimos de dichos multiplicadores.

Por otro lado, Marín (1997) comenta que la técnica de relajación lagrangiana presenta la ventaja de que se pueden obtener diversas cotas: tanto en función de la formulación como del conjunto de restricciones relajadas.

c) Relajación subrogada (*surrogate relaxation*).

La técnica de relajación subrogada, por ejemplo utilizada por Almiñana & Pastor (1995) para resolver un problema de cubrimiento total, consiste en sustituir todas las restricciones del problema por una nueva restricción obtenida como resultado de sumar todas las restricciones sustituidas.

Inmediatamente se pueden plantear diversas cuestiones interesantes: ¿por qué sumar todas las restricciones en una sola?, ¿por qué no sumarlas en dos, o en tres, o ...?; ¿por qué sumarlas directamente?, ¿por qué no ponderarlas con unos pesos: $\alpha \cdot R_1 + \beta \cdot R_2 + \gamma \cdot R_3 + ...$?, el problema reside en hallar unos pesos α, β, γ, ... que suministren la cota más ajustada posible.

Estudiando la literatura publicada sobre el tema, queda patente que en un mismo problema se pueden utilizar diferentes procedimientos de acotado, lo que influye en gran medida en la eficacia del algoritmo *branch and bound* diseñado (Bjorndal et al. 1995). Por otro lado, y como comentan Ibaraki (1988), Gangonells & Gómez (1995), Grötschel & Lovász (1995) y Washburn (1998) entre otros, el tiempo de computación necesario para hallar $g(P_i)$ es un factor crucial para determinar la eficacia del algoritmo; así, para que el podado de vértices mediante el test de acotado inferior sea potente, se debe disponer de una g que pueda ser computada eficientemente y que proporcione una cota precisa (ajustada), lo contrario (o incluso la inexistencia de un principio de acotación) puede proporcionar procedimientos extremadamente ineficientes (Corominas 1974 y Bjorndal et al. 1995). Washburn (1998) va más allá y matiza que, en algunas ocasiones y para problemas particulares, se obtienen mejores resultados con cotas más imprecisas pero mucho más rápidas de calcular (se sacrifica ajuste por velocidad): se puede producir una mayor ramificación que con cotas más ajustadas, pero el tiempo total de cálculo es menor; uno de los argumentos que esgrime este autor es el siguiente: como muchas de las soluciones parciales son muy malas es fácil probar que no son óptimas, y es importante no invertir mucho tiempo comprobándolo.

Apoyando las ideas expuestas en el párrafo anterior, Grötschel & Lovász (1995) comentan que muchos de los desarrollos realizados para obtener soluciones a problemas difíciles se han basado en la invención de mejores relajaciones, aprovechando las propiedades estructurales y con rápidos algoritmos para solucionarlas. Por su parte, Johnson et al. (1997) recuerdan que como los programas lineales mixtos disponen de diferentes formulaciones en términos de variables y restricciones, los programas lineales relajados de dichas formulaciones pueden ser enormemente diferentes en términos de la calidad de las cotas que proporcionan; por tanto, también se puede intentar controlar el tamaño del árbol proporcionando una buena formulación inicial.

Una idea interesante expuesta en Corominas et al. (1984) y Gangonells & Gómez (1995), consiste en introducir el concepto de acotado dinámico. En un mismo algoritmo *branch and bound* puede haber varios procedimientos de acotado, tanto para un mismo vértice de la arborescencia (aplicar varias técnicas de acotado y asignar al vértice la cota más ajustada de entre todas las calculadas), como para los diversos vértices generados (aplicar una técnica a ciertos vértices y otras a otros: por ejemplo, un procedimiento para los vértices más próximos al vértice raíz y, por comportamiento o tiempo de ejecución, otro para los de mayor profundidad en la arborescencia). Corominas et al. (1984) matizan que incluso para hallar una cota de un vértice mediante programación lineal, en vez de resolver completamente el programa lineal resultante, se puede ir calculando una cota cada vez más fina como resultado de ir haciendo iteraciones en el símplex-dual: se busca el empeoramiento que se obtendría si se hiciera una iteración y si la cota es peor que la solución preferible se poda el vértice, si no se hace esta primera iteración y se busca el empeoramiento que se obtendría si se hiciera una nueva iteración, y si la cota es peor que ..., y así sucesivamente. Se trata de obtener cotas menos ajustadas pero más rápidas de calcular, que permitan podar rápidamente los vértices.

Otro concepto a tener presente en los procedimientos de acotado es la existencia de dominancias entre cotas: en un problema de minimización, se dice que una cota domina a otra si la primera es siempre mayor o igual que la segunda y ambas no son siempre iguales (Marín 1997).

Como exponen Bjorndal et al. (1995), otra de las funciones de las cotas, además de la de podar vértices de la arborescencia, es que son necesarias para certificar la calidad de los procedimientos heurísticos, deriven o no de algoritmos *branch and bound.*

Los algoritmos *branch and bound*, y en particular la potencia de sus procedimientos de acotación y podado, pueden ser utilizados para otros procedimientos de búsqueda. Nagar et al. (1996) utilizan un procedimiento *branch and bound* para generar información que es usada para guiar la búsqueda en un algoritmo genético. El procedimiento *branch and bound* se utiliza para generar información que asegure si cierta secuencia parcial, en un problema de secuenciación *flow-shop*, puede generar o no soluciones óptimas cuando se complete en una solución factible: por ejemplo, en un problema con 6 tareas, si se conoce que la cota inferior de una secuencia parcial (1, 3, 4) es mayor que la solución preferible, no es necesario evaluar las secuencias resultantes a partir de esta secuencia parcial (1, 3, 4, 2, 5, 6), (1, 3, 4, 2, 6, 5), ..., etc.

3.5.1.7. Dominancias.

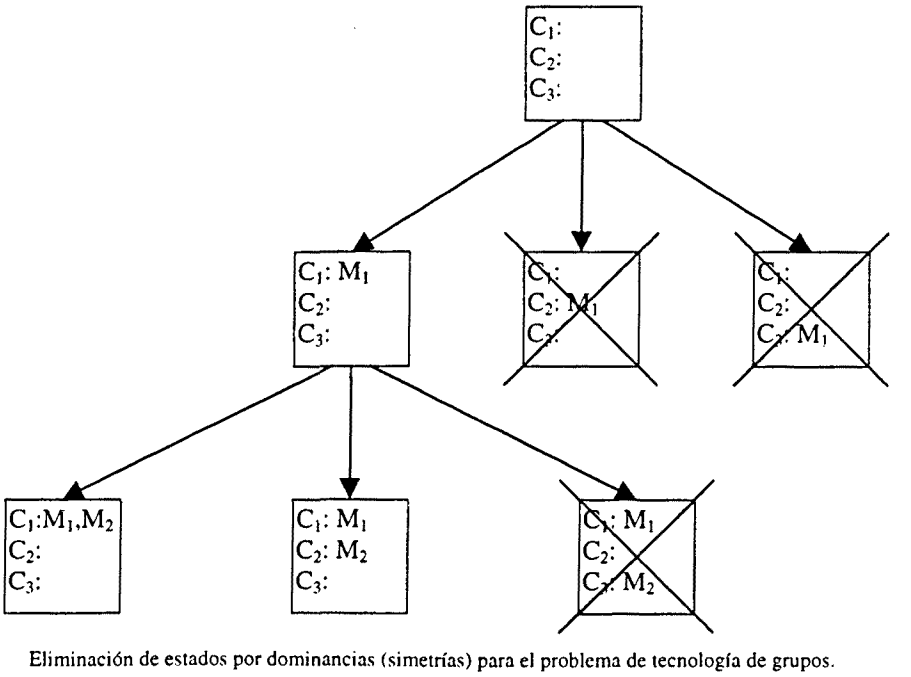
Comprobar si existen relaciones de dominancia entre vértices de la arborescencia que representan el mismo estado, también persigue el objetivo de reducir la enumeración

explícita mediante la poda de vértices, al eliminar aquéllos que son equivalentes; de esta manera, sólo se generan soluciones parciales diferentes (Bosch 1993, entre otros).

Según Ibaraki (1988), el podado por dominancias es un recurso que puede llegar a ser tan potente como el test de cota inferior, pero según Gendron & Crainic (1994) se utiliza raramente. Concretamente, se basa en la relación de dominancia D siguiente: si $P_i D P_j$, es decir si P_i domina a P_j, y P_i ha sido generado, P_j puede ser eliminado, ya que seguro que la solución que se obtenga de P_i no será peor que la que se pudiera obtener de P_j: $f(P_i) \leq f(P_j)$. Como exponen Bruin et al. (1988), un subproblema P_i domina a otro P_j si se puede probar que para toda solución factible de P_j existe al menos una solución factible de P_i con menor o igual valor de la función objetivo.

Ibaraki (1988) propone un sencillo ejemplo aclaratorio: en la búsqueda del camino mínimo desde i hasta j, si se llega a k por dos caminos diferentes, el camino con menor valor asociado de i a k domina al otro.

Kusiak et al. (1993) resuelven el problema de tecnología de grupos mediante una heurística basada en *branch and bound*. En su procedimiento de resolución incorporan una de las relaciones de dominancia que se presentan más comúnmente en los problemas de optimización combinatoria: las simetrías (Bosch 1993, Kusiak et al. 1993, etc.). Las simetrías características del problema de tecnología de grupos las tienen en cuenta realizando una ordenación inicial, de manera lexicográfica o no, de las máquinas a asignar (M_1, M_2, M_3, ...); de esta manera, ya no se generan vértices equivalentes. Sea un problema en el que se han de asignar diversas máquinas a 3 celdas idénticas (C_1, C_2, C_3): en una primera etapa la máquina M_1 se asigna directamente a C_1 y en la segunda se asigna M_2 a C_1 o a C_2:



Eliminación de estados por dominancias (simetrías) para el problema de tecnología de grupos.

Cabe destacar que el tratamiento de las simetrías también se puede realizar en el momento de formular el problema, de esta forma el procedimiento de exploración ya no genera vértices equivalentes que podrían dar lugar a soluciones simétricas.

Por otro lado, Ibaraki (1988) propone la idea de verificar relaciones de dominancia sólo con un subconjunto de los vértices generados; según este mismo autor, esta táctica resulta ser más rápida pero puede llegar a ser menos efectiva.

3.5.1.8. Estrategias de exploración / selección.

Las estrategias de exploración de la arborescencia de búsqueda, también denominadas estrategias de selección del próximo vértice a explorar, ya han sido introducidas en el apartado 3.2. al comentar las funciones de evaluación. Así, para evaluar los vértices candidatos a ser explorados, ordenarlos y poder decir cuál es el considerado como mejor o peor, usualmente se utiliza una estrategia de selección que se puede considerar que incorpora una función de evaluación (también llamada indicador) que, a veces, utiliza información heurística relativa al problema. Como exponen Nemhauser & Wolsey (1989), como introducción se puede decir que los dos objetivos principales de esta fase son los siguientes:

a) seleccionar el vértice que con mayor posibilidad contiene la solución óptima, aunque no se pueda probar;

b) seleccionar el vértice que más rápidamente puede proporcionar una solución factible mejor que la solución preferible, lo que permite podar más eficazmente.

A diferencia de las fases anteriores del algoritmo *branch and bound* en las que se pretende identificar qué soluciones no pueden ser óptimas para eliminarlas de la enumeración, la estrategia de exploración determina únicamente en qué orden se generan y examinan los subproblemas, o, como se expone en Companys (1989a), cómo se hace explícita una porción del grafo implícito suficiente para que incluya por lo menos una solución óptima. De todas maneras, y aunque los algoritmos *branch and bound* siempre convergen en un número finito de pasos, su eficacia y el espacio de almacenamiento necesario dependen fuertemente de la estrategia de búsqueda seleccionada (Ibaraki 1988, Winston 1994a, Bjorndal et al. 1995, Gangonells & Gómez 1995, etc.).

Las estrategias de selección del próximo vértice a explorar no son únicas; de esta manera, y como señalan Nilsson (1971), Barr & Feigenbaum (1981), Hartnell (1986), Álvarez et al. (1988), Roca (1992), Kusiak et al. (1993), Jünger et al. (1994) y Grötschel & Lovász (1995) entre otros, según cuáles sean los criterios para elegir el siguiente

vértice a explorar, se pueden obtener múltiples versiones diferentes del algoritmo *branch and bound*, unas más eficientes que otras.

Son muchos los autores que exponen estrategias de selección del próximo vértice a explorar: Corominas et al. (1984), Álvarez et al. (1988), Ibaraki (1988), Nemhauser & Wolsey (1989), Cortés et al. (1993), Winston (1994a), Fernández & Companys (1995), Gangonells & Gómez (1995), Balas et al. (1996), Nagar et al. (1996), Ragsdale & Shapiro (1996), Roucairol (1996), Sierksma (1996), Johnson et al. (1997), Kubiak et al. (1997), etc. Algunos de estos criterios de selección ya han sido expuestos en el apartado 3.3. al explicar los procedimientos de búsqueda basados en la "fuerza bruta" y algunos otros lo serán en el apartado 3.6. al introducir los procedimientos de búsqueda usualmente descritos en el área de la inteligencia artificial. Cabe destacar de nuevo, que existen diversos procedimientos que han adoptado como rasgo diferenciador de los demás la estrategia de selección del próximo vértice a explorar, con lo cual no está claro si su nomenclatura diferente se refiere al procedimiento de búsqueda o a la estrategia de selección como parte integrante de dicho algoritmo de búsqueda.

A continuación se exponen tres estrategias de búsqueda descritas por Ragsdale & Shapiro (1996), que incorporan diversos paquetes comerciales de programación lineal mixta.

Sea un problema de minimizar, donde Z_{PL} es el valor de la cota inferior de un vértice, Z' el valor de la cota superior y Z^* el valor de la solución óptima. Todas estas estrategias seleccionan vértices, basándose en una estimación de la mejor solución entera factible que se podría obtener de cada problema candidato en la lista de vértices. Sean e^b_i, e^p_i y e^r_i la estimación de la mejor solución entera posible que se podría obtener del vértice i, utilizando respectivamente los estimadores "mejor proyección", "pseudo-coste" y "error relativo".

a) Estrategia de la mejor proyección. Se define el estimador del vértice i de la manera siguiente:

$$e^b_i = Z_{PLi} + p \cdot \sigma_i,$$

donde Z_{PLi} es el valor de la función objetivo relajada en el vértice i, p es el factor de proyección, y σ_i es la suma de las infactibilidades enteras en el vértice i ($\sigma_i = \sum_j \min(f_{ij}, 1 - f_{ij})$, donde f_{ij} representa la parte fraccional de la variable entera relajada j, en la solución del programa lineal en el vértice i). El factor de proyección p es un número que estima la cantidad por la que el óptimo relajado Z_{PLi} se espera que se degrade, por cada unidad de infactibilidad entera en el vértice.

El valor p es igual para todos los vértices estimados. Los paquetes comerciales proporcionan varios valores de p por defecto. Sin embargo, si se especifica una solución inicial Z´, se utiliza este valor para hallar el valor de p. Por ejemplo, en el paquete MIPII (Ragsdale & Shapiro 1996), se calcula como sigue:

$$p = \frac{min\left[0.40, \left|\left(Z' - Z_{PL_0}\right) / Z_{PL_0}\right|\right] x Z_{PL_0}}{\sigma_0}$$

También es costumbre recalcular el valor de p siempre que se encuentra una solución entera factible en el *branch and bound*. En el paquete MIPII se recalcula el valor de p para la solución entera del vértice k, mediante la siguiente expresión:

$$p = \frac{Z_{PL_k} - Z_{PL_0}}{\sigma_0}$$

Cuando el valor de p varía, cambia el valor estimado de todos los vértices representados por e^b_i; se puede observar que el valor e^b_i es una función lineal del valor de p, ya que tanto Z_{PLi} como σ_i son constantes. Esta claro que si se selecciona un vértice para ramificar basándose en el estimador e^b_i, el valor relativo de los vértices depende del valor de p. Sin embargo, p puede estar directa o indirectamente influenciado por Z´ y por el proceso de podado. De esta manera, y como ya se ha comentado, el uso de una solución inicial puede alterar el orden en el que son seleccionados los vértices en un algoritmo *branch and bound*, provocando la necesidad de un mayor número de vértices (Ragsdale & Shapiro 1996).

b) Estrategia del pseudo-coste. Se define el estimador del vértice i como sigue:

$$e^p_i = Z_{PLi} + \sum_j \min [PCL_j \cdot f_{ij}, PCU_j \cdot (1 - f_{ij})],$$

donde los valores PCL_j y PCU_j representan, respectivamente, el pseudo-coste (o penalización) inferior y superior para la variable j. PCL_j es una estimación de la cantidad por la que el óptimo del PL del vértice se espera que se degrade (por unidad de infactibilidad), al forzar la variable j al valor entero inferior de su valor en el vértice i. De forma similar, PCU_j es una estimación de la cantidad por la que el óptimo del PL del vértice se espera que se degrade (por unidad de infactibilidad), al forzar la variable j al valor del entero superior de su valor en el vértice i.

Por defecto, cuando se inicializa el *branch and bound* en el paquete MPSX (Ragsdale & Shapiro 1996), todos los pseudo-costes son nulos. La estimación de estas cantidades comienza a ser posible cuando se selecciona una variable de ramificación. Por tanto, contra más ramas adicionales se generan en el *branch and bound*, más y más pseudo-

costes son posibles para refinar los estimadores de los vértices. De esta manera, mientras una solución inicial puede reducir el número de ramas por la poda, simultáneamente puede alterar los estimadores de los vértices para futuras selecciones (Ragsdale & Shapiro 1996).

c) Estrategia del error relativo. Se define el estimador del vértice i como:

$$e_i^r = \frac{e_i^p - Z'}{Z_{PL_i} - Z'}$$

Como se describe en el punto anterior, es posible un cambio en el valor de e^p_i dependiendo del valor de Z´; esto hace que el valor de e^r_i también pueda cambiar en función del valor de Z´.

Una vez se ha seleccionado el vértice a explorar, a menudo también se ha de concretar la variable con la que se efectuará la separación. Aunque este paso puede ser diferente del de seleccionar el próximo vértice a ramificar, también se podría utilizar para este fin.

En la definición que hacen Balas et al. (1996) de su procedimiento *branch and cut* (expuesto en el apartado 3.5.8.), para decidir el vértice a ramificar se selecciona el de mejor cota; en cambio, para elegir la variable a separar se utiliza el criterio propuesto por Padberg & Rinaldi en 1991. Sean x_j el valor de las variables enteras que en la resolución del programa lineal -resultado de la relajación del programa matemático- no proporciona un valor entero; sean f_0 y f_1 respectivamente, el mayor valor de $x_j \leq 0{,}5$ y el menor valor $x_j \geq 0{,}5$; se define el conjunto de variables candidatas para ser seleccionadas como sigue:

$$I = \{ i \in (1, ..., p): f_0/2 \leq x_i \leq (1+f_1)/2 \},$$

y se selecciona la variable de I con el mayor coeficiente (en valor absoluto) en la función objetivo. En la misma línea argumental, Corominas et al. (1984) comentan que en el algoritmo de Balas se selecciona para ramificar el vértice más profundo, pero para elegir la variable a separar, se define un índice de imposibilidad de obtener una solución factible para cada variable candidata y selecciona la variable de menor índice. Corominas et al. (1984) exponen que también se podría haber tenido en cuenta únicamente la función objetivo y se podría haber seleccionado la variable no fijada de menor coeficiente en valor absoluto en la función objetivo (como expone Sierksma 1996, en programación lineal entera no está claro qué variable se debe elegir para ramificar y se suele seleccionar aquélla de mayor importancia en la función objetivo).

Otros criterios de selección de la variable a separar pueden ser los siguientes:

- la variable que posee el mejor descendiente de todos los posibles (Ibaraki 1988),
- la variable que posee el peor descendiente de todos los posibles, con la esperanza de que éste empeore tanto que no vuelva a ser necesario su exploración (Corominas et al. 1984 e Ibaraki 1988),
- la variable que posee el mejor conjunto promedio de vértices descendientes (Ibaraki 1988),
- la variable que consigue una máxima discriminación entre los vértices hijos resultantes, con el objetivo de no necesitar explorar los peores de ellos (Corominas et al. 1984 e Ibaraki 1988),
- la variable que tiene un hijo con un valor de su cota, obtenida con el empeoramiento que se obtendría al hacer una primera iteración del símplex, mejor (Corominas et al. 1984),
- la variable para la que el valor resultante de resolver el programa lineal asociado, está más cerca de tomar valores enteros (Ibaraki 1988),
- la variable más restrictiva, y, por consiguiente, susceptible de limitar más el proceso de exploración del árbol de soluciones (Fernández & Companys 1995),
- la variable que minimiza la dificultad de obtener soluciones factibles (Ibaraki 1988).

Aunque está ampliamente extendido, en la mayoría de ocasiones explorar el vértice de mejor cota no es determinante para los procedimientos *branch and bound*; muchas veces es mejor utilizar predictores diseñados especialmente para esta misión que incorporan otras informaciones heurísticas o estructurales acerca del problema a resolver. Por otro lado cabe destacar que, aunque de forma tímida, Pearl (1984) e Ibaraki (1988) describen procedimientos en los que las estrategias generales de búsqueda comienzan a tener carácter dinámico al utilizar dos o más criterios de selección, en función del estado de evolución de la búsqueda o en función de los subconjuntos de vértices entre los que se realice dicha selección (aspecto de gran importancia en la función general de evaluación y selección de vértices diseñada en el apartado 7.4.4.).

3.5.1.9. Clasificación de los algoritmos *branch and bound* para problemas de permutación.

Kohler & Steiglitz (1974) exponen la caracterización de los algoritmos de enumeración implícita *branch and bound* para problemas de permutación (problemas de optimización combinatoria cuyo conjunto de soluciones posibles está formado por la permutación de un conjunto de valores y que incluyen a los problemas de *flow-shop* y de asignación cuadrática como casos especiales), en términos de una tupla de seis parámetros (B_p,S,E,D,L,U). De esta forma se puede realizar una clasificación general de este tipo de

algoritmos, así como obtener un algoritmo general que permite obtener otros en función de la selección del valor concreto de cada uno de estos parámetros.

En la caracterización presentada, (B_p,S,E,D,L,U), cada uno de estos seis parámetros de clasificación representa lo siguiente:

- B_p es la regla de ramificación para el problema de permutación, que genera descendientes inmediatos al dividir el subproblema tratado en subconjuntos disjuntos. La generación de descendientes inmediatos, que es total, se realiza en orden lexicográfico.

- S es la regla de selección del próximo vértice a ramificar de entre el conjunto de vértices activos actuales. Según estos autores, aunque existen muchas otras variaciones posibles, las reglas de selección más comunes son las siguientes:

  - S = LLB (*Least Lower-Bound Rules*): selección del vértice activo con menor cota inferior; en caso de empate se selecciona el primer vértice generado, LLB_{FIFO}, o el último, LLB_{LIFO}.
  - S = FIFO (*First-In-First-Out Rule*): selección del vértice activo que fue generado primero.
  - S = LIFO (*Last-In-First-Out Rule*): selección del vértice activo que fue generado último.

- D es la función de dominancia de vértices. Se dice que un vértice v_1 domina a otro vértice v_2, v_1Dv_2, si la mejor solución completa incluida en v_1, x_1, no es peor que la mejor solución completa incluida en v_2, x_2: $f(x_1) \leq f(x_2)$, en problemas de minimización y siendo f la función objetivo.

- L es la función de acotado inferior, que debe cumplir las propiedades siguientes:

  - Si v_2 es un descendiente de v_1, entonces $L(v_2) \geq L(v_1)$, en problemas de minimización.
  - Para cada solución completa v, L(v) = f(v), siendo f la función objetivo.

- U es el valor de la solución preferible.

- E es el conjunto de reglas para utilizar la función de dominancia D y el coste de la solución preferible U, en la poda tanto de vértices acabados de generar como de vértices actualmente activos. Sea v_r el vértice actual de ramificación, v_{rd} un descendiente inmediato de v_r y v_a un vértice de la lista actual de vértices activos. E representa varios subconjuntos de las siguientes cuatro reglas:

(i) U/DBAS (*upper-bound tested for dominance of descendants of branching node and members of currently active set*). Si $L(v_{rd}) > U$ entonces v_{rd} es podado después de ser generado; si $L(v_a) > U$ entonces v_a es eliminado de la lista de vértices activos.
(ii) AS/DB (*active node set tested for dominance of descendants of branching node*). Se comprueba si alguno de los vértices de la lista de vértices activos domina al vértice acabado de generar: si $v_a D v_{rd}$ entonces v_{rd} es podado después de ser generado.
(iii) BFS/DB (*branched-form node set tested for dominance of descendants of branching node*). Se comprueba si alguno de los vértices previamente generados domina al vértice acabado de generar: si $v_p D v_{rd}$ entonces v_{rd} es podado después de ser generado.
(iv) DB/AS (*descendants of branching node tested for dominance of currently active node set*). Se comprueba si el vértice acabado de generar domina a alguno de los vértices de la lista de vértices activos: si $v_{rd} D v_a$ entonces v_a es eliminado de la lista de vértices activos.

Si E contiene reglas AS/DB y DB/AS, entonces el conjunto de vértices activos puede depender del orden en el que estas reglas son aplicadas.

Estos autores comentan que la caracterización propuesta permite describir un algoritmo general que puede ser extendido para describir diversos procedimientos (por ejemplo, con B_p, S = FIFO, E = {AS/DB, DB/AS}, D, L, U, se pueden describir técnicas de programación dinámica) e incluso algoritmos *branch and bound* heurísticos. Por otro lado, cabe destacar que la clasificación presentada para problemas de permutación es clara y muy compacta.

3.5.2. *Branch and search.*

Djerdjour (1997) resuelve el problema de la mochila multidimensional cuadrático, aplicando un procedimiento enumerativo de ramificación y acotación, que denomina *branch and search*. Esta técnica de búsqueda consiste en un *branch and bound* típico, con una estrategia de búsqueda primero en profundidad.

3.5.3. *Branch and relax.*

Hentenryck (1995), un investigador procedente del área de la inteligencia artificial, presenta el término *branch and relax* como un término general para referirse tanto a los procedimientos *branch and bound* (expuestos en el apartado 3.5.1.) como a las técnicas *branch and prune* (introducidas a continuación en el apartado 3.5.5.). Concretamente dice: "*Branch and relax (often called branch and bound for optimization problems and*

branch and prune for decision problems) is a well-known technique to solve combinatorial search problems. It consists ..." (Hentenryck 1995, p. 569).

3.5.4. *Branch and reduce.*

Ryoo & Sahinidis (1996) utilizan el término *branch and reduce* para referirse a un procedimiento de exploración del espacio de soluciones factibles mediante la ramificación, acotado y podado, que, concretamente, definen para resolver problemas de optimización global. De todas formas, y como ellos mismos comentan, aunque el *branch and reduce* resuelve problemas de optimización global lo interesante es intentar adaptarlo para resolver problemas de optimización combinatoria.

Según Ryoo & Sahinidis (1996) *branch and reduce* toma su nombre de la combinación del análisis de intervalos y dualidad, con conceptos de *branch and bound. Branch and reduce* integra un *branch and bound* tradicional junto a una amplia variedad de tests de reducción de rangos. Estos tests (que presentan inecuaciones válidas basadas en la optimalidad y criterios de factibilidad, y técnicas de reducción de rangos utilizadas para reducir el tamaño del espacio de búsqueda en problemas de optimización global) se aplican a cada subproblema en el árbol de búsqueda, para contraer el espacio de estados y reducir el *gap* de la relajación. Otro componente crucial es la implementación de técnicas heurísticas para la resolución aproximada de los subproblemas de optimización, con el propósito de obtener una mejor solución preferible del problema. Por último, y según los mismos autores, se incorpora un conjunto de esquemas de ramificación que aceleran la convergencia de las estrategias de ramificación standard. La estrategia de selección incorporada por Ryoo & Sahinidis en su procedimiento es la búsqueda primero el de mejor cota (denominada por éstos *best bound first*).

Siendo $\overline{Z}$ el valor de la solución preferible y con el objetivo de minimizar, el algoritmo *branch and reduce* se puede describir brevemente como sigue:

Paso 0. Inicialización.

Sea la cota superior $\overline{Z} = +\infty$.

Poner el vértice raíz en la lista Activos.

Paso 1. Finalización.

Eliminar todos los vértices de la lista Activos tales que su cota inferior sea $\geq \overline{Z}$.

Si Activos = $\varnothing$, parar y la solución preferible es óptima.

Paso 2. Selección de subproblemas.

Seleccionar un vértice de la lista Activos según la estrategia de selección y eliminarlo de la lista.

Paso 3. Pre-proceso.

Ajustar las variables acotadas del subproblema seleccionado usando reducción de rango basado en la factibilidad.

Paso 4. Acotado.

Resolver el subproblema seleccionado o acotar su solución.

Si la solución encontrada es factible y su valor de la función objetivo $< \overline{Z}$, tomarla como solución preferible y actualizar $\overline{Z}$; ir a paso 5.

Si el valor de su cota inferior es $\geq \overline{Z}$, ir a paso 1.

Paso 5. Acotado superior opcional.

Aplicar heurísticas de búsqueda local para encontrar una solución factible mejor; si se obtiene, tomarla como solución preferible y actualizar $\overline{Z}$.

Paso 6. Post-proceso.

Ajustar las cotas de las variables usando técnicas de reducción de rangos.

Si la reducción de rango es exitosa en al menos una de las variables del subproblema, entonces reconstruirlo usando las nuevas variables acotadas e ir a paso 4.

Paso 7. Separación.

Ramificar el subproblema obteniendo un conjunto de nuevos subproblemas, incluirlos en la lista Activos e ir a paso 1.

De esta manera, las principales aportaciones de este procedimiento las constituyen los pasos 3, 5 y 6.

3.5.5. *Branch and prune.*

Wagner (1975) y Hentenryck et al. (1997) comentan el algoritmo *branch and prune*, aunque de forma no coincidente: mientras que Wagner expone que el *branch and bound* también se puede llamar *branch and prune*, Hentenryck et al. definen esta técnica como un nuevo procedimiento de búsqueda en el espacio de estados.

Concretamente, Hentenryck et al. (1997) consideran que su procedimiento *branch and prune*, caracterizado por los mismos autores como un método para resolver problemas de búsqueda global, es una técnica diferente a lo que habitualmente se puede conocer como *branch and bound* "... *Operationally, Newton* [el nombre del paquete de software] *uses a branch and prune algorithm which was inspired by the traditional branch and bound approach used to solve combinatorial optimization problems* ...", Hentenryck et al. (1997), p. 797. De todas formas, el procedimiento *branch and prune* que definen es una técnica de ramificación y acotación para permitir la poda del espacio de búsqueda, al que, en este caso, se le incorporan técnicas de propagación de restricciones en cada uno de los vértices de la arborescencia, y, más concretamente, técnicas de reducción de los intervalos de los valores definidos para cada variable de forma que se cumpla una

condición de consistencia local llamada caja-consistencia (en el apartado 3.8.2. se introducen técnicas de reducción de este tipo provenientes del área de la inteligencia artificial). Por otro lado, Granvilliers (1998) describe de forma semejante un algoritmo *branch and prune* aplicado en la resolución de sistemas polinomiales no lineales.

Como se puede comprobar, las descripciones de los procedimientos *branch and prune* y *branch and reduce* para la resolución de problemas de optimización global, muestran que ambos algoritmos son altamente coincidentes. Por otro lado, este tipo de procedimientos son usualmente denominados *interval branch and bound algorithms*, lo que indica que, en realidad, pueden ser considerados algoritmos *branch and bound* que incorporan técnicas de reducción de intervalos.

3.5.6. *Depth-first / Breadth-first / Best-first / ... branch and bound.*

Kumar (1990), Sarkar et al. (1991), Korf (1993) y Zhang & Korf (1995), comentan que la técnica conocida como *depth-first branch and bound* (DFBnB) es un procedimiento *branch and bound* en el que se selecciona para separar el vértice generado más recientemente, incorporando, por tanto, la estrategia de selección *depth first* (ya definida en el apartado 3.3.3. al exponer la búsqueda en profundidad). Zhang & Korf (1995) matizan que para aumentar la eficiencia de este procedimiento, los vértices hijos generados del último vértice separado deben ser explorados en orden creciente del valor de una función de evaluación.

Kumar (1990) también define el procedimiento denominado *best-first branch and bound*, como un algoritmo *branch and bound* en el que la estrategia de selección de los vértices a separar consiste en seleccionar el vértice de mejor valor de un indicador (que para Kumar, y en este caso, está constituido por el valor de la cota inferior), así se utiliza la estrategia *best first* (ya definida en el apartado 3.3.7. al exponer la búsqueda primero el mejor); como señala Kumar, si la función de acotado da una buena aproximación el procedimiento es muy eficiente, pero sino es muy ineficiente.

A pesar de que únicamente se han definido como tales los procedimientos *depth-first branch and bound* y *best-first branch and bound*, se puede considerar con total seguridad que existen muchos otros procedimientos de búsqueda a los que se les ha incorporado un procedimiento de acotado y podado como el *branch and bound*, o que dichas estrategias han sido incorporadas a un procedimiento *branch and bound*. Así por ejemplo, no resulta difícil imaginar la existencia de un procedimiento *branch and bound* en el que la estrategia de selección de los vértices a ramificar sea la utilizada en la búsqueda en anchura (con lo que se obtendría un *breadth-first branch and bound*), la de la búsqueda primero el de menor coste (*uniform-cost branch and bound* o *cheapest-first branch and bound*), la de la búsqueda primero el de mejor cota (*best-bound branch and*

bound), la utilizada en diversos procedimientos de búsqueda usualmente descritos en el área de la inteligencia artificial y expuestos en el apartado 3.6. (la del algoritmo *iterative deepening* (*iterative-deepening branch and bound*), la del algoritmo A* (A* *branch and bound*), la del algoritmo IDA* (IDA* *branch and bound*), etc.), e incluso, por qué no, la incorporación de un *branch and bound* en los procedimientos de búsqueda basados en programación dinámica (*dynamic-programming branch and bound*).

Por último, cabe destacar que para la resolución de problemas de dimensiones interesantes, hoy en día la mayoría de procedimientos diseñados suelen incorporar indefectiblemente procedimientos de acotado y podado.

3.5.7. *Branch and bound* con estrategia primero el de mejor cota local (*locally best bound rule*).

Liao (1994) propone una nueva estrategia de selección de los vértices a explorar, denominada regla el de mejor cota local (*locally best bound rule*). Esta estrategia selecciona para separar el vértice de mejor cota global entre un conjunto reducido de vértices que, al igual que ocurre en el caso de la búsqueda primero el de mejor cota, es el de mejor valor de $g(n) + h(n)$, donde $g(n) = g^*(n)$ y $h(n)$ nunca sobrestima a $h^*(n)$, o, para procedimientos en los que las funciones $g(n)$ y $h(n)$ no tienen sentido, el de mejor valor de $l(n)$, donde $l(n)$ es una cota global de la mejor solución alcanzable desde el vértice n.

La particularidad de esta estrategia consiste que, en este caso, el movimiento de avance no se realiza seleccionando para ramificar el vértice que parece más prometedor de entre todo el conjunto de vértices abiertos que se tiene hasta ese momento, sino de entre los de un subconjunto de dichos vértices abiertos.

Sea un problema de optimización que debe ser minimizado, la idea de la búsqueda el de mejor cota local es la siguiente. En cualquier momento en el que los vértices guardados en la memoria central exceden un rango pre-especificado (Liao cita como ejemplo 200 vértices), se ordenan en una lista en forma no decreciente de sus cotas inferiores y se divide ésta en dos partes: la parte que contiene los primeros vértices de la lista (como por ejemplo cita Liao, un porcentaje del 30%) se continúa almacenando en la memoria central, mientras que el resto de vértices se guarda en una memoria auxiliar como el disco duro de un ordenador. El procedimiento de ramificación, acotación y poda (*branch and bound*) continúa manipulando únicamente los vértices existentes en la memoria central hasta que ésta está vacía: todos los vértices han sido eliminados o guardados en la memoria auxiliar; en este instante, uno de los conjuntos de vértices almacenados en la memoria auxiliar es seleccionado y estos vértices son incorporados a la memoria central para continuar con el procedimiento *branch and bound*. El proceso

continúa hasta que no existe ningún conjunto de vértices guardados en la memoria auxiliar.

Siendo $\overline{Z}$ el valor de la solución preferible, el algoritmo se puede describir brevemente como sigue:

Etapa 1. Inicialización.

Sea la cota superior $\overline{Z} = \infty$.

Etapa 2. Búsqueda del vértice de ramificación.

Entre todos los vértices de la memoria central, seleccionar el de menor valor de la cota inferior.

Etapa 3. Ramificación, acotado y podado.

Ramificar totalmente el vértice seleccionado y descartarlo de la arborescencia; calcular una cota inferior de los nuevos vértices generados: si se obtiene una nueva solución factible, calcular su valor objetivo, compararlo con $\overline{Z}$ y si tiene un valor menor retenerla como solución preferible y actualizar $\overline{Z}$; podar aquellos vértices de la memoria central cuya cota inferior sea mayor o igual que $\overline{Z}$.

Etapa 4. Actualizar la lista de vértices de la memoria central.

Eliminar los vértices podados y guardar los recién generados.

Etapa 5. Test.

(a) Si el número de vértices en la memoria central es mayor que el rango especificado, ir a la etapa 6; si no, continuar.

(b) Si hay vértices en la memoria central, ir a la etapa 2; si no, continuar.

(c) Si hay vértices guardados en la memoria auxiliar, ir a la etapa 7; si no, ir a la etapa 8.

Etapa 6. Ordenar y guardar vértices.

Ordenar los vértices de forma no decreciente del valor de su cota inferior; dividir la lista de vértices en dos partes mediante un porcentaje especificado; mantener la primera parte en la memoria central y guardar la segunda parte en la memoria auxiliar. Ir a la etapa 2.

Etapa 7. Recuperar un conjunto de vértices.

Descartar los conjuntos de vértices de la memoria auxiliar cuyo primer integrante tenga una cota inferior mayor (o, aunque el autor no lo especifica, igual si sólo se busca una solución óptima) que $\overline{Z}$; entre los conjuntos de vértices restantes, seleccionar aquél cuyo primer vértice tenga el menor valor de la cota inferior; copiar en la memoria central aquellos vértices de dicho conjunto cuya cota inferior sea menor (o igual) que $\overline{Z}$ y eliminar los vértices restantes de este conjunto. Ir a la etapa 2.

Etapa 8. Parar.

Liao (1994) comenta algunas características de la técnica descrita. Aunque dentro de los conjuntos de vértices no seleccionados para ser incorporados en la memoria central es probable que existan vértices que pueden ser podados, esto no se hace ya que consume mucho tiempo; en principio, sólo se eliminan conjuntos completos de vértices. Por otro lado, para poder podar rápidamente dichos conjuntos de vértices o parte de un conjunto en el momento de incorporarlos a la memoria central, es recomendable guardarlos en orden no decreciente del valor de su cota inferior. Si el rango especificado es suficientemente grande para que la memoria central pueda acomodar en todo momento la parte generada de la arborescencia, el procedimiento descrito es un *branch and bound* que utiliza como estrategia de exploración la regla *best bound.*

Uno de los principales problemas a resolver en el momento de utilizar este procedimiento consiste en determinar un valor apropiado para el rango de vértices que se han de conservar en la memoria central. Intuitivamente el rango debería ser tan grande como permitiera la memoria central, pero, según Liao, esta intuición es incorrecta y para aclararlo enumera las ventajas de utilizar un rango grande y uno pequeño.

Las ventajas de utilizar un rango grande son las siguientes:

1.- Como el vértice de ramificación es seleccionado entre los de la memoria central, para un rango grande tiende a tener menor valor de la cota; por tanto se tiende a ramificar menos vértices y el tiempo de computación se reduce.
2.- Cuando el número total de vértices es menor que el rango, no se precisa ordenar ni guardar vértices en la memoria auxiliar; esto representa un ahorro de tiempo de ordenación y de acceso a la memoria auxiliar.
3.- Como la poda se realiza en la memoria central, cuando se encuentra una solución mejor se pueden podar muchos más vértices con un decremento en el tiempo de búsqueda y de ordenación.

De todas formas Liao (1994) también describe ventajas al usar rangos pequeños:

1.- Como el vértice de ramificación es seleccionado sólo entre los de la memoria central, un rango pequeño comporta menos tiempo de búsqueda; esto representa un gran ahorro de tiempo de computación ya que la operación de búsqueda se utiliza frecuentemente.
2.- La operación de actualización se realiza cuando se encuentra una solución factible mejor y los vértices son podados; como esta operación se realiza en los vértices de la memoria central, tardará menos contra menor sea el rango.
3.- Cada vez que se deben guardar vértices en la memoria auxiliar, se requiere una operación de ordenación; esta operación requiere menos tiempo para el mismo

número total de vértices cuando el rango es pequeño, ya que es de orden O(n · log n).

De todas formas y tanto para determinar el mejor valor del rango, como el del porcentaje de vértices que se retienen en la memoria central (el otro gran problema a resolver cuando se utiliza esta técnica), Liao (1994) lo hace de forma empírica. Por ejemplo, para el algoritmo de Balas utiliza rangos de 3.000, 1.000, 600, 200 y 50 vértices, reteniendo en la memoria central el 10, 30 o el 50% de la lista de vértices; en este caso Liao comenta que parece que funciona mejor reteniendo el 30% de los vértices de la lista y trabajando con un rango de 200 o 50 vértices. Para este mismo algoritmo, Liao (1994) comenta que, según muestran algunos resultados experimentales, la aplicación de esta estrategia de selección no sólo hace que se utilice mucha menos memoria central que con la regla *best bound*, sino que además en muchos casos se utiliza menos cantidad de computación.

3.5.8. *Branch and cut*[4].

Como exponen Bjorndal et al. (1995), en la resolución de problemas de optimización combinatoria utilizando técnicas enumerativas, el procedimiento de acotado es vital en el desarrollo de un algoritmo eficiente. La actuación de los procedimientos de *branch and bound* suele proporcionar buenos resultados cuando las relajaciones proporcionan cotas muy cercanas al óptimo, pero en los casos en los que se presenta un *gap* importante entre la cota y la solución óptima. estas técnicas pueden resultar poco eficientes.

Bjorndal et al. (1995) exponen de forma actualizada la combinatoria poliédrica, un campo de investigación cuyo objetivo es el de proveer de un conjunto finito de restricciones lineales que encierren al conjunto de soluciones posibles en un poliedro -en realidad, en un politopo o poliedro (politopo acotado) mínimo)- hallando posteriormente y con programación lineal, un vértice óptimo. En la práctica se presentan dos problemas: por un lado, es extremadamente difícil conocer todas las restricciones lineales necesarias para definir el poliedro mínimo (teoría poliédrica), y, por otro, el número de restricciones es exponencial y aunque sólo un número polinomial de éstas son necesarias para definir la solución óptima, no se sabe cuáles son (computación poliédrica).

Las consideraciones anteriores han convergido en aproximaciones basadas en planos de corte, como el *branch and cut*.

[4] Como exponen Jünger & Thienel (1998), en Caprara & Fischetti (1997) se muestra una reciente y completa bibliografía sobre algoritmos *branch and cut*.

Como relatan Hoffman & Padberg (1996) y Jünger & Thienel (1998), los algoritmos de planos de corte tiene su origen en 1958, cuando Gomory desarrolla un algoritmo de planos de corte para problemas de programación entera (según Gass & Harris 1996, se puede definir "corte de Gomory" como una restricción lineal que es incluida en un problema de programación lineal, para reducir el espacio de soluciones sin eliminar ningún valor entero). Como describen brevemente Jünger et al. (1994), Grötschel & Lovász (1995), Hoffman & Padberg (1996) y Sierksma (1996) entre otros, el método de planos de corte de Gomory consta de dos fases iterativas:

Fase 1. Programación lineal: resolución del programa lineal resultante de relajar la condición de integridad de las variables; si en la solución óptima del programa lineal las variables que tienen carácter entero lo son, ya se ha alcanzado la solución óptima del problema original.

Fase 2. Planos de corte: se resuelve un problema de identificación de facetas, cuyo objetivo consiste en encontrar una inecuación lineal que elimine la solución fraccional lineal encontrada, pero sin eliminar ninguna solución de la región factible original[5]. Como Bjorndal et al. (1995) matizan, últimamente se está trabajando en buscar cortes que no sólo eliminen soluciones no factibles (fraccionales en programación lineal entera), sino también soluciones factibles no interesantes.

El algoritmo continúa hasta que, o se encuentra una solución factible, o el programa lineal es infactible, o no se pueden generar más cortes. Como ya se ha comentado, la idea consiste en reducir progresivamente el espacio de soluciones, de forma que la solución entera óptima corresponda a un punto extremo del espacio de soluciones reducido. Considérese el siguiente programa lineal binario que se presenta en Johnson et al. (1997):

$$[\text{MAX}]\ Z = 115 \cdot x_1 + 60 \cdot x_2 + 50 \cdot x_3 + 30 \cdot x_4$$
$$93 \cdot x_1 + 49 \cdot x_2 + 37 \cdot x_3 + 29 \cdot x_4 \leq 111$$
$$x_j \in \{0, 1\}.$$

La solución óptima del programa lineal es $x^* = (74/93, 0, 1, 0)$. Como que $x_1 = 1$ implica que todas las otras variables son iguales a 0 y como mucho dos de las otras variables pueden ser iguales a 1, la restricción $2 \cdot x_1 + x_2 + x_3 + x_4 \leq 2$ es una restricción válida; además, ésta es un corte violado ya que no es satisfecha por x^*. Incluyendo esta restricción en el programa lineal, la cota resultante es más ajustada.

[5] En Corominas et al. (1984) se describe el procedimiento concreto de generación de planos de corte de Gomory.

Jünger et al. (1994) exponen el procedimiento de una forma más general: se comienza con un pequeño subconjunto de las restricciones de un problema de optimización lineal, cuyo conjunto de restricciones es demasiado grande para ser representado explícitamente, y se halla el óptimo sujeto a dichas restricciones; se comprueba si alguna de las restricciones no consideradas no se satisface; si existe alguna restricción violada, se adiciona al programa lineal y se resuelve de nuevo; si no se viola ninguna restricción, la solución obtenida también resuelve el programa original. Cabe destacar que el procedimiento no requiere una lista explícita de todas las restricciones que definen el programa original, únicamente requiere un procedimiento que identifique inecuaciones que son validas para el programa original y que son violadas por la solución actual.

Con el objetivo de asegurar la resolución óptima de un problema de optimización combinatoria, la generación de planos de corte se puede incorporar a un procedimiento de *branch and bound*, con el objetivo de obtener cotas más ajustadas al reducir el conjunto de soluciones factibles en el programa lineal sin eliminar ninguna solución válida para el problema original (Corominas et al. 1984), obteniéndose de esta manera el algoritmo llamado *branch and cut*. Como exponen Jünger et al. (1994), la primera combinación de inecuaciones válidas y métodos de *branch and bound* fue realizada por Miliotis en 1976 para el problema TSP; el uso de facetas que definen planos de corte y el uso de generación automática de planos de corte en combinación con *branch and bound*, fue formulado y publicado en 1984 por Grötschel et al. para un problema de ordenación lineal; y, por otro lado, el término *branch and cut* fue introducido por Padberg y Rinaldi en 1991 para resolver el TSP.

Diversos autores han utilizado y definido el procedimiento *branch and cut* (Hoffman & Padberg 1993, Jünger et al. 1994, Escudero et al. 1995a, Fischetti et al. 1995, Grötschel & Lovász 1995, Saltzman 1995, Balas et al. 1996, Hoffman & Padberg 1996, Johnson et al. 1997, etc.), y, aunque presentan algunas diferencias, se puede definir el procedimiento básico de *branch and cut* como lo hacen Johnson et al. (1997).

Para Johnson et al. (1997), el procedimiento *branch and cut* es simplemente la modificación del algoritmo *branch and bound*, en el que, después de solucionar el programa lineal y no habiéndose podado el vértice, si se encuentran una o más restricciones violadas (planos de corte), se incluyen en la formulación del programa lineal y éste es resuelto de nuevo; si no se encuentra ninguna, se ramifica. De esta manera, la generación automática de cortes no sólo se realiza antes de comenzar el *branch and bound*, sino en cada vértice del árbol de enumeración; además, y para que este procedimiento tenga éxito, es importante que los cortes generados en un vértice del árbol sean válidos para todos los demás vértices, lo que ha limitado el tipo de cortes que

se han utilizado en *branch and cut* (Jünger et al. 1994, Saltzman 1995 y Balas et al. 1996).

Otros autores incorporan diversos refinamientos, aunque son características que también pueden ser incorporadas en cualquier otro esquema de ramificación y acotación, mejorándose, presumiblemente, los resultados obtenidos. Según Fischetti et al. (1995) la ejecución del *branch and cut* mejora si se es capaz de encontrar pronto "buenas" soluciones factibles: proponen un procedimiento heurístico que primero obtiene una solución factible a partir de la última solución parcial, para, posteriormente, aplicarle procedimientos de refinado. Para Hoffman & Padberg (1996) los principales componentes del *branch and cut* son los procedimientos de reformulación automática, heurísticas que proporcionan buenas soluciones enteras factibles, procedimientos de fijación de variables (por implicaciones lógicas o de coste reducido) y el hecho, cada vez más importante, de aprovechar las estructuras y características propias del problema, además de los procedimientos intrínsecos de generación de planos de corte. Por su parte, Escudero et al. (1995a) utilizan un procedimiento de *branch and cut* en el que explotan técnicas de preproceso: procedimientos de reducción del tamaño del problema vía identificación de restricciones redundantes y fijación de variables, y procedimientos de reforzamiento de las condiciones del problema; de igual manera que Hoffman & Padberg (1996), también destacan la importancia de obtener propiedades que permitan fijar nuevas variables, detectar infactibilidades e incluso identificar nuevas propiedades. Por su parte, Jünger et al. (1994) incluyen el uso de diversas heurísticas para calcular el primer valor de la solución preferible, procedimientos de cálculo de soluciones factibles en cada vértice aprovechando la fase de planos de corte (explorando las soluciones fraccionales del programa lineal), técnicas de optimización local para mejorar la solución preferible obtenida por el procedimiento anterior (con un control del tiempo dedicado a la optimización local fijo, según un número predeterminado de iteraciones, o mediante una estrategia dinámica en función de la solución obtenida del programa lineal respecto a la mejor solución factible, etc.), procedimientos de abandono de la fase de generación y adición de planos de corte si la cota superior no disminuye significativamente, técnicas para fijar variables basándose en sus costes reducidos y en implicaciones lógicas desprendidas de explorar las estructuras particulares del problema, etc.

Según la opinión de Johnson et al. (1997), el *branch and cut* generaliza, tanto los algoritmos de planos de corte puros en los que los cortes son incluidos en el vértice raíz hasta que se encuentra una solución óptima del problema, como el *branch and bound*. En esta tesis se argumenta la opinión contraria, siendo el procedimiento *branch and bound* el que generaliza a todos los demás.

Como exponen Balas et al. (1996), a grandes rasgos y para un problema de minimización, el algoritmo *branch and cut* está formado por las siguientes fases:

Fase 1. Iniciación.
Fase 2. Selección del vértice.
Fase 3. Cálculo de la cota inferior.
Fase 4. ¿Ramificar o generar cortes? (decisión de la que depende en gran parte el éxito del algoritmo).
Si se generan cortes, ir a la fase 5.
Si se ramifica, ir a la fase 6.
Fase 5. Generar planos de corte válidos para el programa original, pero violados por la solución de la relajación. Ir a la fase 3.
Fase 6. Ramificación. Ir a la fase 2.

Además, y como estos mismos autores comentan, en el esqueleto fundamental del procedimiento *branch and cut* descrito, se pueden incorporar, como ya se ha comentado, diversos refinamientos: uso de heurísticas, preproceso, fijación de variables, etc. A continuación se describe un ejemplo concreto. Hoffman & Padberg (1993) resuelven un problema de horarios de tripulaciones aéreas mediante un *branch and bound* que genera planos de corte, además de utilizar procedimientos de reformulación, heurísticas y programación lineal. Concretamente, el procedimiento *branch and cut* propuesto para resolver el programa lineal binario resultante tiene cuatro componentes básicos: un preprocesador que ajusta y reduce la formulación (se transforma el problema en una representación equivalente fijando variables, eliminando restricciones que resultan ser redundantes, etc.), una heurística que proporciona buenas soluciones factibles rápidamente, un procedimiento de generación de cortes que ajusta el programa lineal relajado y una estrategia de ramificación que selecciona la variable de separación y determina el árbol de búsqueda.

La siguiente figura muestra un diagrama de flujo del *branch and cut* diseñado por Hoffman & Padberg (1993), donde $\bar{z}$ es el valor de la solución preferible y Z_{PL} es el valor de la solución del programa lineal.

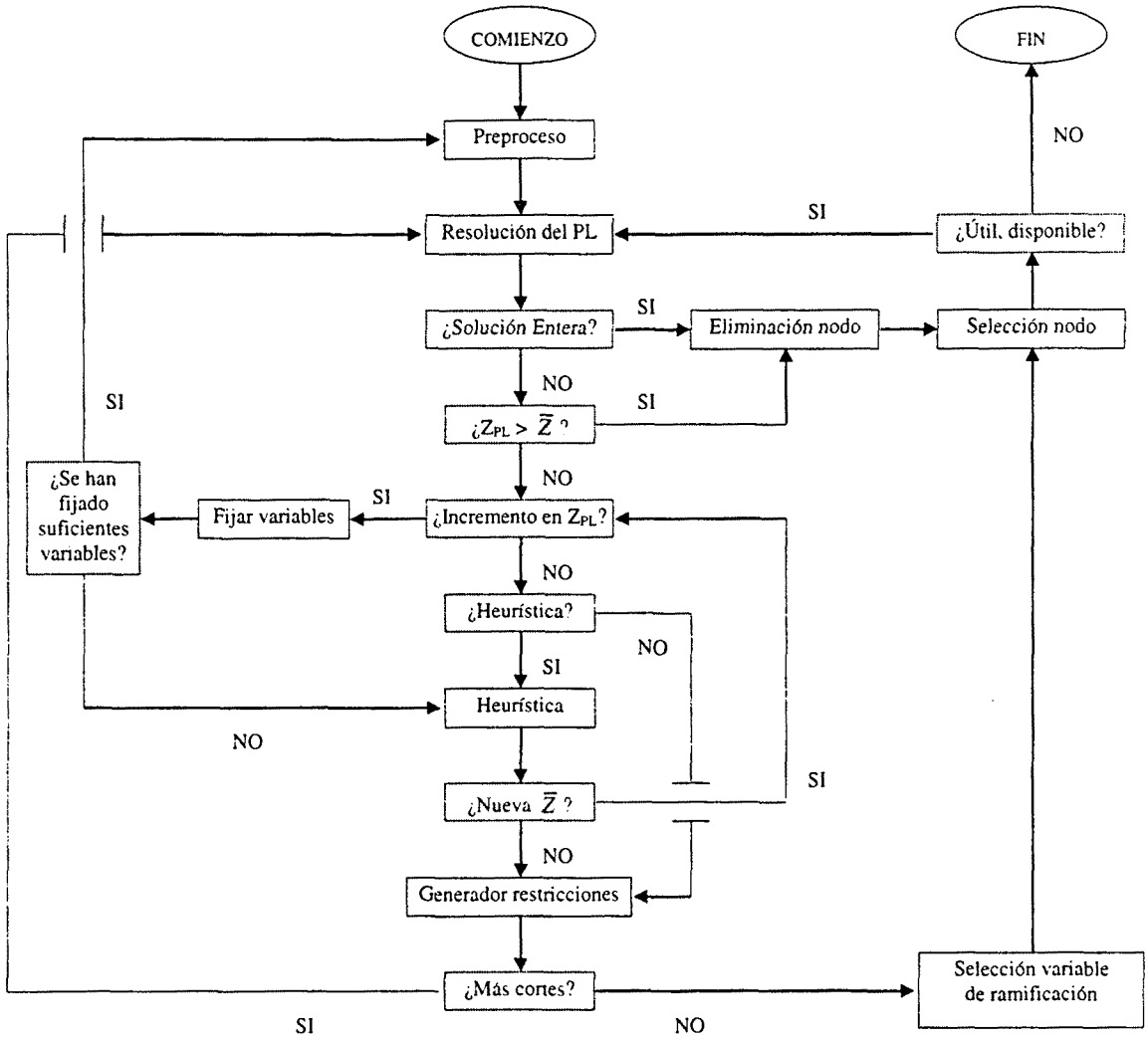


Diagrama de flujo del procedimiento *branch and cut*, de Hoffman & Padberg (1993).

Del gráfico anterior cabe destacar las subfases *¿Incremento en Z_{PL}?, Fijar variables, ¿Se han fijado suficientes variables?*. Aunque no queda claramente especificado en el texto, se puede suponer que "¿Incremento en Z_{PL}?" se refiere a algún procedimiento que tiene en cuenta el coste reducido, que permite asegurar que una variable nunca estará en la base ya que entonces $Z_{PL} > \overline{Z}$; así, siempre que es posible se utiliza un procedimiento para fijar variables que puedan tener implicaciones para otras, de forma que si se fija un % específico de las variables que quedan por fijar, el control se envía de nuevo al preproceso y si no a la heurística.

Cuando se alcanza la fase *Generador restricciones*, ya se dispone de una buena formulación del problema y se ha resuelto el programa lineal, y en este momento se generan y añaden nuevas restricciones basadas en la teoría de poliedros que dan cortes poliédricos. El programa lineal es resulto de nuevo entrando en el bucle representado en la figura anterior.

Con el objetivo de trabajar con arborescencias de tamaño reducido, Hoffman & Padberg (1993) también proponen, aunque sin especificar cómo, una estrategia de ramificación que varía dependiendo de las características de la solución actual del programa lineal.

Entrando en un mayor grado de detalle y también a modo de ejemplo, a continuación se exponen las particularidades del algoritmo *branch and cut* descrito por Balas et al. (1996):

- Para seleccionar el vértice a ramificar se utiliza una estrategia de control *best-bound* (de todas formas, Jünger et al. (1994) comentan que existen diversos procedimientos: búsqueda en anchura, en profundidad, *best-bound*, etc.).

- La selección de la variable a separar se realiza utilizando el criterio propuesto por Padberg & Rinaldi en 1991:

  - sea x la solución del último programa lineal;
  - determinar el mayor valor de $x_j \leq 0{,}5$ y el menor valor $x_j \geq 0{,}5$; sean f_0 y f_1 dichos valores respectivamente;
  - definir el conjunto de variables candidatas para ser seleccionadas como:

    $$I = \{ i \in (1, ..., p): f_0/2 \leq x_i \leq (1+f_1)/2 \};$$

  - seleccionar la variable $\in$ I con el mayor coeficiente (en valor absoluto) en la función objetivo.

Por su parte, Jünger et al. (1994) comentan una mayor variedad de estrategias de selección de la variable de ramificación. Sea $\bar{x}$ la solución del último programa lineal resuelto:

1) Selección de una variable con valor cercano a 0.5 que disponga del mayor coeficiente en la función objetivo. Se busca L y H tales que $L = \min\{0.5, \max\{ \bar{x}_e \mid \bar{x}_e \leq 0.5, e \in E\}\}$ y $H = \max\{0.5, \min\{ \bar{x}_e \mid \bar{x}_e \geq 0.5, e \in E\}\}$. Sea $C = \{e \in E \mid 0.75 \cdot L \leq \bar{x}_e \leq H + 0.25 \cdot (1 - H)\}$ el conjunto de variables cuyo valor es cercano a 0.5. Se selecciona de C la variable con mayor coeficiente en la función objetivo.
2) Selección de la variable con el valor obtenido del programa lineal más cercano a 0.5.
3) Selección de la variable fraccional con el mayor coeficiente en la función objetivo.
4) Si hay variables fraccionales que son iguales a 1 en la solución preferible, selección de la de mayor coeficiente en la función objetivo.
5) Selección de la variable fraccional más cercana a 1.

- Para evitar que aumente desmesuradamente el tamaño del programa lineal y éste sea inabordable, se van eliminando restricciones resultantes de cortes (como hacen Jünger et al. 1994, quienes especifican que eliminan aquellas restricciones dominadas o que no definen facetas, y que por tanto ajustan débilmente, y aquellas que son poco importantes).

Según Hoffman & Padberg (1993), Jünger et al. (1994) y Johnson et al. (1997), la fase más importante en los procedimientos de planos de corte, reside en el hecho de encontrar restricciones violadas dada una solución del programa lineal que no satisface todas las restricciones del problema original. Debido a que el cierre convexo de puntos factibles que se puede asociar a un problema combinatorio es un poliedro que puede ser descrito por un conjunto finito de inecuaciones lineales, tal restricción debe existir, aunque saber cuál es, es muy difícil. En particular, si todas las variables en la restricción $\sum_j a_{ij} \cdot x_j \leq b_i$ deben ser no negativas, el corte es dado por $\sum_j \lfloor a_{ij} \rfloor \cdot x_j \leq \lfloor b_i \rfloor$, donde $\lfloor \alpha \rfloor$ es la parte entera o redondeada hacia abajo del número real α; cuando $\lfloor b_i \rfloor < b_i$, la restricción ajusta el programa lineal. Esta idea está fácilmente extendida a restricciones con variables enteras y reales, y aunque los algoritmos puros de planos de corte que incorporan dichos cortes han proporcionado muy lenta convergencia y no son prácticos, los resultados empíricos obtenidos con estos cortes en un algoritmo *branch and cut* han sido más positivos (Johnson et al. 1997).

Para la programación lineal entera y mixta generales, de entre todas las restricciones posibles interesan, sobre todo, aquellas que definen facetas, y aunque se conoce que el obtener restricciones que definen facetas del poliedro convexo es importante, éstas son extremadamente difíciles de encontrar (Hoffman & Padberg 1993, Jünger et al. 1994 y Johnson et al. 1997). Ante tales dificultades, se persigue el más modesto objetivo de separar en clases de restricciones válidas, que en varios sentidos son fuertes y pueden dar facetas para un poliedro que contenga el cierre convexo, o dar caras de razonablemente gran dimensión, pero no necesariamente facetas. Como exponen Johnson et al. (1997), el problema de encontrar cortes violados puede ser polinomial o NP-hard (y entonces se utilizan heurísticas que pueden fallar y no encontrar ningún corte), y el tiempo que se debe invertir en encontrar cortes violados debe ser resuelto empíricamente caso por caso.

Fischetti et al. (1995), Balas et al. (1996) y Johnson et al. (1997) entre otros, exponen diversos tipos de cortes:

- *Lifted cover inequalities*[6]. Cortes que en algoritmos *branch and cut* y para programación binaria han probado ser muy efectivos, según dichos autores.

[6] Para mayor información sobre los cortes *lifted cover inequalities* se recomienda Crowder et al. (1983).

- *Lift-and-project*[7].

- *Flow cover inequalities*[8]. Generalización de las restricciones de cubrimiento, que en algoritmos *branch and cut* y para programación binaria mixta, han probado ser muy efectivas según estos autores.

- Por su parte, Fischetti et al. (1995) proponen definir nuevas familias de desigualdades que exploten las características propias del problema concreto a resolver (restricciones lógicas, ...).

Como comentan Hoffman & Padberg (1993) y Jünger et al. (1994) entre otros, el *branch and cut* puede utilizarse para resolver problemas de grandes dimensiones (incluso a veces pudiendo probar su optimalidad), aunque también se puede utilizar como un procedimiento heurístico, que además proporciona una cota inferior de la solución óptima. Uno de los problemas que presenta el *branch and cut* es que, aunque obtiene la solución óptima en un número finito de pasos, no se puede estimar el tiempo necesario para obtenerla ni aun sabiendo el tamaño del ejemplar. Además y como se expone en Jünger et al. (1994), la implementación de un *branch and cut* es más complicada que muchos algoritmos puramente combinatorios.

Cabe destacar que ningún autor, excepto Jünger et al. (1994), comentan cómo resolver el programa lineal: mediante el símplex o el símplex dual, según qué base sea factible. También señalan que el programa de resolución del programa lineal es uno de los cuellos de botella de estos procedimientos, ya que muchas veces más del 90% del tiempo de cálculo se lo pasa resolviendo programas lineales. Exponen que existen eficientes programas que implementan el símplex (CPLEX, OSL, etc.) o métodos de punto interior (OSL, CPLEXbarrier, etc.) y que es necesario que éstos dispongan de potentes rutinas de post-optimización (para detectar infactibilidades, etc.).

Como en todo procedimiento de exploración del espacio de estados, en el *branch and cut* también tiene una importancia vital el modelizar y formular el problema a resolver de la manera más adecuada posible. Alonso & Escudero (1995, 1998) resuelven un problema de planificación del tráfico aéreo mediante dos formulaciones equivalentes: la segunda necesita un mayor número de variables y restricciones, pero muchas de éstas definen facetas de la región factible; como consecuencia, es frecuente que en la solución continua relajada del problema se obtengan muchas variables con valor entero y, por consiguiente, el proceso de resolución mediante *branch and cut* es mucho menor y más rápido.

[7] Se puede obtener una información más detallada sobre los cortes *lift-and-project* en Balas et al. (1996) y, como estos mismos autores recomiendan, en Balas et al. (1993).

[8] Una mayor concreción sobre los cortes *flow cover inequalities* se puede encontrar en Padberg et al. (1985).