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

3.5.9. *Branch and price.*

Como definen Johnson et al. (1997), se llama algoritmos *branch and price* a aquellos algoritmos *branch and bound*, en los que los programas lineales resultantes de la relajación son resueltos en cada vértice del árbol de búsqueda mediante generación de columnas.

Gass & Harris (1996) definen la técnica de "generación de columnas" -introducida en 1961 por Gilmore & Gomory en la resolución de problemas de corte- como un procedimiento que permite solucionar problemas de programación lineal de grandes dimensiones, mediante la generación de las columnas de la matriz de restricciones sólo cuando son necesarias (así, es habitualmente empleado cuando la matriz de restricciones es demasiado grande para ser almacenada o cuando sólo es conocida implícitamente); Savelsbergh (1997) define dicha técnica como un esquema *pricing* para resolver programas lineales de gran tamaño: en vez de valorar variables no básicas por enumeración, se encuentra la variable con menor (o mayor) coste reducido mediante la resolución de un problema de optimización. Como aclaran Jünger et al. (1994), Saltzman (1995), Johnson et al. (1997), el propio Savelsbergh (1997) y Jünger & Thienel (1998), la idea básica es simple: consiste en resolver el programa lineal con un número de columnas omitidas (ya que son demasiadas para manejarlas eficientemente y además muchas de ellas tendrán su variable asociada igual a 0 en una solución óptima) mediante, por ejemplo, el símplex, y comprobar la optimalidad de la solución obtenida resolviendo un subproblema, llamado *pricing problem*, con el que se definen las columnas (las variables) que deben entrar en la base; si se encuentran dichas columnas el programa lineal es reoptimizado, si no ya se ha alcanzado la solución óptima. Una condición necesaria para que este procedimiento sea eficaz, es que el *pricing problem* sea fácil de solucionar.

Según Saltzman (1995) y Jünger & Thienel (1998), *branch and price* es un concepto dual al de *branch and cut* (similar en espíritu según Savelsbergh 1997) y puede ser aplicado a programas enteros con un gran número de variables; por tanto, en un esquema *branch and bound* el hecho diferencial de este procedimiento es la forma de resolución del programa lineal, y, como en todos los procedimientos de este tipo, el proceso de ramificación se ejecuta cuando el programa lineal no proporciona una solución óptima entera. Sin embargo, como algunos autores opinan (Johnson 1989, Johnson et al. 1997, Mehrotra & Trick 1997, Savelsbergh 1997, etc.), no es apropiado aplicar un *branch and bound* "standard" sobre las columnas existentes ya que pueden inferir en el algoritmo de generación de columnas, y es conveniente diseñar un conjunto de reglas especiales de ramificación: la regla de ramificación debe ser compatible con el problema *pricing*: debe ser capaz de modificar el subproblema de forma que las columnas que son

infactibles debido a las restricciones de ramificación no sean generadas y el subproblema de generación de columnas permanezca tratable (Savelsbergh 1997).

Como expone Savelsbergh (1997), recientemente se han comenzado a desarrollar estrategias de ramificación que manejan esas dificultades: Desrochers, Desrosiers & Solomon en 1992 para el VRP, Desrochers & Soumis en 1989 y Anbil, Tanga & Johnson en 1991 para el problema de secuenciación de tripulaciones, y Vance, Barnhart, Johnson & Nemhauser en 1992 para el problema de corte. Para el problema de asignación generalizada, Johnson et al. (1997) y Savelsbergh (1997) proponen dos formulaciones alternativas: por una lado la formulación standard (en la que las variables x_{ij} representan si se asigna o no el trabajo j al agente i), y por otro proponen formularlo como un problema de partición (en el que las variables asociadas a un agente representan todas las posibles asignaciones factibles de trabajos j a dicho agente i, acompañadas de una variable binaria que indica si se selecciona o no dicha asignación). Aunque dichos autores resuelven la formulación disgregada mediante *branch and price*, ramifican utilizando la formulación standard y considerando las implicaciones que esto conlleva en las columnas asociadas en la formulación disgregada. En la formulación standard, fijar la variable x_{ij} a 0 prohibe que el trabajo j sea asignado al agente i, y fijar la variable x_{ij} a 1 obliga que el trabajo j sea asignado al agente i. En la formulación disgregada esto se puede cumplir como sigue: para prohibir la asignación de un trabajo j al agente i, todas las variables binarias para columnas asociadas con el agente i que tienen un 1 en la fila correspondiente al trabajo j, son fijadas a 0; para obligar la asignación del trabajo j al agente i, todas las variables binarias para columnas asociadas con el agente i que no tienen un 1 en la fila correspondiente al trabajo j, son fijadas a 0, y todas las variables binarias para columnas no asociadas con el agente i que tienen un 1 en la fila correspondiente al trabajo j, son fijadas a 0. No es difícil comprobar que el esquema de ramificación resultante es también compatible con el problema *pricing*: el problema *pricing* envuelve la solución de un problema de la mochila para cada agente; la prohibición de la asignación del trabajo j al agente i, se cumple al no considerar el trabajo j en la mochila para el agente i, y la obligación de la asignación del trabajo j a la agente i, se cumple al no considerar el trabajo j en la mochila para el agente i y reduciendo la capacidad de la mochila i por la cesión de capacidad para el trabajo j.

Según Gass & Harris (1996), la generación de columnas y los procedimientos *branch and price*, también son utilizados para resolver problemas en los que las variables son formadas por consideraciones combinatorias que incorporan restricciones: en un problema de diseño de rutas, se pueden definir las variables como todos los pétalos posibles que se pueden formar con los clientes, así se les incorpora la imposibilidad de crear ciclos que es lo complicado de formular. Como ya se ha comentado, para el problema de asignación generalizado, Johnson et al. (1997) y Savelsbergh (1997) proponen dos formulaciones alternativas: el programa lineal de la segunda formulación

es más ajustado que el de la formulación standard, pero en contrapartida requiere un número de columnas exponencial con el tamaño del input. Así, dos nuevas razones para utilizar formulaciones con enormes números de variables residen en el hecho de proporcionar cotas ajustadas que compactan la formulación y en el de eliminar problemas de simetrías.

Como señala Johnson (1989) si no se resuelve óptimamente el programa lineal, las cotas obtenidas no son válidas; de todas formas, propone que se detenga dicha optimización para ramificar el vértice original si los subproblemas en los vértices creados por ramificación pueden ser más fáciles de resolver.

Con el objetivo de aclarar el funcionamiento del procedimiento de generación de columnas y de la técnica de ramificación, a continuación se describe la exposición que hace Johnson (1989) de la resolución del problema del corte mediante *branch and price*.[9]

Sean r rollos de papel de longitudes B_1, ..., B_r, y m longitudes L_1, ..., L_m que deben ser cortadas de los rollos con una demanda de D_1, ..., D_m; en la versión considerada, se dispone de un número fijo de rollos de longitudes dadas y se deben cortar minimizando la demanda no servida (s_i).

La formulación clásica es la siguiente:

$$
\begin{aligned}
&[\text{MIN}]\ Z = \textstyle\sum^{m}_{i=1} s_i \\
&\qquad \textstyle\sum^{m}_{i=1} L_i \cdot x_{ij} \le B_j \qquad j = 1, \ldots, r \\
&\qquad \textstyle\sum^{r}_{j=1} x_{ij} + s_i = D_i \qquad i = 1, \ldots, m \\
&\qquad x_{ij} \ge 0 \text{ y entera,}
\end{aligned}
$$

donde x_{ij} es el número de longitudes L_i cortadas del rollo j.

Johnson (1989) comenta que esta formulación no es la mejor para resolver el problema, un *branch and bound* tarda mucho tiempo ya que las variables no representan decisiones importantes y existen muchas simetrías en la formulación. Una formulación mejor consiste en tomar como variables los patrones de corte de los rollos; esto es, sea y^k_j una variable con su correspondiente vector columna $a^k_j = (a^k_{1j}, \ldots, a^k_{mj})^T$ donde k indexa todas las soluciones que cumplen:

[9] En Savelsbergh (1997) se describe la resolución del problema de asignación generalizado mediante *branch and price*, también con gran grado de detalle, sobre todo de implementación práctica: cómo obtener un problema restringido inicial para comenzar el procedimiento de generación de columnas, obtención de soluciones factibles en cada vértice de la arborescencia mediante heurísticas que incorporan optimización local, estrategias de ramificación y de exploración utilizadas, etc.

$$\sum^{m}_{i=1} L_i \cdot a_{ij} \leq B_j$$
$$a_{ij} \geq 0 \text{ y entera,}$$

por tanto, a^k_{ij} representa el número de longitudes L_i cortadas del rollo j en la k-ésima solución del problema. Entonces las y^k_j deben satisfacer:

$$[MIN]\ Z = \sum^{m}_{i=1} s_i$$
$$\sum_k y^k_j = 1 \qquad j = 1, ..., r \quad (2)$$
$$\sum_{j,k} a^k_{ij} \cdot y^k_j + s_i = D_i \qquad i = 1, ..., m \quad (3)$$
$$y^k_j \geq 0 \text{ y entera,} \qquad (1)$$

donde a^k_j es el patrón factible de corte k para el rollo j, y y^k_j es una variable binaria que indica si el patrón de corte k del rollo j se corta o no de dicho rollo j.

Esta segunda formulación es mejor que la primera ya que la solución del programa lineal está más ajustada a una solución entera, pero presenta la gran desventaja de que existen muchas columnas. Gilmore & Gomory muestran cómo superar esta dificultad mediante la generación de columnas y la resolución del *pricing problem*: dada la solución óptima de un programa lineal para un subconjunto de columnas, donde (σ, π) es la solución dual y siendo σ_j la variable dual de la restricción (2) y π_i de la restricción (3), se resuelve el siguiente problema de la mochila:

$$[MAX]\ Z = \sum^{m}_{i=1} \pi_i \cdot u_i$$
$$\sum^{m}_{i=1} L_i \cdot u_i \leq B_j$$
$$u_i \geq 0 \text{ y entera,}$$

y se compara el valor de la función objetivo con σ_j: si es menor que σ_j la solución actual del programa lineal es óptima, si no, se añade una nueva columna k con un 1 en la fila j de (2) y con los siguientes coeficientes en las filas (3):

$$a^k_{ij} = u_i$$

Johnson (1989) también expone la descomposición general y resolución mediante *branch and price* de problemas de programación entera mixta. Sea un problema general entero mixto de la forma:

$$[MIN]\ z = c \cdot x + d \cdot y$$
$$A \cdot x = b$$
$$F \cdot x + G \cdot y = g$$
$$x \geq 0 \text{ y entera, } y \geq 0.$$

Se asume que el problema entero puro:

$$[MIN]\ z = c \cdot x$$
$$A \cdot x = b$$
$$x \geq 0 \text{ y entera,}$$

tiene un número finito de soluciones enteras: $x^1, ..., x^K$. Entonces el problema maestro es el siguiente:

$$[MIN]\ z = \sum_{k=1}^{K} (c \cdot x^k) \cdot \lambda^k + d \cdot y$$
$$\sum_{k=1}^{K} \lambda^k = 1 \qquad (2) \quad \text{(sólo una solución entre } x^1, \ldots, x^K)$$
$$\sum_{k=1}^{K} (F \cdot x^k) \cdot \lambda^k + G \cdot y = g \qquad (3)$$
$$\lambda^k \geq 0 \text{ y entera, } y \geq 0. \qquad (1)$$

Para obtener la columna que entra en la base se utilizan las variables duales π de (3) y se resuelve el siguiente problema de optimización:

$$[MIN]\ z = (c - \pi \cdot F) \cdot x$$
$$A \cdot x = b$$
$$x \geq 0 \text{ y entera.}$$

Una solución x^*, cuyo valor de la función objetivo z^* satisface $z^* < \sigma$, donde σ es la variable dual correspondiente a la restricción (2), proporciona una columna para entrar en la base del programa lineal maestro.

Para finalizar, parece conveniente exponer dos precisiones que realiza Savelsbergh (1997) sobre los algoritmos *branch and price*.

Resolver el problema de generación de columnas implica la resolución de muchos problemas de la mochila, lo que puede ser computacionalmente prohibitivo. Afortunadamente, para que funcione el esquema de generación de columnas no es necesario seleccionar siempre la columna con el mayor coste reducido (en caso de maximizar), cualquier columna con coste reducido positivo puede servir. Así Savelsbergh (1997) propone varios esquemas alternativos de generación de columnas encaminados a reducir el tiempo de computación global: seleccionar la primera columna encontrada con coste reducido positivo, seleccionar todas las columnas encontradas con coste reducido positivo, o utilizar algoritmos heurísticos (en su aplicación, de dos fases) para resolver el problema *pricing*.

Savelsbergh (1997) también expone dos formas de truncar el árbol de búsqueda, comentando que con los algoritmos *branch and price* también se pueden desarrollar algoritmos heurísticos; de todas formas, no se debe perder de vista que éstas pueden ser aplicadas a cualquier algoritmo de exploración de estados tipo *branch and bound.*[10]

3.5.10. *Branch and cut and price.*

En los últimos años se están desarrollando nuevos procedimientos híbridos basados en *branch and cut* y en *branch and price*, denominados algoritmos *branch and cut and price*. En *http://www.informatik.uni-koeln.de/ls_juenger/projects/abacus.html*, Jünger & Thienel describen brevemente el sistema ABACUS, un paquete informático que proporciona una estructura para la implementación de algoritmos *branch and bound*, utilizando como relajación la programación lineal que puede ser complementada con la generación dinámica de planos de corte, de columnas o con la combinación de ambos; de esta manera se obtienen los procedimientos *branch and cut and price*. Como los mismos autores comentan en Jünger & Thienel (1998), ABACUS está diseñado para la resolución de problemas de optimización enteros y mixtos, y para problemas de optimización combinatoria.

Según Jünger & Thienel (1998), existen problemas de optimización combinatoria (por ejemplo, el problema de corte) que pueden ser descritos de dos maneras equivalentes: como un programa entero o de forma combinatoria. Ambas representaciones tienen ventajas e inconvenientes. La formulación como programa entero es la necesaria para aplicar los procedimientos *branch and cut and price*, pero en este caso la estructura real del problema es muy difícil de reconocer si el problema únicamente está representado por una matriz. Esta información estructural está únicamente presente en una formulación combinatoria, que por otro lado es inadecuada para aplicar técnicas de programación entera. Como exponen estos mismos autores, esta pérdida de información estructural en la formulación como programa entero podría ser aceptada si la transformación de la formulación combinatoria puede ser representada explícitamente, y si el problema transformado puede ser resuelto con procedimientos standard; pero, lamentablemente, ambos criterios no son satisfechos por muchos problemas de optimización combinatoria.

El número de restricciones o de variables en una formulación como programa entero de un problema de optimización combinatoria, puede ser exponencial con el tamaño de la formulación combinatoria del mismo problema. En un TSP con n ciudades, el número de restricciones necesarias para eliminar las subrutas en una formulación como programa entero es $O(2^n)$; por tanto, se necesita un procedimiento de planos de corte

[10] Para mayor información sobre *branch and price* se recomienda Barnhart et al. (1998), un brillante y actualizado trabajo sobre el tema.

para manejar un número tan elevado de restricciones. Por otro lado, la formulación como programa entero del problema de corte tiene un número exponencial de variables; si se desea determinar su solución óptima mediante *branch and price*, se necesita, irremediablemente, información sobre la estructura combinatoria del problema para la resolución de los problemas de la mochila en la generación dinámica de columnas.

Además hay que tener presente que el mejor software comercial para la resolución de problemas de programación entera incorpora algoritmos *branch and cut* que utilizan cortes de mochila, que aunque se pueden utilizar con casi cualquier matriz de restricciones, son bastante débiles; el uso de la información estructural que contienen los problemas de optimización combinatoria ayuda a definir inecuaciones, de las que derivan planos de corte que usualmente son mucho más ajustados que los planos de corte de propósito general. Los planos de corte específicos de problemas de optimización combinatoria, son habitualmente descritos de forma combinatoria.

Por otro lado, a menudo la separación o los algoritmos *pricing* para problemas de optimización combinatoria retornan restricciones o variables en una representación combinatoria. Esta representación combinatoria debe ser transformada en una fila o columna de la matriz de restricciones junto con su término independiente o su coeficiente en la función objetivo, respectivamente.

Como se puede comprobar, en un procedimiento *branch and cut and price* como ABACUS, que unifica los planos de corte y la generación de columnas en un mismo algoritmo, se generan dinámicamente tanto restricciones como variables. Y es necesaria durante todo el proceso, tanto la representación combinatoria de restricciones y variables, como su formato de programa lineal.

El sistema ABACUS provee de una variedad de conceptos algorítmicos generales, como por ejemplo estrategias de enumeración y ramificación, de las cuales el usuario puede seleccionar la mejor alternativa para su aplicación; aunque para manejar el enorme número de restricciones y variables que surgen, se utilizan estructuras de datos especiales (Jünger & Thienel 1998).

Como comentario final a los procedimientos descritos en los apartados anteriores, cabe destacar que, concretamente, el *branch and cut*, el *branch and price* y el *branch and cut and price* son utilizados exclusivamente pensando en programación lineal, mientras que el concepto subyacente en el procedimiento *branch and bound* es mucho más general: puede no interesar formular como restricciones lineales algunas condiciones no lineales (pero linealizables).

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

En la bibliografía procedente del área de la inteligencia artificial, usualmente se describen una serie de procedimientos de búsqueda que encajan perfectamente con los procedimientos enumerativos de búsqueda en espacios de estados, y que además a veces coinciden exactamente. Estos procedimientos han evolucionado y han sido diseñados en el seno de la inteligencia artificial, usualmente con el objetivo de encontrar una o diversas soluciones factibles más que resolver un problema de optimización.

Estas técnicas también incluyen entre sus componentes tanto procedimientos que algunos clasifican como ciegos (o sin información) y que no utilizan ningún tipo de información heurística acerca del problema, como los llamados guiados (o con información heurística) que son procedimientos que sí utilizan alguna clase de información para evaluar los vértices activos, a fin de extender la exploración por los vértices más prometedores; éstas son las técnicas más usualmente estudiadas en esta disciplina, ya que son las que presentan mayor grado de "inteligencia" al aprovechar dicha información para restringir la búsqueda y el coste de la exploración. De todas formas y como recuerda Nilsson (1980), dicha información tiene un coste y se debe llegar a un compromiso entre el coste de la estrategia de control de la búsqueda y el coste de la computación.

Para algunos autores como Cortés et al. (1993), los principales puntos de investigación son el tipo de control de la búsqueda y las funciones heurísticas utilizadas para evaluar los vértices. Según estos mismos autores la novedad de las técnicas descritas en el área de la inteligencia artificial reside en la limitación de la información empleada, que genera un espacio de estados mucho menor. Cabe destacar sin embargo, que esta idea se lleva utilizando en investigación operativa desde hace ya casi cuarenta años, cuando Land & Doig ("An Automatic Method of Solving Discrete Programming Problems". *Econometrica*, vol. 28, 1960, pp. 497-520) y Little et al. ("An Algorithm for the Traveling Salesman Problem". *Operations Research*, vol. 11, 1963, pp. 972-989) comienzan a formular los procedimientos denominados de *branch and bound.* Por su parte, Greenberg (1996) señala que en este tema la investigación operativa se diferencia de la inteligencia artificial en que ésta debe encontrar soluciones como la investigación operativa y además proveer de herramientas para probar teorías de conductas inteligentes; según Greenberg (1996), históricamente la investigación operativa se ha centrado en la eficiencia computacional explorando las estructuras matemáticas de los problemas, mientras que la inteligencia artificial se ha centrado en utilizar la lógica en una amplia clase de problemas.

Los procedimientos expuestos en el presente apartado habitualmente también han sido descritos por varios autores, y, como ocurre en algunas de las técnicas ya comentadas, a veces las definiciones que presentan no son suficientemente claras y precisas.

3.6.1. *Backtracking* dirigido por la dependencia (*dependency directed backtracking* o DDB).

Kleer (1990) y Rich & Knight (1994) definen la técnica de *backtracking* dirigido por la dependencia (*dependency directed backtracking* o DDB), como un procedimiento de *backtracking* en el que en el momento que se descubre una contradicción se tiene en cuenta para evitar generar nuevos vértices que incorporen dicha contradicción. Por ejemplo, si se fija una variable x_i a 1 y en un vértice descendiente se fija otra variable x_j a 0 comprobándose que es imposible si x_i vale 1, a partir de este momento y para todos los vértices sucesores de $x_i = 1$ o incluso para todo el árbol, ya no se volverá a generar un vértice en el que x_j valga 0 si x_i vale 1.

A pesar de que este procedimiento puede proporcionar una reducción del espacio de estados, Kleer (1990) comenta que el coste necesario para mantener las dependencias puede llegar ser mayor que el esfuerzo de búsqueda ahorrado.

Este tipo de técnicas que se basan en propagar restricciones o en tener presentes en todo momento algunas o todas las contradicciones descubiertas, se pueden englobar en un conjunto de técnicas conocidas como de propagación o verificación de restricciones (procedimientos introducidos en el apartado 3.8.). De todas formas, Rich & Knight (1994) consideran que la técnica de *backtracking* dirigido por la dependencia utiliza un esquema más sofisticado que los procedimientos usuales de verificación de restricciones, ya que identifica la causa específica de la inconsistencia y, en la vuelta atrás, sólo deshace las restricciones que dependen de esta causa (no modificando las otras, aunque se hayan generado después de la culpable, siempre y cuando sean independientes del problema y de su causa): el *backtracking* se basa en las dependencias lógicas en lugar del orden cronológico en que produjeron las decisiones (como hacen los procedimientos habituales de propagación de restricciones).

3.6.2. Profundización iterativa (*iterative deepening* o ID).

En la literatura se pueden encontrar diversos autores que definen el procedimiento de búsqueda denominado de profundización iterativa (*iterative deepening* o ID), de igual manera que otros definen el algoritmo de profundización iterativa primero en profundidad (*depth first iterative deepening* o DFID); a veces, incluso se considera que ambos son la misma técnica y al procedimiento de profundización iterativa se le denomina indistintamente ID o DFID. Pero como exponen Korf (1993) y Zhang & Korf

(1995), la profundización iterativa incluye al algoritmo de profundización iterativa primero en profundidad (*depth first iterative deepening* o DFID), al procedimiento de profundización iterativa primero el de menor coste (*uniform-cost iterative deepening*) y a la técnica de profundización iterativa A* (*iterative deepening A** o IDA*).

La idea del procedimiento es sencilla y persigue el objetivo de no expandir nunca un vértice cuyo coste asociado sea mayor que el coste de una solución óptima. Utilizando una variable global denominada coste umbral de corte y que inicialmente toma el valor del coste asignado al vértice raíz, se ejecutan una serie de búsquedas en profundidad. En cada iteración se ejecuta una búsqueda en profundidad, expandiendo todos aquellos vértices cuyo coste asociado es menor o igual que el valor del coste umbral de corte dado. Si se encuentra la solución buscada en esta iteración la búsqueda finaliza, sino el árbol es explorado de nuevo y desde el inicio, con una búsqueda en profundidad y un nuevo coste umbral de corte que coincide, según Zhang & Korf (1995), con el mínimo coste de todos los vértices que han sido generados pero no expandidos en la última iteración (ya que su coste excede el valor del coste umbral de corte de dicha iteración); y así sucesivamente.

Como se puede comprobar, las iteraciones individuales se realizan utilizando la búsqueda en profundidad y no el coste asociado al vértice; éste se utiliza para decidir los vértices que están por debajo del valor del coste umbral de corte, pero no para determinar el orden en el que deben ser explorados.

En función del tipo de coste que se asocia al vértice se pueden especificar, como ya se ha comentado, los procedimientos siguientes (Korf 1993 y Zhang & Korf 1995):

- si el coste asociado al vértice es su profundidad, se obtiene el algoritmo de profundización iterativa primero en profundidad (expuesto en el apartado 3.6.3.);

- si el coste asociado es g(n) (coste del camino desde el vértice raíz s al vértice n), se dispone del procedimiento de profundización iterativa primero el de menor coste (expuesto en el apartado 3.6.4.);

- si el coste asociado al vértice es f(n) = g(n) + h(n), donde g(n) es la suma del coste del camino asociado a alcanzar dicho vértice desde el vértice raíz y h(n) es una estimación heurística del coste mínimo de alcanzar un vértice objetivo desde el vértice n, se consigue la técnica de profundización iterativa A* (expuesta en el apartado 3.6.9.).

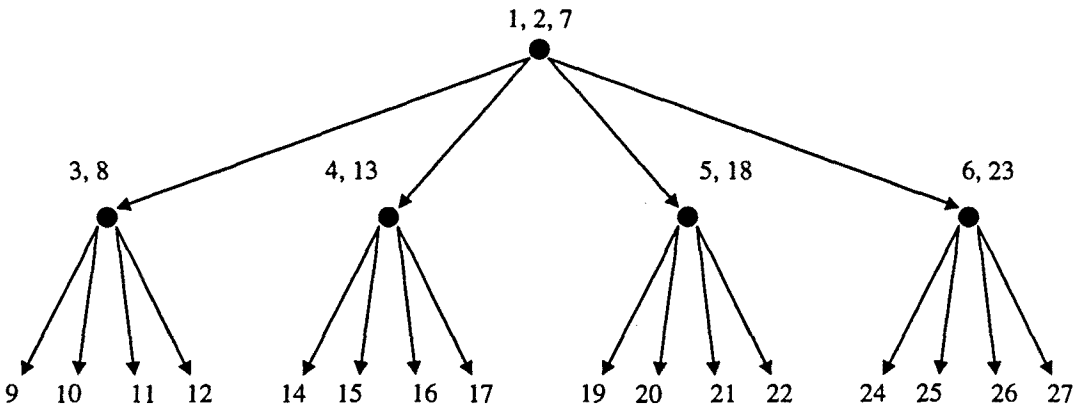
3.6.3. Profundización iterativa primero en profundidad (*depth first iterative deepening* o DFID).

Entre otros, Korf (1985, 1990), Ginsberg (1993), Dasgupta et al. (1994), Rich & Knight (1994) y Zhang & Korf (1995), definen el algoritmo de profundización iterativa primero en profundidad (*depth first iterative deepening* o DFID).

Básicamente, este procedimiento va variando el nivel de profundidad de la búsqueda intentando aprovechar las mejores características de la búsqueda en profundidad y en anchura. La idea consiste en comenzar con una búsqueda en profundidad a una profundidad máxima de corte igual a uno (por lo que los vértices de mayor profundidad no son examinados); si se encuentra la solución buscada en dicho nivel finaliza la búsqueda, sino el árbol es investigado de nuevo y desde el inicio, con una búsqueda en profundidad hasta una profundidad máxima de dos; y así sucesivamente. Como se puede comprobar, es, en cierta forma, una combinación de búsqueda en profundidad y en anchura.

Korf (1990) proporciona una definición más compacta de la técnica de búsqueda de profundización iterativa primero en profundidad: procedimiento que consiste en ejecutar una búsqueda en profundidad hasta una profundidad de corte igual a uno; después se completa una nueva búsqueda en profundidad hasta una profundidad de corte igual a dos, y se continua ejecutando búsquedas en profundidad, incrementando la profundidad de corte en 1 en cada iteración, hasta que se encuentra una solución considerada objetivo.

El gráfico siguiente muestra un ejemplo de cómo actúa la búsqueda de profundización iterativa primero 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:

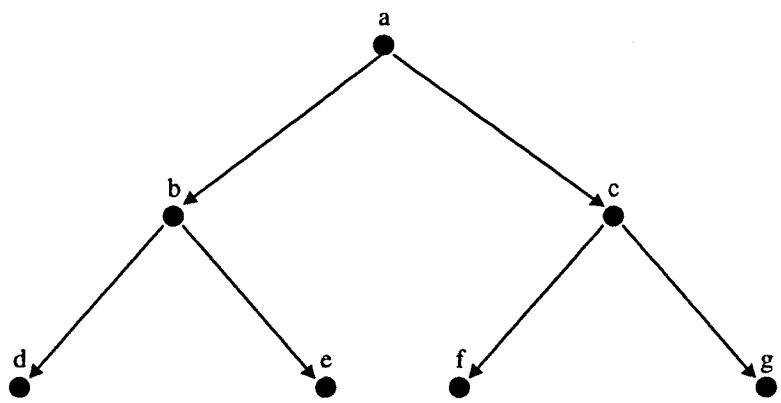


Orden de exploración del espacio de estados en una profundización iterativa primero en profundidad, de Ginsberg (1993).

Dasgupta et al. (1994) definen el procedimiento de manera semejante aunque presentan dos nuevas ideas que pueden hacer mejorar el funcionamiento del mismo.

a) Por un lado, consideran que la profundidad de corte de la próxima iteración se puede fijar a priori de forma estática o se puede decidir de forma dinámica durante la iteración en curso. Esta idea parece que puede acelerar el proceso de búsqueda si se comienza con grandes incrementos de la profundidad de corte y se van disminuyendo a medida que avanza la búsqueda.

b) Por otro lado, también proponen no comenzar la nueva iteración en el vértice raíz, sino aprovechar la posición del último vértice generado en la iteración anterior. Mediante el ejemplo siguiente (Dasgupta et al. 1994) se expone el funcionamiento de una búsqueda en profundidad, una búsqueda de profundización iterativa primero en profundidad usual y una búsqueda de profundización iterativa primero en profundidad que incorpora dicha idea; las letras asociadas a cada estado especifican el vértice referenciado:



Ejemplo para diferentes procedimientos de búsqueda, de Dasgupta et al. (1994).

En una búsqueda en profundidad, el orden de exploración de los vértices es *abdbebacfcg*; en una búsqueda de profundización iterativa primero en profundidad usual, el orden de exploración de los vértices es *abacabdbebacfcg*; y en una búsqueda de profundización iterativa primero en profundidad que incorpora dicha idea, el orden de visita de los vértices es *abacfcgcabdbe*.

Una idea que parece interesante a priori y que no es descrita por ninguno de los autores referenciados o al menos no está claramente especificada, es el hecho de explorar únicamente a partir de los vértices terminales de la iteración anterior sin pasar por los vértices intermedios ya explorados, ya que si éstos son vértices objetivo ya se habrán detectado en las iteraciones anteriores; esto implica que se han de guardar todos los vértices terminales de la iteración anterior, lo que constituye de hecho una búsqueda en

anchura. Dos nuevas ideas a considerar, son el no comprobar si los vértices intermedios son vértices objetivo (ya que seguro que no lo son), y el hecho de no expandir nuevamente los vértices que conducen a vértices terminales en iteraciones anteriores y que en principio serían generados y expandidos en toda nueva iteración (idea que requiere el consumo de mayor cantidad de memoria y que debería ser estudiada con detenimiento para validar su conveniencia).

Ginsberg (1993) comenta que la memoria necesaria para ejecutar este procedimiento es la misma que en la búsqueda en profundidad. Korf (1985) considera que la profundización iterativa primero en profundidad intenta evitar los inconvenientes de la búsqueda en profundidad y en anchura, y que su desventaja es que ejecuta computación inútil antes de encontrar la profundidad de un vértice objetivo (pero de todas formas siempre encuentra el camino más corto a dicho vértice objetivo).

3.6.4. Profundización iterativa primero el de menor coste (*uniform-cost iterative deepening*).

Zhang & Korf (1995) definen una nueva técnica de búsqueda en el espacio de estados, denominada procedimiento de profundización iterativa primero el de menor coste (*uniform-cost iterative deepening*).

Como ya se ha comentado, este procedimiento es una particularización del procedimiento de profundización iterativa: a cada vértice se le asocia el coste del camino desde el vértice raíz a dicho vértice n, g(n). De esta manera, en cada iteración se ejecuta una búsqueda en profundidad, expandiendo todos aquellos vértices cuyo coste asociado, en este caso g(n), es menor o igual que el valor del coste umbral de corte dado (que coincide con el valor mínimo de g(n) de todos los vértices generados pero no expandidos en la iteración anterior). De manera inmediata se puede comprobar que se trata de un procedimiento de profundización iterativa, asociando a los vértices el mismo coste que se utiliza como guía en la búsqueda primero el de menor coste (procedimiento descrito en el apartado 3.3.5.).

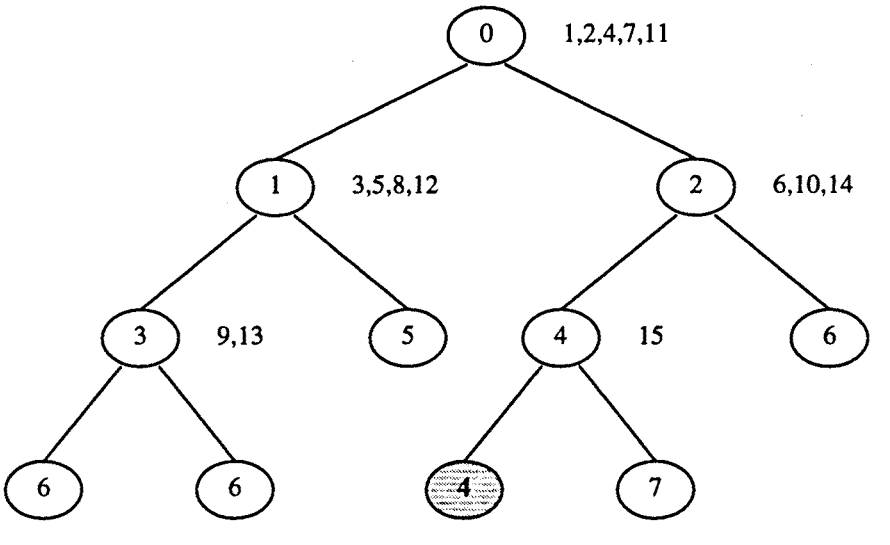
3.6.5. Búsqueda recursiva primero el mejor (*recursive best-first search* o RBFS).

Al analizar el procedimiento de profundización iterativa (ID) se puede comprobar que se realiza una gran cantidad de reexpansiones. Con el objetivo de reducir el número de éstas, Korf (1993) propone una modificación de esta técnica que denomina búsqueda recursiva primero el mejor (*recursive best-first search* o RBFS), también expuesta en Zhang & Korf (1995).

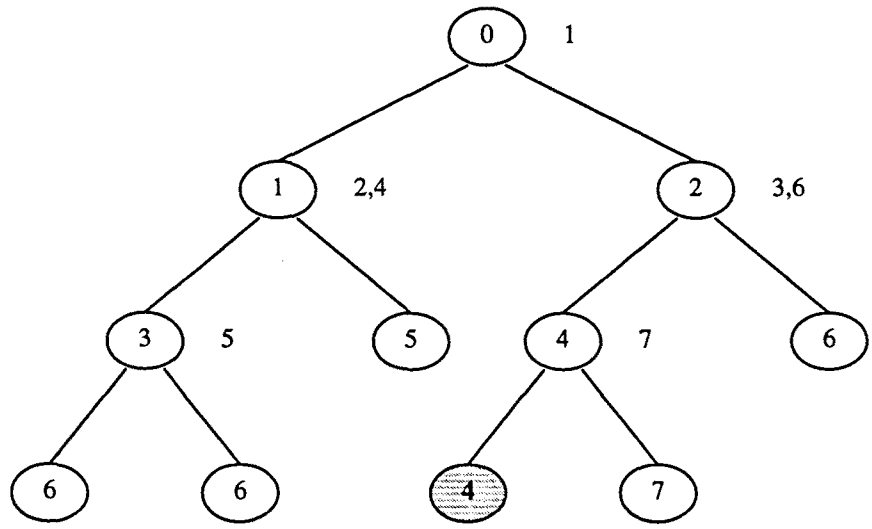
La diferencia clave entre la profundización iterativa y la búsqueda recursiva primero el mejor, consiste en que mientras la profundización iterativa mantiene un único coste umbral de corte para toda la arborescencia, la búsqueda recursiva primero el mejor calcula costes umbrales de corte locales para los descendientes del vértice raíz.

A continuación se describe el funcionamiento de la búsqueda recursiva primero el mejor. En primer lugar se expande el vértice raíz y a sus vértices hijos (vh_i) se les asocia un coste umbral de corte local, f_i, igual al coste asociado al propio vértice. Después de la inicialización, el proceso iterativo es el siguiente: se ordenan los costes umbrales de corte de los vértices vh_i en orden creciente; se realiza una profundización iterativa en el subproblema vh_j que presenta el menor coste umbral de corte f_j, con un nuevo coste umbral de corte igual al valor del segundo menor coste umbral de corte, f_k; si se encuentra la solución buscada el proceso finaliza, sino el valor de f_j del vértice vh_j es actualizado al valor del coste mínimo de todos sus descendientes cuyo valor excede de f_k y que por tanto no son explorados en esta iteración; y así sucesivamente.

En los gráficos siguientes se muestra un ejemplo del funcionamiento de la profundización iterativa primero el de menor coste, caso particular de la profundización iterativa descrita en el apartado 3.6.2., así como un ejemplo de la búsqueda recursiva primero el mejor, en la que la función de coste asociada a los vértices es, también, la utilizada como guía en la búsqueda primero el de menor coste (procedimiento descrito en el apartado 3.3.5.). El número incluido en el estado n representa el valor de g(n), los números que los acompañan especifican el número de orden en el que se explora el vértice referenciado y el estado sombreado corresponde a un vértice óptimo:



Orden de expansión de los vértices según la profundización iterativa primero el de menor coste, de Zhang & Korf (1995).



Orden de expansión de los vértices según la búsqueda recursiva primero el mejor, de Zhang & Korf (1995).

En la aplicación de la búsqueda recursiva primero el mejor, primero se expande el vértice raíz y se obtienen dos vértices hijos, A (el de la izquierda) y B (el de la derecha), cuyos costes umbrales de corte asociados son $f_A = 1$ y $f_B = 2$. En la primera iteración se realiza una profundización iterativa en el subproblema A con un coste umbral de corte igual a 2; no se encuentra una solución óptima y el nuevo valor de f_A pasa a ser 3 (mínimo entre 3 y 5). En la segunda iteración se realiza una profundización iterativa en el subproblema B con un coste umbral de corte igual a 3; no se encuentra una solución óptima y el nuevo valor de f_B pasa a ser 4 (mínimo entre 4 y 6). Se realiza una profundización iterativa en el subproblema A con un coste umbral de corte igual a 4; no se encuentra una solución óptima y el nuevo valor de f_A pasa a ser 5 (mínimo entre 6, 6 y 5). Y, por último, se realiza una profundización iterativa en el subproblema B con un coste umbral de corte igual a 5, se encuentra una solución óptima de valor 4 y el procedimiento acaba con éxito.

Cabe destacar que en el procedimiento de búsqueda recursiva primero el mejor, también se puede utilizar una función de coste igual a la profundidad de los vértices (Zhang & Korf 1995); de esta manera, se obtiene un procedimiento, no denominado de ninguna manera en particular, que intenta reexpansionar menos vértices que la versión de profundización iterativa primero en profundidad, expuesta en el apartado 3.6.3. En este momento, no es difícil de imaginar una versión en la que la función de costes asociados a los vértices sea igual a $f(n) = g(n) + h(n)$.

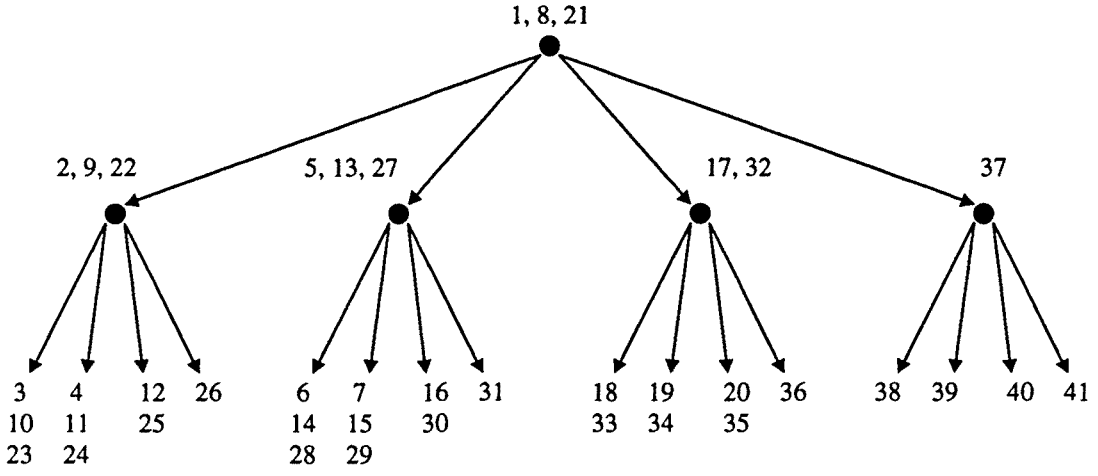
3.6.6. Ensanchamiento iterativo (*iterative broadening*).

Ginsberg (1993) define el algoritmo de ensanchamiento iterativo (*iterative broadening*) como un nuevo procedimiento de búsqueda basado en la "fuerza bruta" (ya referenciado

a esta definición en el apartado 3.3.13.), cuya característica principal reside en ir variando iterativamente el número de sucesores de cada vértice que se generan en cada iteración.

El procedimiento de ensanchamiento iterativo comienza la exploración del espacio de estados con una búsqueda en profundidad y un factor de ramificación igual a dos (entendido éste como el número máximo de hijos generados para cada vértice); si se encuentra la solución deseada con dicho factor de ramificación la búsqueda finaliza, sino el árbol es investigado de nuevo y desde el inicio, con una búsqueda en profundidad y un factor de ramificación igual a tres; y así sucesivamente. Como se puede comprobar, este procedimiento también es, en cierta forma, una combinación de búsqueda en anchura y en profundidad.

El gráfico siguiente muestra un ejemplo de cómo actúa la búsqueda de ensanchamiento iterativo, 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:



Orden de exploración del espacio de estados en un ensanchamiento iterativo, de Ginsberg (1993).

Según parece definir Ginsberg (1993), la generación de los vértices sucesores de uno dado se realiza de forma lexicográfica; una idea para intentar mejorar el rendimiento de este procedimiento, podría ser la de generar aquellos vértices de mejor valor de un indicador que incorpore algún tipo de información heurística acerca del problema. Este comentario pretende recordar la imprecisión existente entre las estrategias de exploración y los algoritmos que las incorporan, ya que a partir de la estrategia aquí definida se pueden diseñar múltiples algoritmos de exploración en función de cómo se generan los vértices sucesores, si se agrupan o no vértices equivalentes, etc.

3.6.7. Algoritmo Z y Z*.

Dentro de las estrategias guiadas o con información heurística acerca del problema, Pearl (1984) considera la búsqueda primero el mejor (*best first search* o BF) como aquel procedimiento que selecciona para expandir el vértice de mejor valor de una función heurística general de evaluación (apartado 3.3.7.). Pearl (1984) define el algoritmo Z como un procedimiento especial de búsqueda primero el mejor, que por tanto es un procedimiento de búsqueda basado en la "fuerza bruta" (ya referenciado a esta definición en el apartado 3.3.13.), en el que la función de evaluación f(n) se calcula recursivamente como una función del coste hasta el vértice anterior más los costes de los vértices sucesores, o con una estimación de éstos. Si se busca la solución óptima se obtiene el algoritmo Z*, para ello se requiere que la estimación cumpla la condición de admisibilidad.

Como señala Pearl (1984), en el algoritmo Z* la función de evaluación f(n) puede ser cualquier combinación de la función de estimación y de un conjunto de parámetros que caractericen a n.

Definiendo el algoritmo A* (procedimiento más ampliamente especificado en el apartado 3.6.8.) como una especialización del Z*, en la que el objetivo perseguido es el camino de mínimo coste y se utiliza una función de evaluación aditiva, Pearl (1984) realiza un diagrama jerárquico que contempla los tres procedimientos aquí comentados:

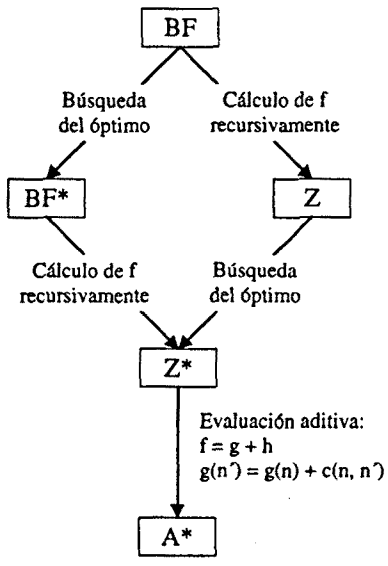


Diagrama jerárquico de los procedimientos BF, BF*, Z, Z*, A y A*, de Pearl (1984).

3.6.8. Algoritmo A y A*.

El procedimiento de búsqueda en espacio de estados más usualmente descrito por los investigadores provenientes del área de la inteligencia artificial es, probablemente, el algoritmo A* (ya referenciado a esta definición en el apartado 3.3.13.), que según Barr & Feigenbaum (1981), Raphael (1990) y Ginsberg (1993) fue desarrollado por primera vez por Hart, Nilsson & Raphael en 1968.

Greenberg (1996) habla de la familia de algoritmos A* como de un conjunto de procedimientos para la selección de los vértices a explorar en un árbol de búsqueda. Este comentario pone de nuevo de manifiesto la contradicción (o al menos la poca precisión) de algunas definiciones, ya que mientras este autor habla de una familia de algoritmos la mayoría lo considera un único procedimiento (de todas formas parece justo matizar que implícitamente es posible que estos autores consideren que, en función de la especificación concreta de los parámetros a considerar en un procedimiento A*, se puedan diseñar diferentes algoritmos A* a partir de esta misma idea). Por otro lado, existen autores que caracterizan a un procedimiento como el algoritmo A* en función únicamente de la estrategia de selección del vértice a expandir, pero también están aquéllos quienes además imponen la agrupación de vértices equivalentes y/o la eliminación de vértices para los que se puede asegurar que no conducen a ninguna solución mejor que la solución preferible (proceso que en esta tesis se denomina poda) (Cortés et al. 1993 y Winston 1994a).

Vistas las imprecisiones anteriores, en este texto se define el procedimiento A* considerándolo únicamente como una estrategia de selección de vértices a explorar, con la particularidad de utilizar información heurística acerca del problema.

A pesar de que el algoritmo A* es probablemente el procedimiento de búsqueda más estudiado en el área de la inteligencia artificial, existe una serie de autores que definen y diferencian entre los algoritmos A y A*, mientras que la mayoría únicamente considera el procedimiento A*.

Nilsson (1980), Mompín et al. (1987), Shirai & Tsujii (1987) y Companys (1989a), se encuentran entre los autores que diferencian entre los algoritmos A y A*. Para éstos:

a) Algoritmo A: procedimiento de búsqueda primero el mejor (por tanto, procedimiento de búsqueda basado en la "fuerza bruta") que selecciona para expandir el vértice n de menor valor de una función de evaluación, formada por la suma del coste mínimo asociado a alcanzar dicho vértice desde el vértice raíz (usualmente representada por g(n)), más una estimación heurística (habitualmente representada por h(n)) del coste mínimo de alcanzar un vértice considerado objetivo desde el vértice n (usualmente

$h^*(n)$). Por tanto se selecciona para expandir el vértice de menor valor de f(n) = g(n) + h(n).

b) Algoritmo A*: algoritmo A en el que la evaluación del vértice n está formada por la suma del coste mínimo asociado a alcanzar el vértice (g(n) como en el algoritmo A), más una estimación heurística h(n) de $h^*(n)$, que en este caso es una cota y por tanto cumple la condición de admisibilidad ($h(n) \leq h^*(n)$ para todo n). Consecuentemente se selecciona para expandir el vértice de menor valor de f(n) = g(n) + h(n), con $h(n) \leq h^*(n)$.

Existen otros autores (Nilsson 1971, Gass & Harris 1996 y Greenberg 1996) que únicamente definen el algoritmo A*, como el procedimiento de búsqueda que selecciona para expandir el vértice n de menor valor de una función de evaluación formada por la suma del coste mínimo asociado a alcanzar dicho vértice desde el vértice raíz, g(n), más una estimación heurística, h(n), que constituye una cota del mínimo coste de alcanzar un vértice considerado objetivo desde dicho vértice n, $h^*(n)$. Así se selecciona para expandir el vértice de menor valor de f(n) = g(n) + h(n), con $h(n) \leq h^*(n)$. Estos autores no especifican qué procedimiento se obtiene si h(n) no constituye una cota de $h^*(n)$.

Dentro de estas definiciones también existen particularidades que dificultan el entendimiento del algoritmo A*. Así por ejemplo, Pearl (1983) define el algoritmo A* considerando la función de evaluación f(n) = g(n) + h(n), con $h(n) \leq h^*(n)$, y donde f(n) representa una estimación heurística de la longitud del camino óptimo construido condicionado a pasar por el vértice n y asumiendo que todos los arcos tienen distancia unitaria, y, por tanto, parece que evaluándose g(n) y h(n) como longitudes en número de etapas.

Por último, existe un conjunto de autores (Barr & Feigenbaum 1981, Korf 1990, Raphael 1990, Cortés et al. 1993, Ginsberg 1993, Kusiak et al. 1993, Rich & Knight 1994, Winston 1994a, etc.) que únicamente consideran la existencia del algoritmo A*, aunque sí diferencian entre la posibilidad de que la función h(n) cumpla o no la condición de admisibilidad. Para éstos, el procedimiento A* es la técnica de búsqueda que selecciona para expandir el vértice n de menor valor de una función de evaluación formada por la suma del coste mínimo asociado a alcanzar dicho vértice desde el vértice raíz g(n), más una estimación heurística h(n) del mínimo coste de alcanzar un vértice considerado objetivo desde dicho vértice n. Si la estimación heurística h(n) constituye una cota de $h^*(n)$, entonces el procedimiento A* garantiza encontrar una solución óptima, si ésta existe. De todas formas, y a menos que se indique lo contrario, suele suponerse que la estimación h(n) es una cota de $h^*(n)$.

En este caso también existen definiciones particulares que pueden dificultar el entendimiento del algoritmo A*. Por ejemplo, Cortés et al. (1993), además de considerar los costes de los caminos en número de etapas, como ya hacía Pearl (1983), asumen la agrupación de estados equivalentes y la conservación únicamente del mejor camino hasta éstos, como también hacen Rich & Knight (1994) y Winston (1994a).

Como comentan diversos autores, Pearl (1984), Mompín et al. (1987), Companys (1989a), Cortés et al. (1993), Ginsberg (1993) y Greenberg (1996) entre otros, si no se dispone de información heurística, es decir si la función h(n) = 0, y g(n) se define como la profundidad del vértice n, el algoritmo A o el algoritmo A* correspondiente equivale a la exploración en anchura. Pearl (1984) también comenta que si la función h(n) = 0 y g(n) se define como menos la profundidad del vértice n, el algoritmo A* correspondiente equivale a la búsqueda en profundidad. Para este mismo autor, Nilsson (1971) y Barr & Feigenbaum (1981) si la función h(n) = 0, el algoritmo A* es el procedimiento de búsqueda primero el de menor coste (apartado 3.3.5.). Rich & Knight (1994) consideran un mayor número de posibilidades: si sólo interesa alcanzar un vértice objetivo sea de la forma que sea, se puede definir g(n) = 0, así se elige el vértice que parece más cercano a un vértice considerado objetivo; si se desea el camino con menor número de pasos se asocia a cada arco un coste 1 (como también expone Ginsberg 1993); si h(n) es una estimación perfecta de $h^*(n)$, entonces el algoritmo A* converge inmediatamente hacia el objetivo (como también comentan Barr & Feigenbaum 1981); y si se garantiza que h(n) nunca sobrestima a $h^*(n)$, el A* garantiza encontrar un camino óptimo hasta un vértice objetivo, si es que existe. Por su parte, Korf (1990), Mahanti & Daniels (1993), Powley et al. (1993) y Zhang & Korf (1995) consideran que el algoritmo A* es un procedimiento de búsqueda primero el mejor (procedimiento descrito en el apartado 3.3.7.) en el que el valor asociado a un vértice se calcula como ya se ha expuesto anteriormente ("*A* ... is a best-first search in which the cost of a node is computed as ...*" Powley et al. (1993, p. 200); "*A special case of best-first search is the A* algorithm, which uses the cost function f(n) = g(n) + h(n), where ...*", Zhang & Korf (1995, p. 248)). Según Greenberg (1996) otro caso particular del A* en investigación operativa es el *branch and bound* para programación entera, donde g(n) + h(n) es el valor objetivo de la relajación lineal en el vértice; nuevamente se advierten imprecisiones en las definiciones, ya que el *branch and bound* utiliza imprescindiblemente el podado mediante cotas, y, para la mayoría de autores, ésta no es una característica principal del A*; la exposición de Ginsberg (1993) también muestra esta imprecisión, ya que considera que si la función h(n) = 0 el procedimiento se conoce como *branch and bound*; cabe destacar sin embargo, que también existen autores que parecen tener clara la distinción entre dichos procedimientos, no exponiendo su coincidencia pero sí su semejanza: "*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).

Según Pearl (1984), para analizar las características de los procedimientos con información heurística se utilizan técnicas probabilísticas. De todas formas, Barr & Feigenbaum (1981), Pearl (1984), Mompín et al. (1987), Shirai & Tsujii (1987), Companys (1989a), Raphael (1990) y Cortés et al. (1993) entre otros, consideran que existe una serie de propiedades que siempre se pueden garantizar para el algoritmo A* (y, según Cortés et al. 1993, en general para cualquier algoritmo de búsqueda con información heurística):

- Completitud (*completeness*): se dice que un algoritmo es completo si siempre encuentra una solución (óptima o no) cuando ésta existe.

- Admisibilidad (*admissibility*): se dice que un algoritmo es admisible si garantiza retornar una solución óptima cuando ésta existe. Los algoritmos A* y de búsqueda en anchura son admisibles (Companys 1989a).

- Dominancia (*dominance*): una algoritmo A_1 domina a A_2, cuando todo vértice expandido por A_1 también es expandido por A_2 (por tanto, A_2 expande al menos tantos vértices como A_1 si la ordenación según el valor de h(n) es la misma para A_1 y A_2). Más concretamente, sean A_1 y A_2 dos algoritmos admisibles con funciones de estimación heurística h_1 y h_2 respectivamente, se dice que A_1 domina a A_2 si $h_1(n) > h_2(n)$ para todo vértice n no objetivo (también se dice que A_1 está mejor informado que A_2).

- Dominancia estricta (*strictly dominance*): una algoritmo A_1 domina estrictamente a un algoritmo A_2, si A_1 domina a A_2 y A_2 no domina a A_1.

Como matiza Companys (1989a), mejor informado no implica forzosamente que sea más eficiente, pues también se debe considerar el trabajo de cálculo de h(n); de todas formas, a menor número de vértices explorados normalmente se obtiene más eficiencia.

- Optimalidad (*optimality*): se dice que un algoritmo es óptimo sobre una clase de algoritmos, si éste domina a todos los miembros de la clase.

Existen autores (Barr & Feigenbaum 1981, Mompín et al. 1987, Companys 1989a, etc.) que también exponen la relación de monotonía o de consistencia: se dice que h satisface la relación de monotonía si para dos vértices n_i y n_j, donde n_j es sucesor de n_i, se cumple que $h(n_i) - h(n_j) \leq c(n_i, n_j)$, donde $c(n_i, n_j)$ es el coste asociado al arco que une n_i y n_j, y h(vértice objetivo) = 0 (de todas formas, o esta propiedad se cumple siempre porque el procedimiento lo garantiza, o se puede hacer cumplir asociando a la cota de un vértice hijo el valor de la cota del vértice padre menos el coste del arco del padre al hijo). Como comentan dichos autores, la relación de monotonía impone que la función h(n) sea

localmente consistente con los costes de los arcos. Si se satisface la restricción de monotonía:

- cuando el algoritmo A* selecciona un vértice n para su expansión, ya ha encontrado un camino óptimo desde el vértice raíz a n: en ese momento $g(n) = g^*(n)$,
- los valores de f(n) de la secuencia de vértices expandidos sucesivamente por A* son no decrecientes.

A modo de resumen se puede decir que cuando un vértice es expandido ya se ha encontrado el camino óptimo hasta él; esta condición que se satisface siempre en los árboles de espacio de búsqueda, también se cumple en un grafo de espacio de estados si se satisface la relación de monotonía.

Korf (1990) expone en su discurso el gran inconveniente del algoritmo A*: como cualquier búsqueda primero el mejor usualmente requiere una gran cantidad de memoria.

Mompín et al. (1987) y Shirai & Tsujii (1987) comentan dos procedimientos mejorados provenientes del algoritmo A*, que intentan evitar expandir un mismo vértice varias veces utilizando funciones heurísticas h(n) cuyo valor va cambiando al ir avanzando la búsqueda; éstos son los algoritmos B y B´, expuestos en el apartado 3.6.16. y desarrollados respectivamente por Martelli & Montanari y Méro (1984). Como introducen Nilsson (1980) y Cortés et al. (1993), modificando la importancia de las funciones que componen f(n) también se pueden obtener nuevas estrategias de búsqueda (denominadas en este texto algoritmos de potencia heurística y expuestos en el apartado 3.6.13.).

3.6.9. Profundización iterativa A* (*iterative deepening A** o IDA*).

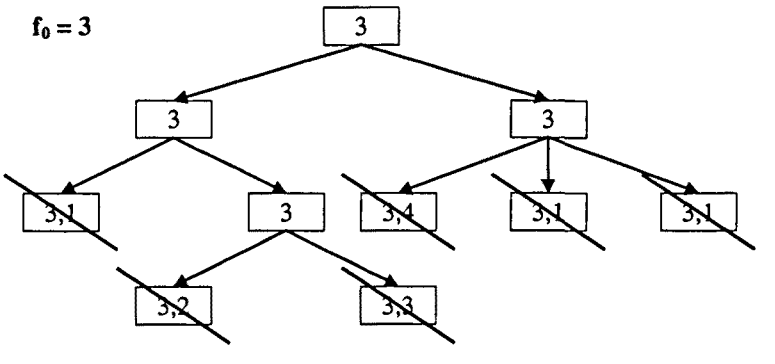
Korf (1985, 1990), Sarkar et al. (1991), Ginsberg (1993) y Rich & Knight (1994), son los autores referenciados que exponen el funcionamiento del procedimiento de búsqueda en espacios de estados denominado profundización iterativa A* (*iterative deepening A** o IDA*).

Como comenta Korf (1985), el algoritmo de profundización iterativa (*iterative deepening*) se puede combinar con una técnica de búsqueda primero el mejor como el A*, lo que proporciona el procedimiento IDA* (que se puede considerar una estrategia de búsqueda basada en la "fuerza bruta"). La idea consiste en ejecutar una búsqueda en profundidad, siempre que el valor del indicador $f(n) = g(n) + h(n)$ asociado a los vértices explorados (donde g(n) indica el coste real de alcanzar el vértice n y h(n) se refiere a una

estimación del coste que falta hasta alcanzar un vértice objetivo), no exceda de un coste umbral de corte f_0 dado (así los vértices con $f(n) > f_0$ no son explorados); si se encuentra la solución buscada en esta iteración la búsqueda finaliza, sino el árbol es explorado de nuevo y desde el inicio, con una búsqueda en profundidad y un nuevo coste umbral de corte f_0 que coincide, según Korf (1985) y Rich & Knight (1994), con el valor mínimo de f(n) para todos los vértices cuya f(n) excede el valor del coste umbral de la iteración anterior; y así sucesivamente. Como aclara Ginsberg (1993), las iteraciones individuales se realizan utilizando la búsqueda en profundidad y no el algoritmo A*: la función heurística se utiliza para decidir los vértices que están por debajo del valor del coste umbral de corte, pero no para determinar el orden en el que deben ser explorados. Si la función heurística h(n) es admisible, y por tanto nunca sobrestima la distancia real, el procedimiento IDA* garantiza encontrar la solución óptima.

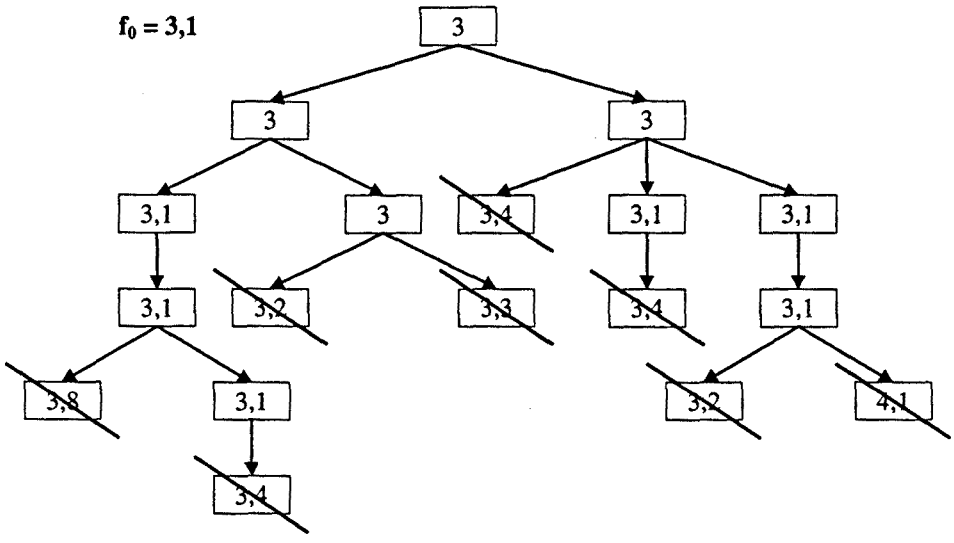
El gráfico siguiente muestra un ejemplo de cómo actúa la búsqueda de profundización iterativa A*, en la generación y exploración de los estados de un espacio de búsqueda; los números asociados a cada estado especifican el valor de f(n) del vértice referenciado.

* Iteración 1: se fija f_0 al valor de la cota del vértice raíz y se realiza una búsqueda en profundidad no considerando los vértices con $f(n) > f_0$.



Ejemplo de actuación de la búsqueda de profundización iterativa A*: iteración 1.

- Iteración 2: se fija f_0 al valor mínimo de f(n) de aquellos vértices cuya f(n) excede al valor de f_0 de la iteración 1 (3,1) y se hace una búsqueda en profundidad no considerando los vértices con $f(n) > f_0$.



Ejemplo de actuación de la búsqueda de profundización iterativa A*: iteración 2.

* Iteración 3: se fija f_0 al valor mínimo de f(n) de aquellos vértices cuya f(n) excede al valor de f_0 de la iteración 2 (3,2) y se vuelve a iterar hasta alcanzar un vértice considerado objetivo.

Como comentan Korf (1990), Ginsberg (1993) y Rich & Knight (1994), la profundización iterativa puede proporcionar un gran servicio a los algoritmos A*, ya que mientras la principal dificultad de éstos es la gran cantidad de memoria que necesitan, el algoritmo IDA* requiere considerablemente menos memoria que el A* sin sacrificar la optimalidad de la solución encontrada; además según Korf (1990), es más fácil de implementar. Por otro lado, Mahanti & Daniels (1993) consideran que tanto el algoritmo A* como el IDA*, pueden ser considerados procedimientos de búsqueda con estrategias primero el mejor ("*It may be noted that A* and IDA* may both be referred to as best-first search strategies ...*", p. 244).

3.6.10. Profundización iterativa A* con control de las reexpansiones (*iterative deepening with controlled reexpansion* o IDA*_CR).

Como se puede comprobar al analizar el procedimiento IDA*, esta técnica realiza una gran cantidad de reexpansiones. Con el objetivo de reducir el número de reexpansiones que se presentan en la ejecución del procedimiento IDA*, Sarkar et al. (1991) proponen una modificación de esta técnica, que denominan profundización iterativa A* con control de las reexpansiones (*iterative deepening with controlled reexpansion* o IDA*_CR).

La idea principal que se incorpora en el IDA*_CR respecto al IDA*, consiste en definir el valor de los sucesivos costes umbrales de corte, f_0, de forma que se garantice la

expansión de un número suficiente de vértices, limitando de esta forma las reexpansiones. De esta manera, y como exponen sus autores, el IDA*_CR es un procedimiento IDA* en el que se realizan los dos siguientes cambios:

1) Por un lado, en vez de realizar en cada iteración una búsqueda en profundidad hasta que el valor del indicador excede el coste umbral de corte dado, f_0, se realiza una búsqueda en profundidad a la que se le incorpora un procedimiento de podado por cotas (procedimiento *depth-first branch and bound*, expuesto en el apartado 3.5.6.). De esta manera, se garantiza la optimalidad de la solución obtenida.

2) La segunda variación se refiere al cálculo iterativo del coste umbral de corte, f_0. Sea b un parámetro de control del vértice, que habitualmente toma el valor del factor de ramificación (número de vértices sucesores de un vértice dado). Se parte de la premisa siguiente: en la iteración k se debe intentar garantizar que al menos b^k nuevos vértices serán expandidos en la iteración k+1. Para ello, el valor de f_0 en la iteración k+1, debería tomar el b^k valor de la secuencia de valores del indicador f(n) que exceden el valor f_0 en la iteración k, en orden ascendente; pero según los autores, este proceso puede consumir mucho tiempo. De esta manera, lo que realmente se hace es aumentar el valor de f_0 en un %, porcentaje que se determina en función del número de vértices que en esa iteración k sobrepasan en diferentes % (y como matizan los autores, con una diferencia mínima de un 1%) el valor del coste umbral de corte, f_0[11].

En Korf (1993) se presenta el funcionamiento del procedimiento IDA*_CR de una forma levemente diferente a la exposición que realizan sus autores: se van fijando los sucesivos costes umbrales de corte a valores mayores que el valor mínimo que excede al coste umbral de la iteración anterior, con lo que se reduce el número de iteraciones al combinar varias a la vez; para garantizar la optimalidad de la solución obtenida, una vez se genera una solución factible candidata a ser óptima, la búsqueda en profundidad se convierte en un procedimiento *depth-first branch and bound* que va podando vértices hasta encontrar la solución óptima.

Este mismo autor (Korf 1993) comenta que las ideas desarrolladas en el procedimiento IDA*_CR también se pueden aplicar en una búsqueda recursiva primero el mejor (*recursive best-first search* o RBFS), técnica descrita en el apartado 3.6.5. De esta manera, se fijaría el coste umbral de corte local a un valor ligeramente superior al del hermano con mejor valor asociado; una vez localizada una solución, el algoritmo continuaría trabajando, con el coste umbral de corte actual, también en todos los subproblemas hermanos restantes para poder asegurar la optimalidad de la solución obtenida.

[11] En Sarkar et al. (1991) se expone el procedimiento particular de cálculo del % de incremento que proponen estos autores.

Partiendo del procedimiento IDA*_CR, Sarkar et al. (1991) proponen una versión heurística que denominan IDA*_CRA. Esta técnica acelera la ejecución del IDA*_CR pero a expensas de garantizar la optimalidad de la solución obtenida; de todas formas, proporciona una estimación de la calidad de dicha solución. El procedimiento finaliza la búsqueda cuando al concluir una iteración dispone de una solución factible; además, garantiza un *gap* máximo del p% respecto al valor de la solución óptima, ya que el incremento del coste umbral de corte, f_0, viene determinado por el mínimo entre el % que se determina en el procedimiento IDA*_CR y el valor p (valor fijado de partida para acotar la calidad de la solución obtenida).

3.6.11. Algoritmo MREC.

Korf (1993) describe el procedimiento MREC, un algoritmo de búsqueda con memoria limitada diseñado por Sen & Bagchi (1989) para reducir las reexpansiones típicas del procedimiento IDA*, como una técnica que ejecuta el algoritmo A* hasta que la memoria disponible en la que se guardan los vértices activos está casi llena; a partir de este momento, se ejecuta el algoritmo IDA* en cada uno de los vértices frontera que están almacenados. Según Korf (1993), aunque no se especifica en la descripción que se realiza en Sen & Bagchi (1989), se pone en práctica un procedimiento de podado, que consiste en la eliminación de los vértices duplicados al comparar los vértices generados en el procedimiento IDA* con los que están almacenados.

Según Sen & Bagchi (1989), el procedimiento MREC combina las buenas características de los algoritmos A* y IDA*. Además, la cantidad de memoria disponible se puede parametrizar con un parámetro M, de forma que:

- con $M = 0$, MREC es idéntico al procedimiento IDA*,

- con M suficientemente grande ($M = \infty$), MREC es tan rápido como el algoritmo A* y los vértices sólo se expanden una vez,

- cuando $0 < M < \infty$, se reexpanden muchos menos vértices que con el algoritmo IDA*, lo que resulta provechoso en problemas en los que el tiempo de expansión es elevado.

Sen & Bagchi (1989), también comentan que MREC es capaz de desempeñar las funciones de A* y de IDA*, y que puede ser visto como una generalización del algoritmo IDA*.

Como expone Korf (1993), el procedimiento MREC puede ser combinado con la búsqueda recursiva primero el mejor (descrita en el apartado 3.6.5.), de forma que se

genera una tabla con vértices frontera, que sirve para probar y localizar vértices duplicados entre los generados recientemente.

3.6.12. Algoritmo MA*.

Como expone Korf (1993), mientras que en el procedimiento MREC se guardan en memoria de forma estática los primeros vértices frontera, el algoritmo MA* almacena dinámicamente los mejores vértices generados hasta el momento. Este procedimiento, en el que en la expansión de un vértice únicamente se genera el mejor de los vértices descendientes todavía no generado (Chakrabarti et al. 1989), se comporta de manera semejante al algoritmo A* hasta que la memoria está casi llena; entonces, para liberar espacio y poder continuar la búsqueda, se poda el vértice de mayor coste de la lista de vértices abiertos (el menos prometedor), actualizando el valor de su vértice padre al valor de su hijo y, si es necesario, cambiando de nuevo el vértice padre, de la lista de vértices cerrados a la lista de vértices abiertos.

Según Chakrabarti et al. (1989), sus creadores, el procedimiento MA* es un procedimiento de búsqueda con memoria restringida, en el que la cantidad de memoria disponible (necesaria para mantener la lista de vértices abiertos y la de vértices cerrados) se puede parametrizar con un parámetro denominado MAX. Cuando el tamaño de la memoria es excedido comienza la poda, y cabe destacar que un vértice puede llegar a ser podado y reexpandido varias veces. Según estos autores, si MAX es suficientemente grande MA* trabaja como el algoritmo A*, y si M = 0 trabaja de forma parecida al procedimiento IDA*.

Para Korf (1993), el algoritmo MA* proporciona resultados mucho peores que el IDA*. Y si fuera práctico, también se podría combinar con la búsqueda recursiva primero el mejor (RBFS, descrita en el apartado 3.6.5.), de forma que se generara una tabla con vértices frontera que sirviera para probar y localizar vértices duplicados entre los generados recientemente.

3.6.13. Algoritmos de potencia heurística.

Como ya se ha comentado al describir las funciones de evaluación usualmente utilizadas en la literatura, para Pearl (1984), el efecto de g(n) frente al de h(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* o HPA, procedimiento de búsqueda

en espacio de estados que selecciona para expandir, en caso de minimización, el vértice de menor valor de 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 $f(n) = h(n)$ (procedimiento de búsqueda primero el de mejor cota si h(n) es admisible, o, en general, búsqueda primero el mejor), haciendo $\omega = 0$ la función de evaluación resultante es $f(n) = g(n)$ (búsqueda primero el de menor coste, o búsqueda en anchura si g(n) evalúa distancias en número de etapas o pasos) y fijando $\omega = 0{,}5$ se obtiene la función de evaluación descrita inicialmente $f(n) = g(n) + h(n)$ (procedimiento primero el mejor como por ejemplo el algoritmo A*). Variando ω entre 0 y 1 se pueden diseñar infinitas variantes de un mismo procedimiento que incorpore esta función de evaluación (Pearl 1984).

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 decidir 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; en cambio, 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 y garantizar que se encuentra 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.

Como ya se ha comentado, el concepto de potencia heurística de un algoritmo de exploración se define para valorar la influencia de la información heurística en la búsqueda. Nilsson (1980) es más concreto y especifica medidas de la potencia heurística:

el poder heurístico de un procedimiento de búsqueda depende enormemente de un conjunto de características particulares del problema, por tanto se pueden calcular ciertas medidas que, aunque no determinan completamente la potencia heurística, pueden ser usadas para comparar diferentes procedimientos de búsqueda:

- Penetración/profundización P: valor que indica cómo se ha focalizado la búsqueda hacia el objetivo y que se define como sigue:

$$P = L / T,$$

donde L es la longitud del camino encontrado hasta el vértice objetivo (en número de etapas) y T es el número total de vértices generados. Si se encuentra el óptimo directamente P vale 1, pero como señala Nilsson (1980) son frecuentes valores mucho más pequeños que 1. P puede dar idea de lo alargada o ancha que es la arborescencia de búsqueda, y depende tanto de la dificultad del ejemplar como de la eficiencia de la estrategia de búsqueda.

- Factor de ramificación B: es el número constante de sucesores que tendría cada vértice en una arborescencia, en la que la profundidad fuera igual a la longitud del camino óptimo en número de etapas y el número total de vértices igual al número total de vértices generados durante la búsqueda. Se define:

$$B + B^2 + ... + B^L = T$$
$$[B^L - 1] \cdot B / (B - 1) = T$$

Un valor de B cercano a la unidad se corresponde con una búsqueda muy bien enfocada hacia el vértice objetivo y con muy pocas ramificaciones en otras direcciones; una búsqueda mal focalizada tendrá un valor alto de B.

- Por último Nilsson (1980) comenta que se puede encontrar una relación entre ambos factores:

$$P = L \cdot (B - 1) / B \cdot [B^L - 1]$$

3.6.14. Algoritmo A* ponderado (*weighted-A** o WA*).

El algoritmo A* ponderado (*weighted-A** o WA*) consiste en un procedimiento A* en el que en vez de que seleccionar para expandir el vértice n de menor valor de la función:

f(n) = g(n) + h(n) (descrita en el apartado 3.6.8.),

se selecciona el vértice de menor valor de la siguiente función ponderada:

$$f(n) = \omega_g \cdot g(n) + \omega_h \cdot h(n) \equiv g(n) + W \cdot h(n),$$

función de evaluación introducida en el apartado 3.6.13. y donde $W = \omega_h / \omega_g$ (Korf 1993).

Como se puede deducir inmediatamente, si W = 1 el procedimiento WA* es el procedimiento A*, o, desde otro punto de vista, el procedimiento A* es un algoritmo WA* con W = 1.

3.6.15. Algoritmo IDA* ponderado (*weighted-IDA** o WIDA*).

Según se describe en Korf (1993), el algoritmo IDA* ponderado (*weighted-IDA** o WIDA*) consiste en un procedimiento IDA* en el que en vez de seleccionar para expandir el vértice n de menor valor de la función utilizada en el procedimiento A*, se elige el vértice de menor valor de la función de evaluación utilizada en el algoritmo WA*.

En este caso también se puede comprobar que si W = 1, el procedimiento WIDA* se transforma en el procedimiento IDA*, o que el procedimiento IDA* es un algoritmo WIDA* con W = 1.

3.6.16. Algoritmo B y B'.

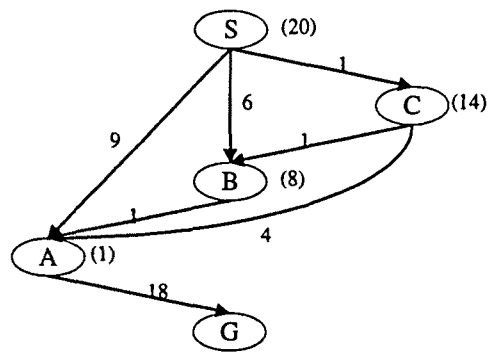
Como ya se ha comentado, el A* es un algoritmo clásico en inteligencia artificial para encontrar el camino de mínimo coste en un grafo desde un vértice inicial hasta un conjunto de vértices considerados objetivo, utilizando estimaciones heurísticas. Mompín et al. (1987) y Shirai & Tsujii (1987) comentan que en el algoritmo A* algunos vértices son expandidos en más de una ocasión; además, Shirai & Tsujii (1987) exponen una mejora de éste procedimiento llamada algoritmo B (introducida por Martelli & Montanari según Méro (1984)), que hace uso de la información heurística sólo cuando ésta está en concordancia con el subgrafo establecido por la búsqueda hasta el estado dado (en este camino, el algoritmo B intenta evitar malas estimaciones heurísticas que desorientan la búsqueda).

Sea $c(n_i, n_j)$ un coste positivo asociado a cada arco (n_i, n_j), $f(n) = g(n) + h(n)$ una función de evaluación, g(n) una estimación del coste del camino de mínimo coste desde el vértice inicial hasta el vértice n, y h(n) una estimación heurística del coste mínimo desde n hasta un vértice considerado objetivo. Como ya se ha comentado, si h(n) es una cota inferior de $h^*(n)$ (coste de un camino de mínimo coste desde el vértice n hasta un vértice

objetivo), entonces el algoritmo A* es admisible y siempre encuentra un camino óptimo. Además, y como expone Méro (1984), si se cumple para todo arco (n_i, n_j) la relación de consistencia o también llamada de monotonía $(h(n_i) - h(n_j) \leq c(n_i, n_j))$, entonces el algoritmo A* nunca reabre un vértice cerrado (vértice ya explorado).

Si no se cumple la relación de monotonía, Martelli & Montanari comentan que el número de reaperturas utilizando el algoritmo A* puede crecer exponencialmente con el número de vértices del espacio de estados. Para evitar que el algoritmo expanda el mismo vértice varias veces, Martelli & Montanari introducen el algoritmo B: se selecciona el vértice de menor f(n) de la lista de vértices abiertos (vértices generados pero todavía no explorados) y se fija $f_m = f(n)$; se expande el vértice seleccionado y los nuevos vértices con $f(n) < f_m$ se incluyen en una nueva lista de vértices abiertos; si la nueva lista no está vacía se selecciona el vértice de menor g(n) y se expande, y los nuevos vértices con $f(n) < f_m$ se incluyen en la nueva lista; si la nueva lista está vacía se selecciona el vértice de la lista de vértices abiertos con menor f(n), se fija $f_m = f(n)$ y se expande; y así sucesivamente.

Sea el siguiente ejemplo propuesto por Shirai & Tsujii en 1987, en el que se desea encontrar el camino de coste mínimo desde el vértice inicial S hasta el vértice objetivo G. En el grafo se muestra el coste asociado a los arcos y entre paréntesis el valor h(n), cota inferior de $h^*(n)$, asociado a cada vértice n.



Ejemplo de camino de mínimo coste de S a G, de Shirai & Tsujii (1987).

La tabla siguiente muestra el proceso de búsqueda seguido por un procedimiento A* que selecciona para expandir el vértice de menor valor de f(n) = g(n) + h(n). Se comienza generando el vértice S que tiene asociado un valor f(n) = 0 + 20 = 20; como éste es el único vértice activo, se expande generándose los vértices A (con f(n) = 9 + 1 = 10 y proveniente de S), B (con f(n) = 6 + 8 = 14 y proveniente de S) y C (con f(n) = 1 + 14 = 15 y proveniente de S); en este instante se expande el vértice A, vértice no explorado de menor valor de f(n), generándose el vértice G de valor f(n) = 27 + 0 = 27; ahora el vértice de menor valor de f(n) no explorado es el vértice B, así que se expande generando de nuevo el vértice A con un valor f(n) = 7 + 1 = 8, por lo que se conserva

esta trayectoria proveniente del vértice B y se elimina la anterior proveniente de S; como el vértice A es el de menor valor de f(n), se vuelve a generar el vértice G ahora con un valor f(n) = 25 + 0 = 25; y así sucesivamente.

S	A	B	C	G
0+20				
0+20	9+1 (S)	6+8 (S)	1+14 (S)	
0+20	9+1 (S)	6+8 (S)	1+14 (S)	27+0 (A)
0+20	7+1 (B)	6+8 (S)	1+14 (S)	27+0 (A)
0+20	7+1 (B)	6+8 (S)	1+14 (S)	25+0 (A)
0+20	5+1 (C)	2+8 (C)	1+14 (S)	25+0 (A)
0+20	5+1 (C)	2+8 (C)	1+14 (S)	23+0 (A)
0+20	3+1 (B)	2+8 (C)	1+14 (S)	23+0 (A)
0+20	3+1 (B)	2+8 (C)	1+14 (S)	21+0 (A)
0+20	**3+1 (B)**	**2+8 (C)**	**1+14 (S)**	**21+0 (A)**

Búsqueda según el procedimiento A*, de Shirai & Tsujii (1987).

En la tabla siguiente se puede observar la evolución del proceso de búsqueda seguido por el algoritmo B, si se desea encontrar el camino de coste mínimo desde el vértice inicial S hasta el vértice objetivo G. Se comienza generando el vértice S que tiene asociado un valor f(n) = 0 + 20 = 20; como éste es el único vértice activo, se fija un valor de f_m = 20 y se expande, generándose los vértices A (con f(n) = 9 + 1 = 10 y proveniente de S), B (con f(n) = 6 + 8 = 14 y proveniente de S) y C (con f(n) = 1 + 14 = 15 y proveniente de S); en este instante la lista de vértices con f(n) < f_m está compuesta por los vértices A, B y C, por lo que se selecciona para expandir el vértice C que es el de menor valor de g(n), y se generan de nuevo los vértices A (ahora con f(n) = 5 + 1 = 6 y proveniente de C al presentar una trayectoria de menor coste asociada) y B (ahora con f(n) = 2 + 8 = 10 y proveniente de C al presentar también una trayectoria de menor coste); en este momento la lista de vértices con f(n) < f_m está compuesta por los vértices A y B, por lo que se selecciona para expandir el vértice de menor valor de g(n) que es en este caso es el vértice B; y así sucesivamente.

S	A	B	C	G	f_m	Lista f(n)<f_m
0+20					0	
0+20	9+1 (S)	6+8 (S)	1+14 (S)		20	A, B, C
0+20	5+1 (C)	2+8 (C)	1+14 (S)		20	A, B
0+20	3+1 (B)	2+8 (C)	1+14 (S)		20	A
0+20	3+1 (B)	2+8 (C)	1+14 (S)	21+0 (A)	20	
0+20	**3+1 (B)**	**2+8 (C)**	**1+14 (S)**	**21+0 (A)**	**21**	

Búsqueda según el procedimiento B, de Shirai & Tsujii (1987).

Como se puede comprobar, el número de vértices expandidos con el procedimiento B hasta alcanzar la solución óptima es menor que el número de vértices explorados con el procedimiento A*.

Si existen vértices en la lista de vértices abiertos cuya f(n) es menor que la f_m en curso, el procedimiento B se comporta como una búsqueda primero el de menor coste. Si la lista de vértices abiertos cuya f(n) es menor que la f_m en curso está vacía, el procedimiento B se comporta como una búsqueda primero el mejor.

Méro (1984) propone una versión mejorada del algoritmo B, que denomina procedimiento B´, que se basa en considerar el estimador h(n) como un valor variable más que constante, y en intentar mejorar dicha estimación heurística utilizando la experiencia reunida por los vértices expandidos. Según Méro (1984), el algoritmo B´ mejora la estimación evitando expansiones inútiles, nunca expande más vértices que B o A* y expande un número mucho menor en algunos casos.

Concretamente, el algoritmo B´ se basa en la observación de que la expansión de un nuevo vértice puede permitir mejorar la estimación heurística de ese vértice y de sus vértices hijos; el algoritmo B´ es una variante del algoritmo B y se puede obtener a partir de éste insertando los siguientes pasos:

- Para cada hijo n_j del vértice n_i, si $h(n_j) < h(n_i) - c(n_i, n_j)$, entonces hacer $h(n_j) = h(n_i) - c(n_i, n_j)$.
- Sea n_j el hijo de n_i para el cual $h(n_i) + c(n_i, n_j) = h_m(n_i)$ es mínimo; si $h_m(n_i) > h(n_i)$, entonces hacer $h(n_i) = h_m(n_i)$.

De todas formas, Méro (1984) también comenta que el algoritmo B´ puede obtenerse directamente del algoritmo A* insertándole dichos pasos.

Según Méro (1984), suponiendo que los arcos del grafo tienen asociado costes positivos y que las estimaciones heurísticas iniciales de cada vértice son cotas inferiores ($h(n) \leq h^*(n)$), entonces:

- el algoritmo B´ es admisible,
- para cada grafo, el algoritmo B´ expande cada vértice como mucho tantas veces como el algoritmo B, si ambos resuelven los empates de la misma forma.

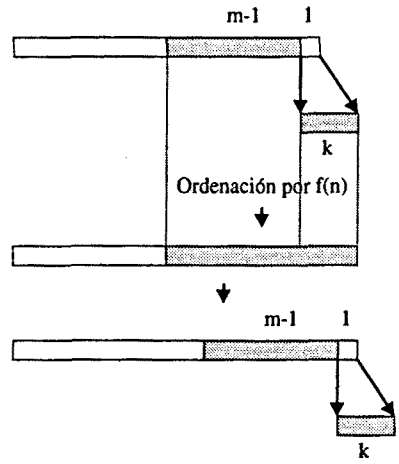
A pesar de las ventajas que pueda presentar el algoritmo B´ y según el propio Méro (1984), el procedimiento para mejorar la estimación heurística h(n) consume mayor tiempo, tiempo que puede llegar a ser significativo en casos simples. De todas formas y como comenta su autor, el tiempo de cálculo de la estimación heurística se omite, ya

que se considera que el tiempo consumido por la expansión es mucho mayor que éste, y así la complejidad del algoritmo se mide únicamente por el número total de vértices expandidos, que está de acuerdo con el uso general en la literatura de inteligencia artificial. Cabe recordar sin embargo, que en investigación operativa y para casos reales, la calidad de un algoritmo se debería medir considerando el tiempo total consumido para solucionar un problema con relación al tiempo disponible, por lo que el hecho de no considerar el tiempo consumido puede llegar a ser nefasto en contextos industriales.

3.6.17. Búsqueda en profundidad-m (*depth-m search*).

Procedimiento descrito por Ibaraki (1988), que consiste en seleccionar para expandir el vértice de mejor valor de entre los m últimos vértices activos de una lista de vértices activos ordenados según f(n) (siendo f(n) una estimación del coste necesario para alcanzar un vértice objetivo desde el vértice raíz y pasando por el vértice n).

El gráfico siguiente muestra un ejemplo de cómo actúa la búsqueda en profundidad-m, en la generación y exploración de los estados de un espacio de búsqueda:



Generación y exploración de estados en una búsqueda en profundidad-m, de Ibaraki (1988).

El vértice de mejor valor del indicador f(n) asociado, de entre un conjunto de m vértices, es expandido generándose k vértices sucesores; estos nuevos vértices, junto con los m-1 vértices no seleccionados para expandir en esta iteración, son ordenados en función de f(n), conservándose como vértices activos los m mejores y expandiéndose en la siguiente iteración el de mejor indicador f(n) dentro de estos m vértices.

Según Ibaraki (1988), este procedimiento constituye una opción interesante cuando el espacio de memoria está restringido, ya que se limita el tamaño del conjunto de vértices activos; si m = 1 equivale a la búsqueda en profundidad (desempatando entre los vértices de un mismo nivel en función del valor de la función f(n)) y si m = ∞ equivale a

una búsqueda primero el mejor. Además, este mismo autor señala que todo el procedimiento anterior también es válido si en vez de trabajar con una función f(n) cualquiera se impone que f(n) cumpla la condición de admisibilidad.

A pesar de la explicación anterior, el funcionamiento de este procedimiento no está claro, ya que se presentan diversos interrogantes no resueltos en Ibaraki (1988): ¿qué se hace en cada iteración con los vértices que no son seleccionados entre los m mejores?, ¿se guardan en una memoria auxiliar y cuando se necesita un vértice para completar la lista de longitud m se recurre a ellos (por ejemplo al haber llegado a un vértice terminal y tener que hacer *backtracking*)?, o ¿son eliminados y este procedimiento es una técnica heurística que finaliza cuando se encuentra un vértice considerado objetivo o la lista de m vértices se queda vacía?

3.6.18. Búsqueda en anchura limitada (*breadth-limited search* o BLIS).

Procedimiento descrito por Ibaraki (1988) que consiste en guardar en cada nivel w vértices y elegir aquél de mejor indicador f(n) (siendo f(n) una estimación del coste necesario para alcanzar un vértice objetivo desde el vértice raíz y pasando por el vértice n), siempre que sus sucesores quepan en el nivel siguiente y se puedan guardar.

Según Ibaraki (1988), si w = k (donde k es el número de hijos de cualquier vértice) este procedimiento equivale a la búsqueda en profundidad (desempatando entre los vértices de un mismo nivel en función del valor del indicador f(n)) y si w = ∞ equivale a una búsqueda primero el mejor.

3.6.19. Procedimientos de búsqueda *barried method.*

Un concepto que es esencial tener presente cuando se utilizan procedimientos de búsqueda en contextos industriales, es que el tamaño de memoria disponible (M) es limitado y conocido, ya que éstos se ejecutan en ordenadores reales.

Aunque algunos procedimientos descritos por Ibaraki (1988), como la búsqueda en profundidad-m (*depth-m search*), la búsqueda en anchura limitada (*breadth-limited search* o BLIS) o el procedimiento *parallel depth first search* (PDFS), tienen algún control sobre la memoria utilizada, se pueden combinar diversas estrategias y, como expone dicho autor, obtener dos nuevos tipos de procedimientos que permiten controlar este parámetro:

(1) *Hard barried method*: son combinaciones de procedimientos de búsqueda primero el mejor (donde f(n) cumple, o como otra opción no cumple, la condición de admisibilidad) y de búsquedas en profundidad en las que para desempatar entre dos

vértices de un mismo nivel se utiliza la función f(n) (que cumple, o como otra opción no cumple, la condición de admisibilidad). Estas combinaciones pueden dar lugar a los siguientes procedimientos:

a) Utilizar búsquedas primero el mejor mientras exista memoria y sino usar búsquedas en profundidad.
b) Utilizar búsquedas en profundidad mientras el número de vértices activos sea menor que un cierto valor, que puede depender de la máxima profundidad alcanzada y del número de hijos generados por cada vértice, y aplicar búsquedas primero el mejor al vértice activo más profundo y a sus sucesores. Ibaraki (1988) opina que este procedimiento es bueno si se desea explorar en niveles profundos y etapas tempranas de la computación.

(2) *Soft barried method*: procedimiento que consiste en controlar dinámicamente el parámetro m de un procedimiento de búsqueda en profundidad-m (*depth-m search*), ordenado por f(n) que cumple, o como otra opción no cumple, la condición de admisibilidad. El funcionamiento del procedimiento resultante consiste en forzar a que m adopte un valor grande si la memoria disponible es grande (calculada por ejemplo, como M - número de vértices activos), y adopte un valor pequeño cuando el espacio disponible es pequeño. Según el propio Ibaraki, este procedimiento también es posible con la búsqueda en anchura limitada (*breadth-limited search* o BLIS) y el procedimiento *parallel depth first search* (PDFS), ya que ambos contienen parámetros de control del tamaño de memoria necesario.

3.7. Procedimientos de búsqueda híbridos.

Como se ha podido comprobar, a partir de un conjunto de procedimientos, estrategias, características y técnicas básicas, se pueden describir la mayoría de procedimientos de búsqueda existentes. Muchos de estos procedimientos utilizan varias técnicas y/o estrategias de forma simultánea, por lo que se podrían considerar procedimientos de búsqueda híbridos. Recordando la gran dificultad de realizar una clasificación precisa de las técnicas de exploración del espacio de estados, es fácil comprender que en los apartados anteriores ya se hayan definidos diversos de estos procedimientos, englobándolos y comentándolos en otros conjuntos de técnicas. A continuación se describen algunos otros procedimientos de búsqueda que parecen ser considerados claramente por sus autores como procedimientos híbridos.

De todas formas, y considerando que un procedimiento de búsqueda híbrido es una técnica de exploración que incorpora diversas técnicas que podrían ser consideradas como "puras", incluso aquí se produce inexactitud en las definiciones de las mismas.

3.7.1. Algoritmo a pila.

Mompín et al. (1987) definen de forma vaga y muy imprecisa el algoritmo a pila, comentado que puede considerarse como un algoritmo híbrido preferente en anchura-profundidad, aunque dirigido por una estrategia primero el mejor.

3.7.2. Estrategias híbridas como combinación de las estrategias *hill climbing* (HC), *backtracking* (BT) y *best first* (BF).

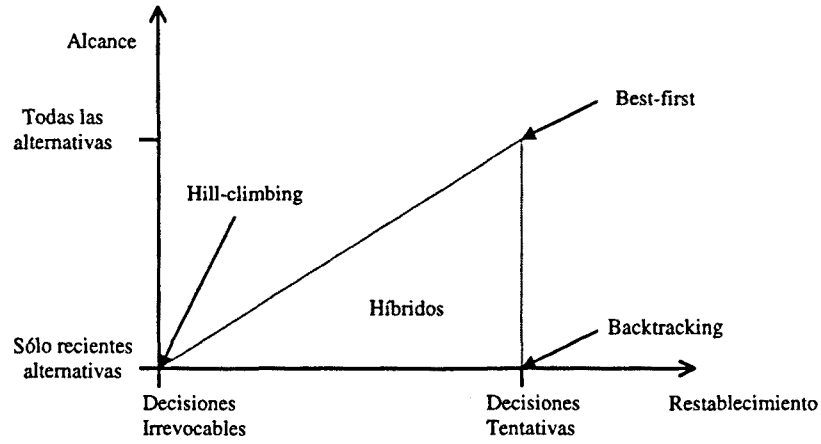
Pearl (1984) especifica un conjunto de estrategias híbridas resultado de la combinación de las estrategias *hill climbing* (HC), *backtracking* (BT) y *best first* (BF). A continuación, se expone una definición de estas tres estrategias:

- Estrategia de búsqueda *hill climbing* (HC): técnica consistente en expandir un vértice y en seleccionar para explorar el mejor de sus sucesores, si es que existen; de esta manera, este procedimiento no garantiza la obtención de soluciones factibles.

- Estrategia de *backtracking* (BT): con este nombre Pearl (1984) se refiere a la búsqueda en profundidad (ampliamente especificada en el apartado 3.3.3.): explorar siempre un vértice hijo del vértice expandido más recientemente, y si éste no tiene sucesores, realizar un proceso de *backtracking* reinicializando la búsqueda en el vértice antecesor más cercano que posea vértices hijos sin explorar.

- Estrategia de búsqueda *best first* (BF): estrategia definida en el apartado 3.3.7., 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; de esta forma, el movimiento de avance se realiza sin importar dónde está el vértice seleccionado en el árbol parcialmente desarrollado, únicamente se tiene en cuenta el valor de la función de evaluación.

Pearl (1984) realiza una representación bidimensional de dos características que pueden presentar estas técnicas y especifica dónde se pueden encuadrar las estrategias *hill climbing*, *backtracking* y *best first*:



Restablecimiento de las decisiones vs alcance de las alternativas para las búsquedas HC, BT y BF, de Pearl (1984).

Como comenta el mismo autor, en el eje de ordenadas se representa la posibilidad de restablecimiento de la búsqueda: el grado con el que se pueden recuperar alternativas abandonadas con anterioridad, presentando estrategias basadas en decisiones irrevocables (*hill climbing*) y otras que presentan decisiones tentativas (*backtracking*, *best first* y otras como la búsqueda en anchura -especificada en el apartado 3.3.4.- o la búsqueda primero el de menor coste -definida en el apartado 3.3.5.-). Por otro lado, en el eje de coordenadas se representa el alcance de la evaluación: el número de alternativas (vértices en este caso) consideradas en cada decisión, presentando estrategias con selección sólo entre los vértices recientes (*hill climbing*, *backtracking* y otras como la búsqueda en anchura) y otras que incorporan todos los vértices ya generados (*best first*, búsqueda primero el de mejor cota -especificada en el apartado 3.3.6.-, etc.).

Concretamente, Pearl (1984) describe los siguientes tipos de estrategias híbridas:

a) Aplicación de BF en el inicio del grafo y de BT en el final: se comienza la exploración aplicando una estrategia BF en el vértice raíz; cuando la memoria se agota se aplica una estrategia BT desde el mejor de entre todos los vértices abiertos y hasta encontrar una solución; si no se encuentra ninguna solución se toma el segundo vértice abierto mejor, y así sucesivamente.

b) Aplicación de BT en el inicio del grafo y de BF en el final: la búsqueda comienza aplicando una estrategia BT en el vértice raíz hasta que se alcanza una profundidad d_0; en el vértice de profundidad d_0 se utiliza una búsqueda BF hasta obtener una solución o hasta finalizar sin obtener nada; si no se obtiene ninguna solución se utiliza BT hasta alcanzar de nuevo la profundidad d_0 y a partir de aquí de nuevo BF. La elección de d_0 debe tener en cuenta que la búsqueda BF no exceda la memoria disponible.

c) Aplicación de BF de forma local y BT de forma global: se comienza con una estrategia BF hasta que se acaba una memoria M_0; en este punto se consideran todos los vértices abiertos como sucesores directos del vértice raíz y se inicia una estrategia BT, donde BT selecciona el mejor vértice de entre estos sucesores y lo expande usando BF, esto es, se refiere al vértice elegido a una búsqueda local BF hasta que una nueva memoria M_0 es consumida y trata a los nuevos vértices abiertos como sucesores directos del vértice expansionado. Aunque no se especifica, parece que se van guardando toda la información para poder hacer el BT.

d) BF con HC: en este procedimiento híbrido no se mantiene todo el árbol que se crea con una búsqueda BF, únicamente se va manteniendo el subárbol más prometedor. Concretamente consiste en hacer una búsqueda BF hasta que se agota la memoria, momento en el que sólo se guardan los mejores vértices de abiertos y sus caminos, descartando todo lo demás; la búsqueda continúa hasta que se vuelve a agotar la memoria, momento en el que se vuelve ha hacer una selección y así sucesivamente.

e) Aplicación de BF de forma local y BT de forma global con HC: técnica que consiste en la aplicación de una búsqueda BF de forma local y una estrategia BT de forma global, pero sólo guardando un subconjunto de los vértices en el momento que se agota la memoria M_0; así, en este punto se consideran sólo un conjunto de los mejores vértices abiertos como sucesores directos del vértice expansionado y se inicia una estrategia BT, donde BT selecciona el mejor vértice de entre el conjunto de sucesores y lo expande usando BF.

Según expone Pearl (1984), Ibaraki propone otro esquema híbrido: mientras que la búsqueda en profundidad sólo considera para expandir los nuevos vértices generados, este esquema incorpora al conjunto de vértices seleccionables para expandir (es decir, los nuevos sucesores), los k mejores vértices del conjunto considerado previamente. Por tanto, el conjunto de vértices que puede ser seleccionados para un vértice t es el formado por los descendientes directos más los k mejores vértices del vértice antecesor. Si $k = 0$ el procedimiento resultante constituye una búsqueda en profundidad y si $k = \infty$ se obtiene búsqueda primero el mejor.

3.7.3. Procedimientos de *branch and bound* en programación dinámica discreta o programación dinámica discreta en procedimientos de *branch and bound*.

Como comenta Kumar (1990), el *branch and bound* está íntimamente ligado a la programación dinámica discreta: históricamente, la programación dinámica utilizaba dominancias para podar, pero no cotas; en las primeras formulaciones del *branch and bound* se usaban cotas para seleccionar y podar, pero no dominancias para podar; en las formulaciones más recientes de programación dinámica y *branch and bound* se usa

tanto la poda por dominancias como por cotas. Así, existen diversos autores que comentan la utilización de procedimientos de *branch and bound* y de programación dinámica de forma simultánea, para, aprovechando las mejores características de ambos, restringir el espacio de soluciones explorado (Morin & Marsten 1976, Marsten & Morin 1978, Ibaraki 1988, Bautista et al. (1992, 1994), Dyer et al. 1995, etc.). De todas maneras, mientras algunos de estos autores no opinan que dicha combinación constituya un nuevo algoritmo de búsqueda híbrido, otros como Morin & Marsten (1976), Marsten & Morin (1978), Bautista et al. (1992, 1994) y Dyer et al. (1995), sí.

A continuación se especifican las exposiciones que se realiza en los tres primeros trabajos, relatando en el apartado 3.7.4. los realizados por Bautista et al. (1992, 1994) y otros muchos autores que han utilizado el procedimiento descrito por éstos: un procedimiento particular al que los mismos autores denominan programación dinámica acotada (*bounded dynamic programming* o BDP).

En problemas de dimensiones industriales, la programación dinámica discreta (cuya característica principal consiste en la exploración del espacio de soluciones factible utilizando el podado por infactibilidades y dominancias) necesita una gran cantidad de memoria y de tiempo de computación para obtener resultados. La memoria, y posiblemente la computación, puede verse grandemente reducida si se utilizan técnicas de ramificación y acotación. En particular, si se puede demostrar que no existe ninguna secuencia de decisiones que aplicadas a un estado x pueden suministrar una solución óptima, no es necesario guardar dicho estado. Estos estados pueden ser identificados utilizando, como ya introducen Morin & Marsten (1976), relajación y criterios de podado usualmente utilizados en *branch and bound* y otros algoritmos enumerativos.

El algoritmo híbrido descrito en primer lugar por Morin & Marsten (1976), se inspira, según estos mismos autores, en los esquemas de acotado utilizados con programación dinámica y propuestos por Proschan & Bray en 1965, Weingartner & Ness en 1967 y otros.

El algoritmo híbrido puede ser visto como una recursión de programación dinámica que utiliza un test de acotado en cada nivel para eliminar estados y reducir el tamaño del espacio de búsqueda. Básicamente el procedimiento funciona de la siguiente manera. Sea un problema de minimización, x un estado factible no dominado, UB una cota superior de la solución óptima, LB(x) una cota inferior del estado x (obtenida al resolver una versión relajada del problema residual en el que x es el estado inicial) y f(x) el coste mínimo de alcanzar el estado x desde el estado inicial. Si:

$$f(x) + LB(x) > UB,$$

entonces no existe ninguna secuencia de decisiones que, aplicadas al estado x, puedan suministrar una política óptima, y el estado x puede ser podado así como todos los que podrían generarse a partir de él; si en vez de buscar todas las políticas óptimas sólo se busca una, se puede podar el estado x si:

$$f(x) + LB(x) \geq UB$$

Por otro lado, si en un nivel se podan todos los estados, la solución que proporciona la cota superior UB es la óptima.

Morin & Marsten (1976) opinan que el algoritmo propuesto incluye la programación dinámica convencional como un caso especial: cuando U es suficientemente grande para no podar ningún estado.

Marsten & Morin (1978) desarrollan una versión más general del procedimiento híbrido que permite obtener algoritmos heurísticos a partir de éste: en un problema de maximización, introducen un parámetro ε que permite dar por finalizado el procedimiento, si la diferencia entre la mejor cota superior y la cota inferior (valor de la solución preferible) es suficientemente pequeña. Así, se selecciona un $\varepsilon \in [0, 1]$ y el criterio de final del procedimiento consiste, para Marsten & Morin (1978), en finalizar la búsqueda si:

$$(UB - LB) / UB \leq \varepsilon,$$

constituyendo la solución preferible la solución buscada, al considerar que ésta está suficientemente cerca del valor de la solución óptima. El parámetro ε determina la aproximación a la optimalidad, correspondiendo $\varepsilon = 0$ a la búsqueda de la solución óptima. Por otro lado, el criterio de finalización también puede ser otro, como por ejemplo finalizar la búsqueda si $UB - LB \leq \varepsilon$.

Morin & Marsten (1976) también proponen una interpretación alternativa del algoritmo híbrido: puede verse como un procedimiento de programación dinámica que utiliza un test de acotado en cada nivel para eliminar estados y reducir el tamaño del espacio de estados; o, alternativamente, puede ser visto como un procedimiento *branch and bound* (que, según Marsten & Morin 1978, utiliza la estrategia *breadth first*) que utiliza las dominancias como un criterio adicional de podado. De todas formas, y como señalan Marsten & Morin (1978), el uso de la estrategia *breadth first search* es un extremo y se podrían utilizar algoritmos *branch and bound* más convencionales con la estrategia híbrida expuesta (*branch and bound* con otras estrategia de búsqueda, pero incorporando la poda por dominancias).

Según Dyer et al. (1995), el algoritmo híbrido depende claramente de un método eficiente para obtener ajustadas cotas LB(x) y UB. Marsten & Morin (1978) también proponen utilizar diversas heurísticas para obtener soluciones factibles, y, por tanto, cotas: la mejor de éstas, pero también más costosa en tiempo, al inicio, y otras más rápidas en los vértices intermedios.

Respecto a las experiencias computacionales realizadas por los diferentes autores, Dyer et al. (1995) exponen las conclusiones siguientes:

- Parece interesante calcular en cada estado su cota inferior y superior, ya que, según estos autores, el problema está en el espacio más que en el tiempo de ejecución (lo que puede indicar una visión poco real de los problemas a resolver, desde el punto de vista industrial). De todas formas, Marsten & Morin (1978) opinan (posiblemente debido a que los ordenadores existentes a finales de los años 70 eran mucho más lentos que los utilizados por Dyer et al.) que se puede reducir la cantidad de tiempo de calculo, si las cotas sólo se determinan de forma intermitente.

- Interesa mantener el número de estados bajo, ya que sino la computación no puede proceder por falta de memoria y además el esfuerzo computacional necesario en los siguientes niveles aumenta innecesariamente.

- El orden en el que se procesan los conjuntos de variables no tiene ningún efecto en la solución óptima de un problema concreto, pero el número de estados generados y la eficiencia del algoritmo, sí que puede depender del orden del proceso de los conjuntos de variables. Para seleccionar la permutación en la que procesar los conjunto de variables, existe una heurística que consiste en procesar primero el conjunto de variables de menor cardinal; esta ordenación previa produce drásticas restricciones del número de estados generados (Dyer et al. 1995). De todas formas, Marsten & Morin (1978) proponen, para un caso de maximización, una ordenación previa de las variables en orden no decreciente de su valor en la función objetivo (el hecho de ordenar con este criterio se basa en seleccionar las variables peores para generar estados malos que pueden ser eliminados rápidamente).

Como conclusión, Dyer et al. (1995) y Morin & Marsten (1976) especifican que el algoritmo híbrido parece muy práctico y proporciona grandes ahorros, tanto en necesidad de almacenamiento como de computación; el único factor límite es la memoria.

3.7.4. Programación dinámica acotada (*bounded dynamic programming* o BDP).

En la línea de los trabajos de Morin & Marsten (1976), Marsten & Morin (1978) e Ibaraki (1988), Bautista et al. (1992, 1994) describen un nuevo procedimiento híbrido, la programación dinámica acotada (*bounded dynamic programming* o BDP), que también aprovecha las mejores características tanto de los procedimientos de *branch and bound* como de la programación dinámica, para reducir el espacio de soluciones factibles generado y explorado.

Como los mismos autores aclaran, la utilización de cotas en un esquema de programación dinámica no es una idea nueva. En todos los casos, el uso de cotas permite eliminar vértices, gracias a lo cual es factible resolver problemas de dimensiones algo mayores; sin embargo, cuando el número de vértices crece exponencialmente con la dimensión de los ejemplares, el aumento de las dimensiones tratables es reducido y, en definitiva, la programación dinámica, con cotas o sin ellas, sólo permite resolver problemas de dimensiones reducidas.

La consideración explícita de la limitación en la disponibilidad de memoria a la hora de almacenar vértices permite resolver problemas de grandes dimensiones, sin renunciar a priori a la posibilidad de obtener el óptimo y de tener la certeza de que ha sido alcanzado. El procedimiento basado en la utilización de cotas en un esquema de programación dinámica (en el que se agrupan los estados equivalentes) y con un tratamiento que tiene en cuenta la limitación en el número de vértices del grafo que se puede almacenar, ha sido denominado por Bautista et al. (1992) programación dinámica acotada (*bounded dynamic programming* o BDP).

Concretamente, Bautista et al. (1994) definen la BDP como un procedimiento de búsqueda en el espacio de estados basado en la utilización de cotas en un esquema de programación dinámica, teniendo en cuenta, en base a la disponibilidad de la capacidad de la memoria, una limitación en el número de vértices del grafo de estados que se almacenan en cada etapa. Para ello se introduce el concepto de ventana de anchura H sobre un grafo de estados polietápico, que, como máximo, en cada nivel guarda para explorar en la siguiente etapa los H mejores vértices. El proceso de selección de vértices en cada nivel retiene aquéllos que presentan los valores (de una cota o de un indicador) más prometedores para alcanzar la solución óptima. La BDP permite ir eliminando vértices poco o nada prometedores al acotar el valor de la función objetivo, además de descartar aquellos vértices que violan las restricciones asociadas al problema.

Supóngase disponer de una solución factible en un problema de minimización obtenida mediante algún procedimiento heurístico, de valor $\overline{Z}$; en toda etapa, cada vértice dispone de un valor de la función objetivo hasta ese nivel, si es posible calcular una cota

inferior del valor de la función objetivo desde ese vértice hasta el final, la suma de ambos valores constituye una cota inferior Z para ese vértice: si $Z \geq \overline{Z}$ se puede podar el árbol y eliminar directamente el vértice. Evidentemente, si se dispone de una buena solución inicial, $\overline{Z}$, y de un procedimiento de acotado muy ajustado, el número de vértices se puede reducir considerablemente.

Como exponen Bautista et al. (1992), a pesar de la poda realizada en el grafo, el número de vértices restantes todavía podría exceder la capacidad de la memoria disponible: cuando en un nivel se sobrepasa el ancho de la ventana, H, se expulsan los vértices de peor Z. Esta eliminación puede impedir que se halle la solución óptima, o también puede permitir obtenerla pero sin disponer de argumentos para garantizar que lo es. De esta manera, si hay expulsiones es conveniente guardar el valor del mejor de los vértices expulsados: si al finalizar el procedimiento no se ha expulsado ningún vértice o la solución obtenida tiene un valor de Z menor o igual que la mejor cota de los vértices expulsados, entonces la solución es óptima; si no, se obtiene una cota de la separación máxima entre el valor de la solución obtenida y el valor óptimo.

A continuación se enumeran brevemente las fases del procedimiento para una más fácil comprensión del mismo:

Fase 1. Cálculo de una solución inicial $\overline{Z}$.
Fase 2. Generación de vértices.
Generación de los vértices descendientes de los del nivel actual.
Fase 3. Evaluación de la cota de la solución en construcción.
Evaluar el valor de la cota.
Si la cota es peor que $\overline{Z}$ eliminar el vértice e ir a fase 2.
Fase 4. Test de cumplimiento de restricciones.
Si se viola alguna restricción ahora o en el futuro, eliminar el vértice e ir a fase 2.
Fase 5. Actualización de la lista de vértices por limitación de memoria.
Si hay memoria disponible, almacenar el vértice generado para considerarlo en la etapa siguiente e ir a 2.
Si no hay memoria suficiente, rechazar el peor vértice entre los generados y los almacenados, actualizar la cota del mejor vértice rechazado e ir a 2.

En Sirer (1992), Bautista (1994) y Pastor (1994), se proponen diversas estrategias y variantes en la aplicación de la BDP:

- Cambiar el orden de alguno de los pasos del procedimiento: se puede demorar el cálculo de la cota hasta el momento en el que se tenga la seguridad que el vértice no ha sido ya generado, esto ahorraría cálculos de cota a costa de hacer más comparaciones.

- Como ya se ha comentado, el número de vértices retenidos en cualquier nivel es función de la bondad de la solución inicial, $\overline{Z}$, y del procedimiento de acotación empleado; en caso de que el procedimiento acabe sin garantizar el óptimo, pero dando una solución mejor que la disponible inicialmente ($Z_j < \overline{Z}$), puede resultar útil reintentarlo tomando como $\overline{Z}$ el valor de la solución obtenida ($\overline{Z} = Z_j$) y hasta que $Z_{j+1} = \overline{Z}$ ó hasta que se garantice la consecución del óptimo. Se obtendrá una secuencia monótona decreciente de Z_j y una secuencia monótona creciente de cotas.

- Si en el caso anterior, además se itera con $H_j > H_{j-1}$, lo más probable, pero no seguro ya que los autores han observado casos en los que no se cumple, es que se obtenga una secuencia monótona decreciente de Z_j y una secuencia monótona creciente de cotas.

- Antes de comenzar se puede intentar mejorar el valor de $\overline{Z}$ utilizando la misma BDP pero con un ancho de ventana H' < H (y en particular con H' = 1); de esta manera, este procedimiento también se puede utilizar para obtener rápidamente soluciones iniciales.

Varios de los autores que han trabajado con la BDP (Bautista et al. 1992, Roca 1992, Bosch 1993, Bautista et al. 1994, Franco & López 1995, Gangonells & Gómez 1995, etc.), comentan que ésta es más una metodología para resolver problemas que no un algoritmo bien determinado, de manera que son muchas y diversas las formas de implementarla; algunas estrategias para aplicar eficientemente la BDP son, según estos mismos autores:

- esmerarse en el cálculo de la solución inicial,
- obtener para cada vértice cotas muy ajustadas,
- realizar iteraciones con una anchura de ventana fija, utilizando en cada iteración como solución inicial la solución obtenida en la iteración anterior,
- iterar de igual manera que en el caso anterior, utilizando en cada iteración la solución obtenida en la iteración anterior como cota superior del óptimo, pero además aumentando en cada iteración el ancho de ventana empleado.

Por otro lado, los elementos fundamentales que intervienen en la eficacia de la BDP y que se han de regular con cautela son, para dichos autores:

- la calidad de la solución inicial,
- la anchura de la ventana (en contra de lo que se puede pensar, el aumento de este valor no implica la obtención de soluciones mejores),
- el proceso de eliminación de vértices:
    a) la calidad del predictor y/o la cota,
    b) la complejidad de cálculo de la cota y/o el predictor,
    c) las dominancias entre vértices.

Cabe destacar que la BDP ha sido utilizada como un potente procedimiento de resolución de diversos problemas de optimización combinatoria: problema del viajante de comercio (Roca 1992), equilibrado de líneas de producción o montaje (Sirer 1992), corte unidimensional de materiales (Bosch 1993), determinación de secuencias regulares en líneas de montaje mixtas con restricciones en la elaboración de productos (Bautista et al. 1994), problema del taller mecánico n/m/P/Fmax (Franco & López 1995) y programación de proyectos con limitación de recursos (Gangonells & Gómez 1995).

3.7.5. Procedimientos *branch and bound* con técnicas de exploración de entornos.

Portmann et al. (1998) resuelven problemas de *flow-shop*, utilizando un procedimiento *branch and bound* al que incorporan una técnica de exploración de entornos, en este caso un algoritmo genético, para intentar mejorar el valor de la solución preferible.

A grandes rasgos, la idea básica que introducen Portmann et al. (1998) consiste en ejecutar un algoritmo genético en algunos vértices del árbol de búsqueda, de forma que éste construye una serie de poblaciones de soluciones completas a partir de la solución parcial resultado del conjunto de decisiones ya tomadas por el *branch and bound*. El inicio de las soluciones completas viene fijado por las decisiones tomadas por el *branch and bound* y sólo las últimas decisiones pueden ser "optimizadas" por el algoritmo genético; además, y debido al excesivo tiempo que consume el algoritmo genético, éste no es ejecutado en todos los vértices generados. El objetivo final consiste en mejorar la solución preferible.

Como comentan estos autores, la eficiencia del algoritmo resultante depende del tiempo de búsqueda asignado a la técnica de exploración de entornos y de los vértices en los que se ejecute; estos son los principales parámetros de ajuste del procedimiento expuesto.

3.8. Procedimientos de búsqueda basados en técnicas de reducción.

En diversas ocasiones a lo largo del texto se ha hecho referencia a la reformulación de los problemas de optimización combinatoria, como una herramienta importante en los procedimientos utilizados para su resolución. Como exponen Hoffman & Padberg (1996), existen diferentes maneras de representar un mismo problema, y obtener la solución óptima en un tiempo razonable puede depender enormemente de la formulación empleada. Así no es de extrañar que autores como Johnson et al. (1997) propongan algunos principios generales para obtener buenas formulaciones en programación lineal mixta:

- Disgregación de restricciones: trabajando con variables binarias y para representar las condiciones $x_0 = 0 \Rightarrow x_j = 0, j = 1, ..., m$, es más conveniente utilizar m restricciones del tipo $x_j \leq x_0 \; \forall \; j$, más que la restricción $x_1 + ... + x_m \leq m \cdot x_0$.

- Reducción de coeficientes: sea x_{jk} la cantidad de producto j fabricado en la factoría k y sea $y_k = 0$ (ó 1) si la fábrica está cerrada (o abierta); se plantea la restricción $x_{jk} \leq M \cdot y_k$, donde M es cualquier cota superior de x_{jk}; para obtener una buena formulación, M debe ser lo más pequeña posible.

Con el objetivo de mejorar o simplificar la formulación y reducir el espacio de búsqueda, la mayoría de autores, entre ellos Corominas et al. (1984), Ibaraki (1988), Hoffman & Padberg (1993), Savelsbergh (1994), Escudero (1995a, 1995b), Fischetti et al. (1995), Balas et al. (1996) y Johnson et al. (1997), comentan que antes de aplicar un procedimiento de resolución se puede ajustar la formulación con técnicas automáticas de preproceso. Como exponen Balas et al. (1996), estas consideraciones están teniendo un considerable impacto y en trabajos recientes se está investigando intensamente en este tema; según Hoffman & Padberg (1993), el preproceso es una forma efectiva y no cara de alcanzar estos objetivos.

Por su parte, autores provenientes del área de la inteligencia artificial proponen algunos procedimientos, principalmente para resolver problemas denominados de satisfacción de restricciones, en los que estas técnicas adquieren todo el protagonismo, de forma que se convierten en el arma principal para generar y explorar el menor número de estados posibles del árbol de búsqueda y resolver el problema combinatorio propuesto.

Muchas de estas técnicas son de uso general, pero como señalan Ibaraki (1988) y Fischetti et al. (1995) entre otros, también se pueden diseñar otras que exploten las características propias del problema concreto a resolver.

De esta manera, se puede considerar que existen unos procedimientos o técnicas enumerativas particulares, desarrolladas en el área de la inteligencia artificial para resolver el problema de satisfacción de restricciones, frente a los procedimientos de optimización combinatoria expuestos hasta el momento. A continuación se enumeran las diferencias principales que se presentan entre estos dos grupos de técnicas:

1) Las técnicas enumerativas utilizadas en la resolución del problema de satisfacción de restricciones principalmente intentan resolver problemas de existencia (obtención de soluciones factibles), mientras que en los procedimientos expuestos hasta el momento el objetivo principal es el de optimizar; de todas formas, dichos procedimientos pueden ser adaptados para trabajar con el objetivo de optimizar (Corominas 1996).

2) Habitualmente, las técnicas de preproceso no se utilizan como herramientas exclusivas de resolución, aunque, ocasionalmente, un problema de dimensiones reducidas puede ser resuelto en la fase de preproceso: la resolución del problema no es su misión principal, es la de preparar de forma automática una formulación rápida para la aplicación de otros algoritmos (Nemhauser & Wolsey 1988). Todo lo contrario que ocurre en estos procedimientos procedentes del área de la inteligencia artificial, que se sustentan, de manera casi exclusiva, en el tratamiento sistemático de las restricciones (Corominas 1996).

3) En los procedimientos de optimización expuestos hasta el momento, las técnicas de preproceso se pueden considerar como una fase entre la formulación y la solución, de forma que habitualmente sólo se utilizan antes de comenzar con el procedimiento de resolución (aunque no en todos los casos: algoritmos de Balas, algoritmo propuesto por Dyer et al. 1995, etc.). En las técnicas utilizadas para la resolución del problema de satisfacción de restricciones, se utilizan métodos de propagación y satisfacción de restricciones en todos y cada uno de los subproblemas generados.

Las definiciones y comentarios que se realizan en el área de la investigación operativa con relación a las técnicas de preproceso y en el área de la inteligencia artificial respecto a las técnicas utilizadas en la resolución del problema de satisfacción de restricciones, permiten hablar de ambos conjuntos de técnicas como de uno solo, ya que algunas en el fondo son muy semejantes, y, sobretodo, persiguen los mismos objetivos: son las técnicas de reducción.

En el apartado siguiente se exponen diferentes técnicas de reducción procedentes del área de la investigación operativa (técnicas de preproceso) y qué entienden por éstas los autores que las utilizan (apartado 3.8.1.); y a continuación se introduce en qué consiste el problema de la satisfacción de restricciones, algunas de las técnicas de reducción que se utilizan en su resolución y algunos procedimientos enumerativos de búsqueda que las incorporan para resolver este problema (apartado 3.8.2.). De esta manera, se pretende realizar una breve exposición que permita argumentar, en apartados posteriores, la consideración de que ambos conjuntos de técnicas pueden ser utilizados en un mismo esquema de resolución, a la vez que los procedimientos enumerativos de resolución de problemas de satisfacción de restricciones se pueden enmarcar en una formulación general de *branch and bound.*

3.8.1. Técnicas de reducción procedentes del área de la investigación operativa.

Dada una formulación de un problema de optimización combinatoria, el preproceso se refiere a las operaciones elementales que se pueden utilizar para mejorar o simplificar la formulación; de esta manera, se transforma el problema en una representación

equivalente que ajusta la solución, al transformar, añadir, fijar y/o eliminar variables, y al detectar inconsistencias entre las restricciones (redundancias o dominancias que permiten eliminarlas, infactibilidades, etc.), e incluso al añadir nuevas restricciones (Corominas et al. 1984, Ibaraki 1988, Hoffman & Padberg 1993, Savelsbergh 1994, Escudero et al. (1995a, 1995b), Fischetti et al. 1995, Johnson et al. 1997, etc.).

Escudero et al. (1995a) y Kubiak et al. (1997) destacan la importancia de obtener propiedades (de obligado cumplimiento por toda solución óptima) que permitan alcanzar el objetivo que se persigue con estas técnicas: la simplificación de la búsqueda. Por otro lado, es necesario aclarar que estas técnicas no son propias de un procedimiento, se utilizan en *branch and bound*, *branch and cut* (Hoffman & Padberg 1993 y Fischetti et al. 1995), etc. Además, como ya se ha comentado y exponen Savelsbergh (1994) y Corominas (1996), estas técnicas pueden llegar a ser muy efectivas, y lo son en mayor grado cuanto más restrictivo es el problema a resolver y si se utilizan en combinación con otras técnicas: en la resolución de problemas de programación lineal mixta utilizando la programación lineal como procedimiento de acotado, los programas matemáticos son reformulados para que la diferencia entre el valor de la solución del programa lineal y el valor de la solución del programa entero sea lo menor posible, de esta manera se potencia el procedimiento de acotado (Savelsbergh 1994).

Según exponen Hoffman & Padberg (1993), estas técnicas fijan variables, eliminan columnas y buscan inconsistencias entre las restricciones (dominancias, infactibilidades, etc.), pero, ¿cómo lo hacen?

Por un lado, existen técnicas de reducción que son propias y particulares del problema a resolver. Por ejemplo, en los problemas de equilibrado de líneas de montaje se pueden aprovechar las relaciones de precedencia entre tareas, que originan la imposibilidad de asignación de todas las tareas a todas las estaciones de trabajo (la primera tarea difícilmente podrá asignarse a la última estación), para obtener una amplia reducción en las posibles asignaciones de las tareas a las estaciones; básicamente, el preproceso consiste en propagar las limitaciones que imponen las relaciones de precedencia entre tareas, por las que antes de que se asigne una tarea se debe asignar el tiempo total de las tareas que la preceden y después el tiempo total de las tareas que la suceden; como resultado se obtiene un rango reducido de estaciones a las que se puede asignar cada tarea. En el problema de cubrimiento, Ibaraki (1988) propone la reducción de los problemas parciales: eliminar las variables ya fijadas y las restricciones de cubrimiento de zonas ya cubiertas, fijar el valor de una variable si una zona sólo puede ser cubierta esa variable, solución infactible si existe una zona no cubierta que no puede ser cubierta (tal que los coeficientes a_{ij} de las variables que quedan libres son nulos), etc.

Y por otro lado, existen técnicas de reducción que son mucho más generales y ampliamente utilizables. El algoritmo de Balas (introducido en el apartado 3.5.1.1.) es uno de los primeros procedimientos en los que utilizan las técnicas de preproceso de forma sistemática. Como exponen Corominas et al. (1984), este procedimiento de resolución de programas lineales binarios puros realiza una búsqueda en profundidad con *backtracking*, seleccionando para explorar la variable que presenta un menor índice de imposibilidad de obtener una solución factible; además, en cada vértice de la arborescencia se realizan tests de eliminación de variables, con los que se reduce el número y complejidad de las restricciones, además de fijar el valor de alguna variable.

A continuación se exponen los tests que se utilizan en el algoritmo de Balas según la terminología utilizada por Corominas et al. (1984), tests que también se comentan en Garfinkel & Nemhauser (1972) y Nemhauser & Wolsey (1988), así como algunas extensiones realizadas por este mismo autor en Corominas (1996). Sea F el conjunto de variables todavía no fijadas (variables libres), S^+ el conjunto de variables con valor igual a 1, S^- el conjunto de variables con valor igual a 0, N^+ el conjunto de coeficientes positivos de variables de F, N^- el conjunto de coeficientes negativos de variables de F, y se desea optimizar el siguiente programa matemático:

$$\begin{array}{lll} [\text{MAX}]\ z = c \cdot x & & [\text{MAX}]\ z = \sum_{1\le j\le n} c_j \cdot x_j \\ A \cdot x \le b & \text{es decir} & \sum_{1\le j\le n} a_{ij} \cdot x_j \le b_i \quad i=1,\ldots, m \\ x_j \in \{0, 1\} & & x_j \in \{0, 1\} \quad j=1, \ldots, n \end{array}$$

donde se asume sin pérdida de generalidad que $c_j < 0\ \forall\ j$, ya que siempre puede sustituirse una variable por su complementaria: $x_j' = 1 - x_j$.

* Test 1:

Para cada restricción se calculan los siguientes valores:

$$s_i = b_i - \sum_{j\in S+} a_{ij}$$
$$t_i = \sum_{j\in F} \min(0, a_{ij})$$

Entonces, si existe una variable x_j tal que $|a_{ij}| > s_i - t_i$, se puede presentar una de las dos opciones siguientes:

- si $a_{ij} > 0$ entonces $x_j = 0$,
- si $a_{ij} < 0$ entonces $x_j = 1$.

* Test 1 generalizado:

Se disponen las restricciones como se muestra a continuación:

$\sum_{j \in N+} a_j \cdot x_j - \sum_{j \in N-} a_j \cdot x_j \leq s$
$a_j > 0, \forall j \in N^+ \cup N^-$

Para cada restricción se definen los siguientes valores:

$r = \sum_{j \in N+} a_j$, valor máximo que puede alcanzar la parte izquierda de la restricción,
$t = - \sum_{j \in N-} a_j$, valor mínimo que puede alcanzar la parte izquierda de la restricción,
$s - t$, margen para satisfacer la restricción.

Y entonces se pueden obtener diversas conclusiones:

0) $r \leq s \rightarrow$ la restricción se cumple siempre y se puede eliminar.
1) $t > s \rightarrow$ la restricción no se puede cumplir nunca y no existe ninguna solución factible.
2) $a_j > s - t$ y $j \in N^+ \rightarrow x_j = 0$.
3) $a_j > s - t$ y $j \in N^- \rightarrow x_j = 1$.

Las relaciones anteriores se pueden extender a parejas de variables obteniendo nuevas conclusiones:

4) $a_i + a_j > s - t$ y $i, j \in N^+ \rightarrow x_i + x_j \leq 1$.
5) $a_i + a_j > s - t$ y $i, j \in N^- \rightarrow x_i + x_j \geq 1$.
6) $a_i + a_j > s - t$ y $i \in N^+, j \in N^- \rightarrow x_i \leq x_j$.

Y también se podrían extender a ternas de variables, etc.

Del tratamiento de cada restricción, y considerando las conclusiones anteriores, se pueden obtener los resultados siguientes:

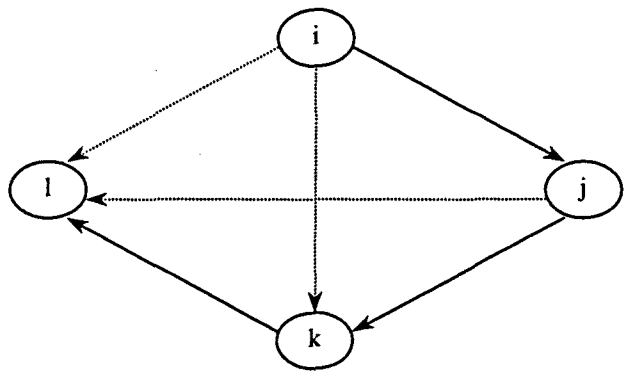
(a) $x_j = 0$.
(b) $x_j = 1$.
(c) $x_i + x_j \leq 1$.
(d) $x_i + x_j \geq 1$.
(e) $x_i \leq x_j$.
(e´) $x_j \leq x_i$.

Considerando que estas propiedades se pueden aplicar a todas las restricciones, se obtienen conjuntos de resultados que pueden ser analizados para extraer nuevas conclusiones:

(a) y (b) $\rightarrow$ el programa no es factible.
(a) y (d) $\rightarrow x_i = 1$.
(a) y (e) $\rightarrow x_i = 0$.
(b) y (c) $\rightarrow x_i = 0$.
(b) y (e') $\rightarrow x_i = 1$.
(c) y (d) $\rightarrow x_i + x_j = 1$; $x_i = 1 - x_j$ y se puede eliminar una variable.
(c) y (e) $\rightarrow x_i = 0$.
(d) y (e') $\rightarrow x_i = 1$.
(e) y (e') $\rightarrow x_i = x_j$ y se puede eliminar una variable.

Además, también se puede concluir lo siguiente:

7) $x_i \leq x_j$ y $x_j + x_k \leq 1 \rightarrow x_i + x_k \leq 1$.
8) $x_j \leq x_i$ y $x_j + x_k \geq 1 \rightarrow x_i + x_k \geq 1$.
9) La propiedad (e) es transitiva: $x_i \leq x_j$, $x_j \leq x_k \rightarrow x_i \leq x_k$. Una forma de proceder de manera práctica, consiste en considerar un grafo en el que las variables se asocian a los vértices y donde existe un arco ij si $x_i \leq x_j$:



Grafo de transitividad de valores, de Corominas (1996).

De esta manera, los caminos en este grafo proporcionan nuevas propiedades de este tipo; además, si aparece un circuito todas las variables tienen el mismo valor.

* Test 2:

En el vértice k de la arborescencia que está siendo procesado se genera una restricción, llamada restricción 0, de la siguiente forma:

$\sum_{j \in Fk} a_{0j} \cdot x_j \leq s_0$,

donde:

$a_{0j} = - c_j$,

$s_0 = Z_k'$ (cota superior del vértice) - $\bar{Z}$ (valor de la solución preferible).

Para cada restricción i se realiza el siguiente proceso:

1°. Si existe una variable $x_h \in F_k$ tal que $a_{ih} > s_i$, calcular: $a_0^* = \min [a_{0j} \mid j \in F_k - \{x_h\}, a_{ij} < 0]$.

2°. Si $a_{0h} + a_0^* > s_0$, entonces $x_h = 0$.

Como se expone en Corominas et al. (1984), la interpretación es la siguiente. Si se hiciera $x_h = 1$, cumpliéndose la condición $a_{ih} > s_i$, ello significaría que en el siguiente vértice el término independiente de la restricción i-ésima sería negativo, pues valdría $-a_{ih} + s_i < 0$. Para que el vértice fuera factible, posteriormente habría que hacer igual a 1 por lo menos otra variable con coeficiente negativo ($a_{ij} < 0$), lo que provocaría una disminución en la función objetivo que, en el mejor de los casos, sería la actual, a_{0h}, más el mínimo posible correspondiente a la otra variable que se ha hecho 1, a_0^*. Si la suma de ambos valores fuera superior a la holgura del vértice ($s_0 = Z_k' - \bar{Z}$), $a_{0h} + a_0^* > s_0$, significaría que la solución sería peor que la preferible. La única forma de evitarlo es imponiendo $x_h = 0$.

* Test 3:

Se disponen las restricciones de la siguiente manera:

$\sum_{j \in F} a_j \cdot x_j \leq b$

$a_j > 0, \forall j \in F$

Si existe $a_j < 0$, se puede remplazar x_j por $1 - x'_j$ y obtener la restricción:

$\sum_{j \in N+} a_j \cdot x_j + \sum_{j \in N-} |a_j| \cdot x'_j \leq b - \sum_{j \in N-} a_j$,

de esta manera, y sin pérdida de generalidad, se puede asumir que $a_j > 0$ para $j \in F$.

Entonces, si existe un conjunto de variables $C \subseteq F$ tal que $\sum_{j \in C} a_j > b$, se puede plantear la restricción:

$$\sum_{j \in C} x_j \leq |C| - 1$$

Una vez se han obtenido restricciones de este tipo, es posible combinar varias de ellas de forma semejante a como se realiza en el Test 1 generalizado, para fijar variables.

La aplicación de estas ideas es fácil de asimilar con un ejemplo sencillo (Nemhauser & Wolsey 1988):

$$\begin{array}{rcr} -3x_2 - 2x_3 \leq -2 & & 3x'_2 + 2x'_3 \leq 3 \\ -4x_1 - 3x_2 - 3x_3 \leq -6 & \text{se transforma en} & 4x'_1 + 3x'_2 + 3x'_3 \leq 4 \\ 2x_1 - 2x_2 + 6x_3 \leq 5 & & 2x_1 + 2x'_2 + 6x_3 \leq 7 \\ x_j \in \{0, 1\} & & x_j \in \{0, 1\} \end{array}$$

La primera restricción proporciona $x'_2 + x'_3 \leq 1$ ó, después de la transformación, $x_2 + x_3 \geq 1$; la tercera restricción puede proporcionar $x'_2 + x_3 \leq 1$ ó $x_3 \leq x_2$; y combinando ambas se obtiene que $x_2 = 1$. En este momento la primera restricción es redundante ($-2x_3 \leq 1$), y la segunda y la tercera se reducen respectivamente a $4x'_1 + 3x'_3 \leq 4$ y $2x_1 + 6x_3 \leq 7$. De estas dos restricciones se obtiene $x'_1 + x'_3 \leq 1$ ó $x_1 + x_3 \geq 1$ y $x_1 + x_3 \leq 1$, por lo que se deduce que $x_1 + x_3 = 1$; por sustitución se puede eliminar una de las dos variables ($x_3 = 1 - x_1$) y se obtiene que $-x_1 \leq -3$ y $x_1 \leq 1$, lo que muestra que el programa es infactible y el vértice puede ser podado.

Para la resolución de problemas de programación entera, Ibaraki (1988) expone, de manera concreta, un conjunto de sencillos tests que permiten reformular estos programas matemáticos. Sea el siguiente programa lineal entero:

$$\begin{aligned} [\text{MAX}]\ z &= \textstyle\sum_{1 \leq j \leq n} c_j \cdot x_j \\ \textstyle\sum_{1 \leq j \leq n} a_{ij} \cdot x_j &\leq b_i \\ x_j &\text{ entera.} \end{aligned}$$

* Una columna nula, es decir $c_j = a_{ij} = 0\ \forall i$, permite eliminar la variable x_j.

* Una fila nula, es decir $a_{ij} = 0\ \forall j$, implica una restricción redundante o un programa matemático infactible, dependiendo si $b_i \geq 0$ ó $b_i < 0$.

* Filas sencillas, es decir todas las $a_{ij} = 0$ excepto una, implica que la restricción se convierte en una cota superior o inferior de la variable x_j; esta nueva cota de x_j puede aplicarse a las otras restricciones y ver si son infactibles, redundantes, si restringe más la variable, etc.

* Columnas sencillas, es decir sólo uno de los valores c_j ó a_{ij} $\forall i$ es $\neq 0$: si $c_j > 0$ y $a_{ij} = 0$, entonces x_j adopta su cota superior; si $c_j = 0$, $a_{ij} < 0$ y la restricción es $\leq$, entonces x_j adopta su cota superior. De esta manera se trabaja con una variable menos, x_j.

* Filas con todas las a_{ij} del mismo signo: la restricción puede ser infactible (pe. $x_1 + 2x_2 \leq -1$), redundante (pe. $x_1 + 2x_2 \geq 0$) ó se puede fijar el valor de alguna variable (pe. $x_1 + 2x_2 = 0 \Rightarrow x_1 = x_2 = 0$).

* Si se dispone de columnas con $c_j \geq 0$ y $a_{ij} \leq 0$ $\forall i$, y restricciones $\leq b_i$, entonces la variable x_j adopta su cota superior.

* Si se dispone de columnas con $c_j \leq 0$ y $a_{ij} \geq 0$ $\forall i$, y restricciones $\leq b_i$, entonces la variable x_j adopta su cota inferior.

* Reducciones de equivalencia: si se presenta una restricción de igualdad, una variable se puede poner en función de las demás y ser sustituida en las restricciones y en la función objetivo.

De una manera menos precisa, Ibaraki (1988) también comenta otras técnicas de reducción:

- Reducción de actividad en las filas y definición de nuevas cotas para las variables.

- Reducción de coeficientes de las restricciones: sean x_1 y $x_2 \in \{0, 1\}$, $3x_1 + 2x_2 \leq 4 \Rightarrow x_1 + 2x_2 \leq 2 \Rightarrow x_1 + x_2 \leq 1$.

- Reformulación por adición de restricciones: inclusión de planos de corte de la resolución del programa lineal utilizado como relajación del programa entero, para la obtención de cotas más ajustadas.

El uso de técnicas de preproceso y de prueba en la resolución de problemas de programación entera mixta, es expuesto de forma muy completa en Savelsbergh (1994). Este autor presenta técnicas para mejorar la representación de un problema, de forma que es reformulado para que la diferencia entre el valor de la solución del programa lineal utilizado en el procedimiento de acotado y el valor de la solución del programa entero mixto (el *gap* de integridad) sea lo menor posible. Como expone, con el objetivo de reducir el espacio de soluciones factibles del programa lineal sin afectar al espacio de soluciones factibles del programa entero mixto, se puede ajustar la formulación del programa lineal mediante preproceso y pruebas (que identifican infactibilidades y

redundancias, mejoran las cotas y los coeficientes, y fijan variables), y utilizando técnicas de generación de restricciones (que generan restricciones válidas fuertes).

Concretamente, Savelsbergh (1994) presenta un *survey* de varias técnicas básicas de preproceso y de prueba muy conocidas. Sea el siguiente programa lineal entero mixto:

$$[\text{MIN}]\ z = c \cdot x + h \cdot y$$
$$A \cdot x + G \cdot y \leq b$$
$$x \in \{0, 1\}^n$$
$$y \in R^m$$

La idea básica de la utilización de estas técnicas consiste en analizar sucesivamente cada una de las restricciones del sistema de inecuaciones que definen la región de soluciones factibles, intentando establecer si dicha restricción:

- fuerza a que la región de soluciones factibles esté vacía,
- es redundante,
- puede ser utilizada para mejorar las cotas de las variables,
- puede ser ajustada modificando los coeficientes de las variables,
- o fuerza a que alguna variable binaria tome el valor 1 ó 0.

Sea la restricción general:

$$\sum_{j \in B+} a_j \cdot x_j - \sum_{j \in B-} a_j \cdot x_j + \sum_{j \in C+} g_j \cdot y_j - \sum_{j \in C-} g_j \cdot y_j \leq b,$$

donde:

B^+ y B^-, son el conjunto de variables binarias con coeficientes $a_j > 0$ y $a_j < 0$, respectivamente.

$B = B^+ \cup B^-$, es el conjunto de variables binarias.

C^+ y C^-, son el conjunto de variables enteras y reales (sin distinción) con coeficientes $g_j > 0$ y $g_j < 0$, respectivamente.

$C = C^+ \cup C^-$, es el conjunto de variables enteras y reales.

$a_j > 0$ para $j \in B$ y $g_j > 0$ para $j \in C$.

l_j y u_j son las cotas inferiores y superiores de las variables enteras y reales: $l_j \leq y_j \leq u_j$.

$A \cdot x + G \cdot y \leq b$, $x \in \{0, 1\}^n$, $y \in \Re^m$ representa el programa matemático.

$a^i \cdot x + g^i \cdot y \leq b_i$ representa una restricción del sistema.

$A^i \cdot x + G^i \cdot y \leq b^i$ representa el sistema obtenido al eliminar la restricción $a^i \cdot x + g^i \cdot y \leq b_i$ de $A \cdot x + G \cdot y \leq b$.

Savelsbergh (1994) expone las siguientes técnicas básicas de preproceso:

* Identificación de infactibilidades: el problema es infactible si

$$- \sum_{j \in B^-} a^i_j + \sum_{j \in C^+} g^i_j \cdot l_j - \sum_{j \in C^-} g^i_j \cdot u_j > b_i$$

* Identificación de redundancias: la restricción es redundante si

$$\sum_{j \in B^+} a^i_j + \sum_{j \in C^+} g^i_j \cdot u_j - \sum_{j \in C^-} g^i_j \cdot l_j \leq b_i$$

* Mejora del valor de las cotas de las variables no binarias:

Sea y_k, $k \in C^+$; el valor de u_k puede ser mejorado hasta un nuevo valor, si este valor es menor que el u_k actual:

$$(b_i + \sum_{j \in B^-} a^i_j - \sum_{j \in C^+ \setminus \{k\}} g^i_j \cdot l_j + \sum_{j \in C^-} g^i_j \cdot u_j) / g^i_k < u_k$$

Sea y_k, $k \in C^-$; el valor de l_k puede ser mejorado hasta un nuevo valor, si este valor es mayor que el l_k actual:

$$(- \sum_{j \in B^-} a^i_j + \sum_{j \in C^+} g^i_j \cdot l_j - \sum_{j \in C^- \setminus \{k\}} g^i_j \cdot u_j - b_i) / g^i_k > l_k$$

Además, este mismo autor muestra las siguientes técnicas básicas de prueba, que se basan en analizar las consecuencias lógicas derivadas de fijar de manera tentativa una variable binaria x_k a 0 ó a 1 y en aplicar las técnicas de preproceso descritas anteriormente:

* Fijar variables:

Sea x_k, $k \in B^+$; x_k se puede fijar a 0 si

$$a^i_k - \sum_{j \in B^-} a^i_j + \sum_{j \in C^+} g^i_j \cdot l_j - \sum_{j \in C^-} g^i_j \cdot u_j > b_i$$

Sea x_k, $k \in B^-$; x_k se puede fijar a 1 si

$$- \sum_{j \in B^- \setminus \{k\}} a^i_j + \sum_{j \in C^+} g^i_j \cdot l_j - \sum_{j \in C^-} g^i_j \cdot u_j > b_i$$

* Mejora de coeficientes de las variables:

Sea x_k, $k \in B^+$; si

$$\sum_{j \in B+ \setminus \{k\}} a^i_j + \sum_{j \in C+} g^i_j \cdot u_j - \sum_{j \in C-} g^i_j \cdot l_j < b_i,$$

se puede disminuir el valor de a^i_k y de b_i en la cantidad siguiente:

$$b_i - (\sum_{j \in B+ \setminus \{k\}} a^i_j + \sum_{j \in C+} g^i_j \cdot u_j - \sum_{j \in C-} g^i_j \cdot l_j)$$

Sea x_k, $k \in B^-$; si

$$\sum_{j \in B+} a^i_j - a^i_k + \sum_{j \in C+} g^i_j \cdot u_j - \sum_{j \in C-} g^i_j \cdot l_j < b_i,$$

se puede disminuir el valor de a^i_k en la siguiente cantidad:

$$b_i - (\sum_{j \in B+} a^i_j - a^i_k + \sum_{j \in C+} g^i_j \cdot u_j - \sum_{j \in C-} g^i_j \cdot l_j)$$

Las implicaciones lógicas también pueden ser utilizadas para derivar restricciones "clique" de la forma $\sum_{j \in S+} x_j - \sum_{j \in S-} x_j \leq 1 - |S^-|$ (Savelsbergh 1994). Su construcción se basa en las implicaciones lógicas entre variables binarias, implicaciones que pueden ser de cuatro tipos:

$$x_i = 1 \Rightarrow x_j = 0$$
$$x_i = 1 \Rightarrow \bar{x}_j = 0$$
$$\bar{x}_i = 1 \Rightarrow x_j = 0$$
$$\bar{x}_i = 1 \Rightarrow \bar{x}_j = 0,$$

donde $\bar{x}_k = 1 - x_k$, es el complemento de x_k. De esta manera, una implicación lógica identifica a dos variables (originales o complementarias) que no pueden ser iguales a 1 al mismo tiempo en ninguna solución factible. Se puede construir el grafo $G = (B^0 \cup B^C, E)$, siendo B^0 el conjunto de variables binarias originales, B^C el conjunto de variables binarias complementarias, y E el conjunto de arcos, donde dos variables están unidas por un arco sólo si no pueden ser iguales a 1 al mismo tiempo en ninguna solución factible; consecuentemente, cada implicación define un arco en el grafo. Además existe un arco entre cada variable y su complementaria.

Como expone Savelsbergh (1994), no es difícil ver que toda "clique" maximal $C = C^0 \cup C^C$, con $C^0 \subseteq B^0$ y $C^C \subseteq B^C$, define una restricción "clique" válida de la forma:

$$\sum_{j \in Co} x_j + \sum_{j \in Cc} \bar{x}_j \leq 1,$$

que implica:

$$\sum_{j \in Co} x_j - \sum_{j \in Cc} x_j \leq 1 - |C^C|$$

Analizando este tipo de restricciones "clique", se pueden hacer dos observaciones:

a) Si $|C^0 \cap C^C| = 1$ y $k \in C^0 \cap C^C$, entonces $x_j = 0$ para todo $j \in C^0 - \{k\}$ y $x_j = 1$ para todo $j \in C^C - \{k\}$.

b) Si $|C^0 \cap C^C| > 1$ el problema es infactible.

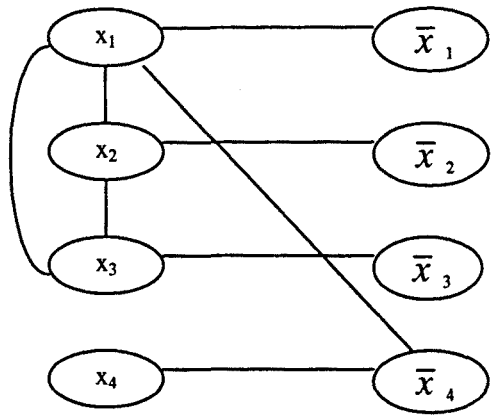
La aplicación de estas ideas es más fácil de asimilar con un ejemplo sencillo (Savelsbergh 1994). Sea el siguiente sistema de restricciones:

$$\begin{gathered} 4x_1 + x_2 - 3x_4 \leq 2 \\ 2x_1 + 3x_2 + 3x_4 \leq 7 \\ x_1 + 4x_2 + 2x_3 \leq 5 \\ 3x_1 + x_2 + 5x_3 \leq 6 \\ x_1, x_2, x_3, x_4 \in \{0, 1\} \end{gathered}$$

Se pueden identificar las siguientes implicaciones lógicas:

$x_1 = 1 \Rightarrow$	$x_4 = 1$	$(x_1 = 1 \Rightarrow x_4 = 1)$
	$x_2 = 0$	$(x_1 = 1 \Rightarrow x_4 = 1 \Rightarrow x_2 = 0)$
	$x_3 = 0$	$(x_1 = 1 \Rightarrow x_3 = 0)$
$x_2 = 1 \Rightarrow$	$x_3 = 0$	$(x_2 = 1 \Rightarrow x_3 = 0)$
$x_3 = 1 \Rightarrow$	$x_2 = 0$	$(x_3 = 1 \Rightarrow x_2 = 0)$
	$x_1 = 0$	$(x_3 = 1 \Rightarrow x_1 = 0)$
$x_4 = 0 \Rightarrow$	$x_1 = 0$	$(x_4 = 0 \Rightarrow x_1 = 0)$

Y también se puede construir el grafo $G = (B^0 \cup B^C, E)$ para el sistema de restricciones anterior:



Grafo auxiliar $G = (B^0 \cup B^C, E)$, de Savelsbergh (1994).

Como se puede comprobar, el grafo G contiene un único "clique" maximal de cardinal más grande que dos: $\{x_1, x_2, x_3\}$. Este "clique" define la restricción $x_1 + x_2 + x_3 \leq 1$.

Por otro lado, Savelsbergh (1994) comenta que existen autores que han estudiado subestructuras especiales de programas matemáticos: Dietrich & Escudero[12] consideran la reducción de coeficientes para problemas de programación lineal 0-1, con restricciones de cotas superiores de las variables; y Hoffman & Padberg[13] discuten la implementación de la reducción de coeficientes para problemas de programación lineal 0-1, con restricciones SOS (*special ordered set*).

Johnson et al. (1997) exponen, por su parte, técnicas de reducción para utilizar en la reformulación y compactación de programas matemáticos generales:

* El test lógico más simple se basa en las cotas del valor de la parte izquierda de las restricciones. Sean m_i y M_i una cota inferior y superior, respectivamente, de la fila $A_i \cdot x$ sólo sujeta a $l \leq x \leq u$: una restricción de la forma $A_i \cdot x \leq b_i$ es redundante si $M_i \leq b_i$, y es infactible si $m_i > b_i$.

* La cota del valor de una variable puede ser ajustada, reconociendo cuándo la restricción pasará a ser infactible si el valor de la variable se mueve hacia la cota.

* Cuando se imponen restricciones con variables binarias, sus cotas inferiores y superiores iniciales son 1 y 0; si estas cotas se mejoran, la variable puede ser fijada: si la cota superior es menor que 1 entonces puede fijarse a 0, si la cota inferior es mayor que 0 puede fijarse a 1, y si la superior es menor que 1 y la inferior mayor que 0, entonces el problema es infactible.

[12] Para un mayor detalle se recomienda consultar Dietrich & Escudero (1990).

[13] Para una información más amplia se recomienda consultar Hoffman & Padberg (1991).

* Los resultados obtenidos de los tests lógicos pueden mejorarse si se combinan con pruebas temporales. Sea una variable binaria 0-1, si se fija temporalmente a 0 ó a 1 y se rehacen los tests lógicos, puede presentarse alguna de las situaciones siguiente: si el problema es infactible la variable puede ser fijada a su otro valor; si se fija el valor de otra variable, se puede derivar una restricción con dos variables (por ejemplo, si x_2 se debe fijar a 0 si x_1 se fija a 1, se puede imponer la restricción $x_1 + x_2 \leq 1$).

* Si las prueban temporales conducen a una restricción redundante, se puede ajustar con el llamado coeficiente de reducción o de mejora: por ejemplo, la restricción $2 \cdot x_1 + x_2 + x_3 \geq 1$ será estrictamente redundante si se fija la variable x_1 a 1. Cuando una variable binaria toma el valor 1 y conduce a una restricción $\geq$ redundante, su el coeficiente puede ser reducido por la cantidad por la que la restricción llega a ser redundante. Además, también se pueden mejorar los coeficientes de las variables durante esta etapa de pruebas.

* Una restricción de la forma $\sum_{j \in S} x_j \leq 1$ se llama "clique". Estas restricciones pueden ser derivadas de implicaciones de grado dos: por ejemplo, $x_1 + x_2 \leq 1$, $x_2 + x_3 \leq 1$ y $x_1 + x_3 \leq 1$, implican la restricción más fuerte $x_1 + x_2 + x_3 \leq 1$; de todas formas, existe una forma más general que incluye restricciones de grado dos de la forma $x_1 + x_2 \leq 1$, $x_1 \leq x_2$ y $x_1 + x_2 \geq 1$. En general, una "clique" puede ser puesta de la forma: $\sum_{j \in S^+} x_j - \sum_{j \in S^-} x_j \leq 1 - |S^-|$. Esas restricciones tienen las siguientes propiedades:

- para $j \in S^+$: si x_j se fija a 1 entonces todas las otras variables x_j pueden ser fijadas a 0 para $j \in S^+$ y a 1 para $j \in S^-$;
- para $j \in S^-$: si x_j se fija a 0 entonces todas las otras variables x_j pueden ser fijadas como en el caso anterior.

* Otra clase de restricciones basadas en implicaciones que pueden ser encontradas por pruebas, son de la forma $y_j \leq u_j \cdot x_k$; esta clase de restricción es válida cuando $x_k = 0$ fuerza a la variable continua $y_j \leq u_j$ a ser 0. Según Nemhauser & Wolsey (1988) y Johnson et al. (1997), se debe intentar obtener el menor valor posible de u_j para ajustar al máximo el valor de la variable y_j. Por otro lado, se pueden derivar restricciones similares cuando probando una variable binaria x_k, se fuerza a una variable a tomar el valor de su cota inferior o superior.

3.8.2. El problema de satisfacción de restricciones: técnicas de reducción y procedimientos enumerativos para su resolución.

Como ya se ha comentado, el problema de satisfacción de restricciones principalmente consiste en resolver problemas de existencia más que de optimización; además, las

técnicas enumerativas procedentes del área de la inteligencia artificial utilizadas en su resolución se sustentan, de manera casi exclusiva, en el tratamiento sistemático de las restricciones, mediante técnicas de propagación y satisfacción de restricciones, en todos y cada uno de los subproblemas generados. De esta manera y al utilizar procedimientos de búsqueda arborescentes, estas técnicas se convierten en el arma principal para generar y explorar el menor número de estados posibles del árbol de búsqueda.

De forma similar a como sucede con las técnicas de preproceso procedentes del área de la investigación operativa, estas técnicas de reducción pueden llegar a ser muy efectivas y lo son en mayor grado contra más restrictivo es el problema a resolver (Corominas 1996). Pero en este caso, este argumento adquiere mayor relevancia ya que los procedimientos de resolución del problema de satisfacción de restricciones únicamente utilizan las técnicas de reducción para solucionar el problema, y si sus efectos no son importantes, estos procedimientos pueden llegar a ser extremadamente ineficientes.

A continuación se define brevemente el problema de satisfacción de restricciones, un problema muy estudiado y tratado en el área de la inteligencia artificial, y se exponen diferentes procedimientos para su resolución: técnicas de reducción, en muchos casos desarrolladas *adhoc* en el área de la inteligencia artificial, incorporadas en cada uno de los estados generados por los procedimientos enumerativos de búsqueda utilizados. Cabe destacar que la exposición siguiente únicamente pretende ser una introducción al tema, un área de estudio muy extensa en la que hay una gran cantidad de investigadores trabajando. Para ello, en parte de la exposición se siguen los trabajos presentados por Kumar (1992) y Barták (1998).

Como introduce Barták (1998), los problemas de satisfacción de restricciones han sido estudiados en el área de la inteligencia artificial desde los años setenta, aunque el tratamiento sistemático de las restricciones en programación comenzó a estudiarse en los ochenta. En la programación de restricciones, el proceso de programación consiste en la generación y resolución de las restricciones mediante procedimientos especializados de resolución de éstas. De esta manera, el uso de estas ideas para programar, modelizar y resolver problemas, se ha unido bajo el concepto de programación de restricciones. Le Provost & Wallace (1992) y Barták (1998) señalan que actualmente se distinguen dos ramas de estas técnicas:

1ª) La satisfacción de restricciones. Se puede definir un problema de satisfacción de restricción (*constraint satisfaction problem* o CSP) de la siguiente forma. Dados:

- un conjunto finito de variables $X = \{x_1,\ldots,x_n\}$,
- para cada variable x_i, un conjunto finito D_i de valores posibles (su dominio),

- y un conjunto finito de restricciones que limitan los valores que las variables pueden tomar simultáneamente debido a las relaciones existentes entre ellas;

una solución al problema de satisfacción de restricciones es una asignación de un valor de su dominio a cada variable, de forma que todas las restricciones son satisfechas: como comenta Mackworth (1992), un subconjunto del producto cartesiano de los dominios de las variables. Además y como expone Barták (1998), se puede encontrar una solución factible, todas las soluciones o una solución óptima (para ello, Le Provost & Wallace (1992) matizan que en estos problemas los objetivos del programa también son tratados como restricciones -el proceso concreto se expone en el apartado 3.8.2.4.-).

Se sabe que, desafortunadamente, encontrar soluciones a un problema de satisfacción de restricciones es, en general, un problema NP-completo.

2ª) La resolución de restricciones. Este enfoque difiere de la satisfacción de restricciones en el hecho de utilizar variables con dominios infinitos, y que, por tanto, no es objeto del estudio de esta tesis.

Cabe destacar que existen otros autores (por ejemplo Kumar 1992 y Mackworth 1992) que consideran problemas de satisfacción de restricciones tanto con dominios finitos como infinitos. De todas maneras, de aquí en adelante únicamente se consideran problemas de satisfacción de restricciones con variables de dominio finito (*finite constraint satisfaction problem* o FCSP); de esta forma, cualquier referencia hecha a un CSP se está realizando en realidad a un FCSP. De la misma forma, solamente se consideran problemas de optimización y satisfacción de restricciones con variables de dominio finito, a los que se harce referencia como CSOP (*constraint satisfaction optimization problem*).

Según Kumar (1992) y Barták (1998), para la resolución de problemas de satisfacción de restricciones existen tres tipos de procedimientos: por un lado, están los algoritmos sistemáticos de búsqueda que resuelven el problema únicamente probando las asignaciones posibles de los valores a las variables (expuestos en el apartado 3.8.2.1.); en segundo lugar están las técnicas de consistencia y de propagación local que intentan resolver el problema ajustando al máximo los dominios de las variables (introducidas en el apartado 3.8.2.2.); y, por último, está la propagación de restricciones, un tercer esquema que consiste en integrar un algoritmo de consistencia dentro de un algoritmo de búsqueda (técnicas comentadas en el apartado 3.8.2.3.). Cabe destacar que en este caso también existe cierta confusión en la terminología utilizada, ya que existe una gran cantidad de autores que incorporan las técnicas de propagación de restricciones en el segundo tipo de procedimientos; esta inconcreción puede ser debida al hecho de haber

asociado el término referido a las técnicas de reducción utilizadas junto a un procedimiento de búsqueda, al procedimiento híbrido de búsqueda en sí.

Para finalizar el presente apartado, se expone cómo se puede tratar un problema de optimización con variables de dominio finito como un problema de satisfacción de restricciones (apartado 3.8.2.4.); y, por último, se introduce la denominada programación lógica de restricciones y su relación con los procedimientos de resolución de problemas de satisfacción de restricciones expuestos anteriormente (apartado 3.8.2.5.).

3.8.2.1. Algoritmos sistemáticos de búsqueda.

Los problemas de satisfacción de restricciones pueden solucionarse implementado algoritmos sistemáticos de búsqueda, que resuelven el problema probando las posibles asignaciones a las variables de los valores de sus dominios; por otro lado, y como exponen Le Provost & Wallace (1991), el método que se utiliza por defecto para guiar la evolución de la búsqueda es la estrategia primero en profundidad (expuesta en el apartado 3.3.3.). Estos algoritmos garantizan encontrar una solución, si existe, o prueban que el problema es insoluble; su gran desventaja es que tardan mucho tiempo.

Freuder & Wallace (1992) denominan a este tipo de procedimientos técnicas retrospectivas -para intentar extender una solución parcial se prueba un nuevo valor por *looking back*: se comprueba si el nuevo valor es compatible o no (consistente en la terminología habitual) con los valores seleccionado anteriormente en la solución incompleta-, frente a lo que estos autores denominan técnicas prospectivas -los valores son probados sobre los dominios de las variables que todavía no han sido fijadas en la solución parcial de forma *look ahead*, con el objetivo de descubrir inconsistencias antes que los valores de dichos dominios sean considerados para su inclusión- y que en este texto se consideran procedimientos de propagación de restricciones (apartado 3.8.2.3.).

Entre estos procedimientos sistemáticos de búsqueda, cabe considerar los siguientes:

a) *Generate and Test* (GT). Kumar (1992) y Barták (1998) definen el procedimiento GT como una enumeración, en la que cada combinación posible de las asignaciones de los valores de sus dominios a las variables es sistemáticamente generada y probada, para ver si satisface todas las restricciones; la primera combinación que satisface todas las restricciones es la solución. Presenta la enorme desventaja de generar muchas asignaciones no válidas. Se puede mejorar su eficiencia, si se comprueba la validez de las restricciones cuando sus variables son fijadas (como se hace en los procedimientos siguientes).

b) *Backtracking* (BT). Como exponen diversos autores, Le Provost & Wallace (1991), Kumar (1992), Bacchus & Grove (1995), Barták (1998), etc., el procedimiento BT intenta extender una solución parcial que incorpora valores consistentes para algunas variables hacia una solución completa, seleccionando repetidamente valores consistentes con los valores actuales para otra variable. Las variables son fijadas secuencialmente y tan pronto como todas las variables que participan en una restricción son fijadas, se verifica la validez de dicha restricción; si una solución parcial viola cualquier restricción, se realiza *backtracking* a la variable fijada más recientemente que todavía tiene alternativas disponibles. Como exponen Bacchus & Grove (1995) se realizan comprobaciones hacia atrás: cuando se hace una nueva asignación, se prueba su compatibilidad con todas las asignaciones previas.

Según Le Provost & Wallace (1991), BT es muy ineficiente, aunque una reordenación de las restricciones puede proporcionar leves mejoras. Kumar (1992) y Barták (1998) exponen las tres mayores desventajas del procedimiento BT standard y comentan qué alternativas se han diseñado para superarlas:

- Se producen fracasos repetidos debido a la misma razón, ya que el algoritmo no identifica la razón verdadera de los conflictos: las variables conflictivas. Para evitarlo, se puede realizar un *backtracking* inteligente, retrocediendo directamente hasta la variable que ocasionó el fracaso. En esta dirección están los procedimientos *backjumping* (BJ) y *backmarking* (BM), expuestos a continuación y que como exponen Bacchus & Grove (1995) nunca van peor que BT y generalmente van mucho mejor.

- Se realiza trabajo redundante. Incluso si los valores conflictivos de las variables son identificados durante el *backtracking* inteligente, no son recordados para la detección del mismo conflicto en la siguiente computación. Para evitar este trabajo redundante se utiliza el procedimiento de *backtracking* dirigido por la dependencia (*dependency directed backtracking* o DDB) ya introducido en el apartado 3.6.1.

- El algoritmo BT básico todavía detecta los conflictos demasiado tarde, ya que no es capaz de detectar el conflicto antes de que ocurra. Esta desventaja puede ser evitada aplicado técnicas de consistencia (introducidas en el apartado 3.8.2.2.).

c) *Backjumping* (BJ)[14]. Diversos autores exponen de forma vaga el procedimiento BJ (Freuder & Wallace 1992 y Bacchus & Grove 1995), como una técnica de búsqueda BT que recuerda información sobre fallos previos (guarda un conjunto de pruebas de consistencias) para reducir la necesidad de comprobar restricciones y redescubrir inconsistencias; de esta forma, se detectan y eluden partes del árbol de búsqueda que no

[14] Para una mayor precisión sobre el procedimiento *backjumping* se recomienda consultar Freuder & Wallace (1992), o, como estos mismos autores referencian, Gaschnig (1978).

puede contener ninguna solución. La forma práctica de realización consiste en saltar a variables pasadas cuando el procedimiento BT continuaría trabajando hasta detectar la inconsistencia.

d) *Backmarking* (BM)[15]. El procedimiento BM también es expuesto de forma vaga por algunos autores, Freuder & Wallace (1992) y Bacchus & Grove (1995) entre otros, como una técnica de búsqueda BT que guarda un número de pruebas de consistencias redundantes con sencillas anotaciones, para evitarlos. Cuando se hace la asignación de un valor a una variable, BT comprueba la consistencia de esta asignación con todas las asignaciones previas: si cualquiera de las pruebas de consistencia falla, BM realiza el paso adicional de recordar el primer punto del fallo en una matriz; esta información es utilizada para evitar comprobaciones de consistencia posteriores innecesarias.

3.8.2.2. Técnicas de consistencia y de propagación local.

Las técnicas de consistencia y de propagación local, también denominadas técnicas de consistencia de redes, son técnicas de reducción que buscan restringir el dominio de las variables y fijar alguna de ellas, utilizando localmente las restricciones para excluir los valores localmente inconsistentes (Shirai & Tsujii 1987, Le Provost & Wallace 1991, Hentenryck et al. 1995 y Corominas 1996). De esta forma, su uso exclusivo puede ser utilizado para intentar resolver un CSP: reduciendo al máximo el dominio de las variables de forma que se puedan acabar fijando todas, aunque eso sí, asegurando el cumplimiento de todas las restricciones. De todas maneras, hay que tener presente que los procedimientos que utilizan de forma exclusiva técnicas de consistencia y de propagación local, son incompletos: incluso con propagación, en general para resolver un CSP se necesita algún procedimiento de búsqueda (la unión de ambas técnicas se expone en el apartado 3.8.2.3. bajo el término genérico de propagación de restricciones).

La idea clave de estas técnicas consiste en hallar valores de las variables que sean consistentes, es decir, y según Little & Darby-Dowman (1995), alcanzar un estado en el que las variables de una restricción no contienen valores que podrían violar la restricción en función de los valores de las demás. Rich & Knight (1994) comentan que la principal condición que es necesario que cumplan todas estas reglas de ajuste (reducción) consiste en que la propagación no produzca falsas consistencias, no siendo necesario que se exploten todas las propagaciones posibles.

Little & Darby-Dowman (1995) exponen con más detalle el proceso seguido por estas técnicas: los dominios de las variables pueden cambiar dinámicamente para asegurar que las restricciones son satisfechas; así, cuando las restricciones son consideradas, la

[15] Para una mayor información sobre el procedimiento *backmarking* se recomienda consultar Freuder & Wallace (1992), o, como estos mismos autores referencian, Gaschnig (1977).

consistencia del sistema de restricciones con la que se acaba de considerar, ajusta los dominios de las variables adecuadamente: un cambio del dominio de una variable puede hacer que otras restricciones sean tratadas y, como resultado, otras variables también puedan alterar sus dominios.

Según exponen Le Provost & Wallace (1991), las técnicas de propagación de restricciones aparecen íntimamente ligadas a la noción de dominios finitos; de todas formas, también se han desarrollado generalizaciones que son aplicables a cualquier dominio: es la noción de propagación de dominio independiente. Además, estos autores en el mismo artículo y en Le Provost & Wallace (1992), comentan que la consistencia puede aplicarse como un paso preliminar en la búsqueda (lo que proporciona procedimientos de búsqueda con una etapa inicial de preproceso) o intercalada en ella (lo que conduce a los procedimientos de propagación de restricciones expuestos en el apartado siguiente).

Para poner en práctica estas ideas, se han desarrollado algoritmos de consistencia para redes de restricciones.

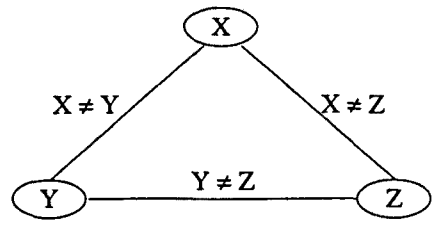
Como expone Barták (1998), una restricción puede afectar cualquier número de variables desde 1 a n, aunque se pueden distinguir dos casos particulares: las restricciones unarias (que pueden ser descartadas directamente con preproceso y reducción de los dominios de las variables afectadas) y las binarias. Si todas las restricciones de un CSP son unarias y binarias, las variables y las restricciones pueden representarse en un grafo o red de restricciones, y el algoritmo de satisfacción de restricciones puede explotar las técnicas de búsqueda en grafos (Shirai & Tsujii 1987 y Barták 1998). Esto es interesante ya que cualquier restricción de grado más alto puede ser expresada en términos de restricciones binarias, de forma que los CSPs binarios son representativos de todos los CSPs. De todas maneras, cabe destacar que Mackworth (1992) comenta que este tipo de representación también puede generalizarse a restricciones de grado k utilizando hipergrafos.

Barták (1998) señala que la mayoría de algoritmos de satisfacción de restricciones se restringen a CSPs en los que las restricciones son unarias o binarios; así, los CSPs son referidos comúnmente como CSPs binarios. Un CSP binario puede ser representado por un grafo de restricciones (a veces denominado red de restricciones), en el que cada vértice representa una variable y cada arco representa una restricción entre las variables que representan los vértices que une el arco. Una restricción unaria es representada formalmente por un bucle sobre sí misma, y, como ya se ha comentado, puede ser inmediatamente satisfecha al reducir el dominio de la variable restringida; de esta forma puede ser eliminada del grafo de restricciones, simplificando el problema a resolver. A continuación se muestra un ejemplo:

Sistema de restricciones

$X \neq Y$
$Y \neq Z$
$X \neq Z$

Red de restricciones



Grafo de restricciones de un sistema de restricciones, de Barták (1998).

En este apartado se van a introducir técnicas de consistencia basadas en grafos de restricciones binarias, ya que, como se concreta a continuación y ya se ha comentado anteriormente, es posible convertir cualquier CSP con restricciones de grado n en un CSP binario. Aunque los algoritmos de consistencia en grafos de restricciones binarias son los más sencillos y la posibilidad de expresar restricciones de mayor grado en términos de restricciones binarias es importante desde el punto de vista teórico, Barták (1998) también expone que en la práctica la binarización de restricciones no es apropiada y las aplicaciones prácticas tienden a trabajar con restricciones de grado n, más que con restricciones binarias. En esta misma dirección, Regin (1998) comenta en *The 13th Biennial European Conference on Artificial Intelligence*, que en los últimos años se ha prestado una gran atención a la modelización y resolución de problemas utilizando restricciones unarias y binarias, y que sólo recientemente se está dando más importancia a las restricciones no binarias, que puede ser vistas como restricciones más globales. Así, el uso de restricciones no binarias es una vía prometedora para modelizar y resolver problemas industriales, aunque este tema se encuentra en etapas iniciales de investigación.

A continuación se detalla el procedimiento de conversión de un CSP arbitrario en un CSP binario equivalente, que se expone en Barták (1998). Este proceso se basa en la idea de introducir una nueva variable, denominada variable de encapsulado, que encapsula el conjunto de variables restringidas y que tiene asignado como dominio, también finito, el producto cartesiano de los dominios de las variables. A continuación se muestra un ejemplo aclarativo:

Variables originales y sus dominios	Variable de encapsulado y su dominio
X: [1,2]	U: [(1,3,5),(1,3,6),
Y: [3,4]	(1,4,5),(1,4,6),
Z: [5,6]	(2,3,5),(2,3,6),
	(2,4,5),(2,4,6)]

Dominio de una variable de encapsulado dadas las variables originales, de Barták (1998).

En este momento una restricción de grado n puede ser sustituida por una variable de encapsulado con el dominio que corresponde a la restricción, una vez convertida la restricción de grado n en una restricción unaria equivalente de la variable de encapsulado y al satisfacer la restricción unaria reduciendo el dominio de las variables encapsuladas (Barták 1998). De nuevo un ejemplo:

Variables y restricción original	Variable de encapsulado y dominio reducido
X + Y = Z X: [1,2]; Y: [3,4]; Z: [5,6]	U: [(1,4,5),(2,3,5),(2,4,6)]

Dominio de una variable de encapsulado dadas las variables originales y la restricción que las une, de Barták (1998).

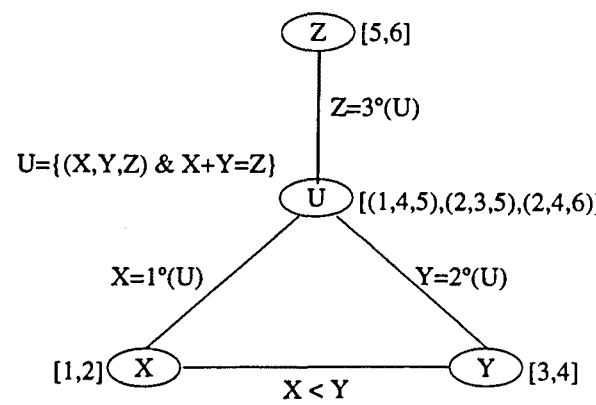
La transformación expuesta resuelve restricciones individuales, con lo que falta combinar estas soluciones individuales con la solución del sistema de restricciones. Según Barták (1998) existen las dos maneras de introducir las variables de encapsulado en el grafo de restricciones que representa el CSP convertido.

1ª) Con las variables originales. En este enfoque se combinan las variables originales y encapsuladas para expresar el CSP no-binario original, de forma que las variables de encapsulado se corresponden directamente con las restricciones originales. Las nuevas restricciones introducidas relacionan las variables originales y encapsuladas de la manera siguiente:

X=i_posición_en_el_argumento_de(U),

donde X es la variable original, U es la variable de encapsulado e i es la "la posición de X dentro de U".

Sea el siguiente ejemplo:

CSP no-binario original	CSP binario equivalente
X + Y = Z, X < Y X: [1,2]; Y: [3,4]; Z: [5,6]	

Grafo de restricciones con variables originales y de encapsulado, de Barták (1998).

La ventaja de este enfoque es la conservación de las variables originales, de forma que la solución del CSP original puede ser obtenida directamente de la solución del CSP binario; además, se conservan las restricciones binarias del CSP original.

2ª) Sin variables originales. El segundo enfoque se basa utilizar únicamente variables de encapsulado, de forma que las nuevas restricciones introducidas relacionan esas variables por medio de componentes comunes, de la manera siguiente:

$$i\_posición\_en\_el\_argumento\_de(U) = j\_posición\_en\_el\_argumento\_de(V),$$

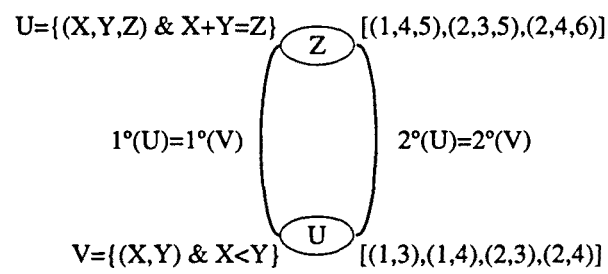
donde U y V son variables de encapsulado e i y j son respectivamente "las posiciones del componente común".

Sea el ejemplo siguiente:

CSP no-binario original

$X + Y = Z;$
$X < Y$
X: [1,2]; Y: [3,4]; Z: [5,6]

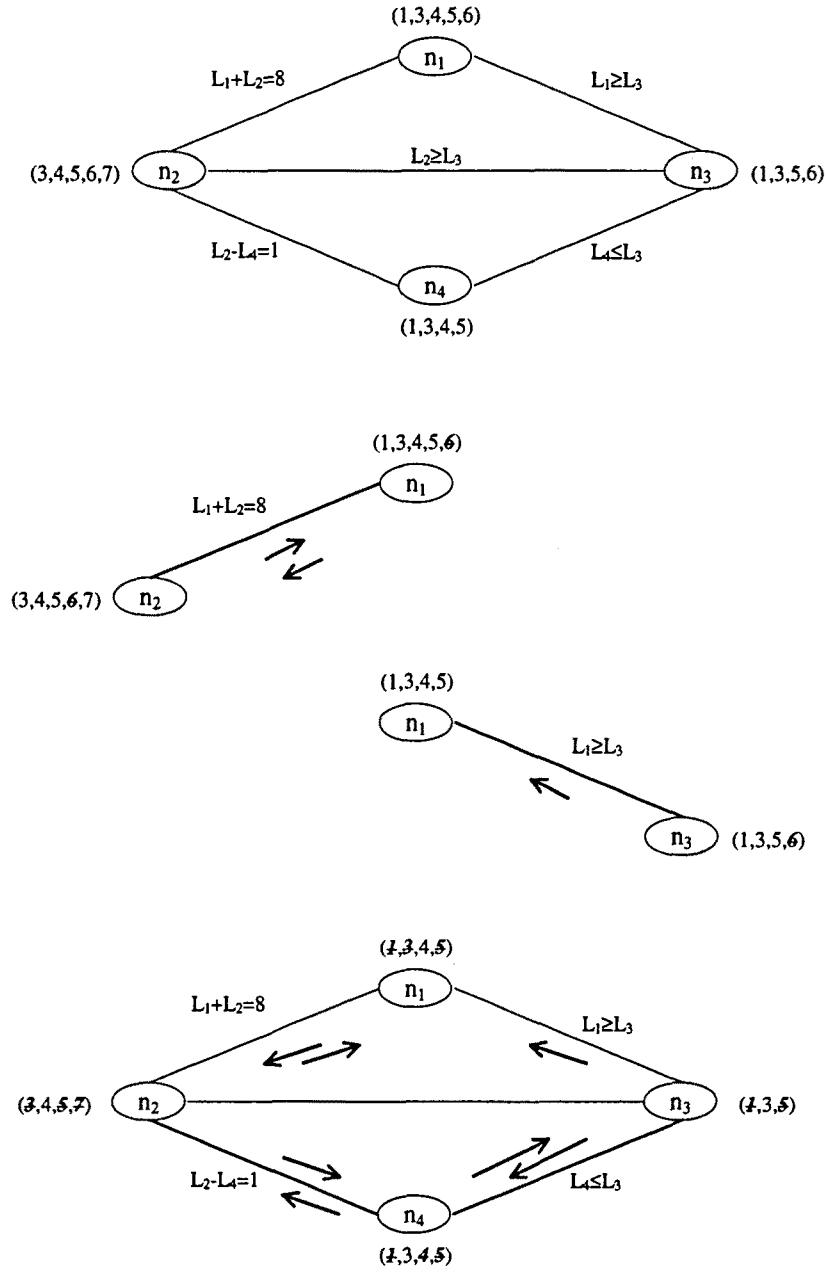
CSP binario equivalente



Grafo de restricciones únicamente con variables de encapsulado, de Barták (1998).

En este enfoque, cada restricción del CSP original es representada por una variable de encapsulado. La red de restricciones resultante es menor que la que se obtiene con el método anterior, sin embargo, la solución (valor de las variables originales) ha de ser extraída del valor de las variables de encapsulado.

Una vez se dispone del grafo de restricciones, se pueden aplicar técnicas de consistencia al CSP binario, para intentar resolverlo. A continuación se expone un ejemplo extraído de Shirai & Tsujii (1987), en el que se puede comenzar a intuir el funcionamiento de estas técnicas. De todas formas, a continuación se exponen diversos algoritmos que permiten formalizar dichas técnicas.



Resolución de un CSP binario mediante técnicas de consistencia en el grafo de restricciones, de Shirai & Tsujii (1987).

Según Barták (1998), para el ajuste y resolución de CSPs binarios se han desarrollado diversas técnicas de consistencia para grafos de restricciones, con el objetivo de podar el espacio de búsqueda. No obstante, y como ya se ha comentado, las técnicas de consistencia raramente se usan solas para resolver problemas de satisfacción de restricciones, y se aplican lo antes posible en combinación con un procedimiento enumerativo, dejando la búsqueda para cuando ya no hay más propagación a realizar. Las técnicas de consistencia van desde la consistencia de vértice y la consistencia de arco, a la completa, pero cara de calcular, consistencia de camino (Kumar 1992 y Barták 1998).

a) Consistencia de vértice. El vértice que representa a una variable V en el grafo de restricciones G es consistente, si para cada valor x del dominio actual de V cada restricción unaria sobre V es satisfecha. Si el dominio D de una variable V contiene un valor "x" que no satisface la restricción unaria sobre V, el vértice es inconsistente; se puede obtener la consistencia del vértice eliminando los valores del dominio D de cada variable V que no satisfacen la restricción unaria sobre V.

Barták (1998) propone el algoritmo NC para obtener una red de restricciones vértice-consistente:

```
Procedimiento NC
   Para cada V ∈ vértices(G) hacer
      Para cada x ∈ DV hacer
         Si cualquier restricción unaria sobre V es inconsistente con x entonces
            Borrar x de DV;
         Fin_si;
      Fin_para;
   Fin_para;
Fin NC
```

b) Consistencia de arco. Si el grafo de restricciones es vértice-consistente entonces las restricciones unarias pueden eliminarse, y, en CSPs binarios, sólo queda asegurar la consistencia de las restricciones binarias que se corresponden con los arcos del grafo de restricciones: consistencia de arco.

Según exponen Kumar (1992), Mackworth (1992) y Barták (1998) entre otros, el arco (V_i,V_j) es consistente, si para cada valor x del dominio actual de V_i existe algún valor y en el dominio de V_j tal que $V_i = x$ y $V_j = y$ está permitido por la restricción binaria entre V_i y V_j; de esta manera un arco (V_i,V_j) puede hacerse consistente, eliminando los valores del dominio de V_i para los que no existe un valor correspondiente en el dominio de V_j tal que la restricción binaria entre V_i y V_j es satisfecha. Como matizan estos autores, cabe destacar que el concepto de arco-consistencia es direccional: un arco (V_i,V_j) consistente no significa automáticamente que (V_j,V_i) sea también consistente.

Hentenryck et al. (1995) comentan que la arco-consistencia puede ser forzada en tiempo $O(ed^2)$, donde d es el tamaño del dominio más largo y e es el número de restricciones. Kumar (1992) y Barták (1998), además exponen diversos algoritmos para obtenerla.

- Algoritmo REVISE:

```
Procedimiento REVISE (Vi,Vj)
   BORRAR ← falso;
   Para cada x ∈ Di hacer
      Si no existe un y ∈ Dj tal que (x,y) es consistente, entonces
         Borrar x de Di;
         BORRAR ← cierto;
      Fin_si;
   Fin_para;
   Devolver BORRAR;
Fin REVISE
```

Cabe destacar que, para hacer el grafo de restricciones arco-consistente, no es suficiente ejecutar REVISE para cada arco una sola vez: una vez REVISE reduce el dominio de alguna variable V_i, todo arco (V_j,V_i) previamente revisado ha de se revisado de nuevo. El algoritmo siguiente, denominado AC-1, realiza esta función.

- Algoritmo AC–1:

```
Procedimiento AC-1
   Q ← {(Vi,Vj) ∈ arcos(G), i≠j};
   Repetir
      CAMBIO ← falso;
      Para cada (Vi,Vj) ∈ Q hacer
         CAMBIO ← REVISE(Vi,Vj) o CAMBIO;
      Fin_para;
   hasta no(CAMBIO);
Fin AC-1
```

Este algoritmo no es muy eficiente ya que la revisión de un arco en alguna iteración fuerza a una nueva revisión de todos ellos en la siguiente iteración, cuando los únicos arcos afectados por la reducción del dominio de V_k son los arcos (V_i,V_k); además, si se revisa el arco (V_k,V_m) y el dominio de V_k se reduce, no es necesario revisar de nuevo el arco (V_m,V_k). El algoritmo siguiente, llamado AC-3, sólo revisa de nuevo aquellos arcos que podrán estar afectados por una revisión previa.

- Algoritmo AC-3:

```
Procedimiento AC-3
  Q ← {(Vi,Vj) ∈ arcos(G), i≠j};
  Mientras que Q no este vacío hacer
    Seleccionar y borrar cualquier arco (Vk,Vm) de Q;
    Si REVISE(Vk,Vm) entonces
      Q ← Q ∪ {(Vi,Vk) tales que (Vi,Vk) ∈ arcos(G), i≠k, i≠m};
    Fin_si;
  Fin_mientras;
Fin AC-3
```

Cuando el algoritmo AC-3 revisa los arcos por segunda vez, prueba de nuevo muchas parejas de valores que ya se sabe, desde la iteración anterior, que son consistentes o inconsistentes y que no son afectadas por la reducción del dominio. Estas pruebas constituyen una fuente potencial de ineficacia, y, para evitarlas, se ha diseñado el algoritmo AC-4 que se expone en Barták (1998).

- Algoritmo AC-4. Este procedimiento intenta refinar la manipulación de los arcos, trabajando con parejas de valores individuales. En primer lugar, el algoritmo AC-4 inicializa sus estructuras internas que se utilizan para recordar parejas de valores consistentes (inconsistentes) de las variables -vértices- incidentes (la estructura S_i,x); además, se cuentan los valores de "soporte" del dominio de la variable donde incide el arco (la estructura $contador_{(i,j),x}$); y se eliminan aquellos valores que no tienen soporte. Una vez se elimina un valor del dominio de una variable, el algoritmo agrega la pareja <Variable,Valor> a la lista Q para la nueva revisión de los valores afectados de las variables correspondientes.

Algoritmo INITIALIZE.

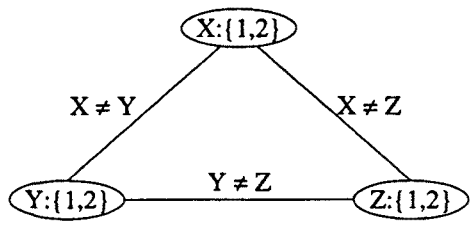
```
Procedimiento INITIALIZE
  Q ← {};
  Para cada (Vi,Vj) ∈ arcos(G) hacer
    Para cada x ∈ Di hacer
      total ← 0;
      Si,x ← {}; inicializar sólo para el primer arco (Vi,Vj)
      Para cada y ∈ Dj hacer
        Si (x,y) es consistente según la restricción (Vi,Vj) entonces
          total ← total+1;
          Si,x ← Si,x ∪ {<j,y>};
        Fin_si;
      Fin_para;
      contador[(i,j),x] ← total;
      Si contador[(i,j),x]=0 entonces
        Borrar x de Di;
        Q ← Q ∪ {<i,x>};
      Fin_si;
    Fin_para;
  Fin_para;
  Devolver Q;
Fin INITIALIZE
```

Después de la inicialización, el algoritmo AC-4 únicamente revisa de nuevo las parejas de valores de variables incidentes que han sido afectadas por una revisión previa:

```
Procedimiento AC-4
  Q ← INITIALIZE;
  Mientras que Q no este vacío hacer
    Seleccionar y borrar cualquier pareja <j,y> de Q;
    Para cada <i,x> de Sj,y hacer
      contador[(i,j),x] ← contador[(i,j),x] - 1;
      Si contador[(i,j),x]=0 & x está todavía en Di entonces
        Borrar x de Di;
        Q ← Q ∪ {<i,x>};
      Fin_si;
    Fin_para;
  Fin_mientras;
Fin AC-4
```

- Algoritmos AC-5, AC-6, AC-7, etc. Como expone Barták (1998) existen otros algoritmos para obtener la arco-consistencia de un grafo de restricciones; de todas formas, este mismo autor opina que los algoritmos AC-3 y AC-4 son los más ampliamente utilizados.

Si el tamaño del dominio de toda variable llega a ser uno, el CSP tiene exactamente una solución. De todas maneras y como ya se ha comentado, habitualmente no se obtiene respuesta, ya que los procedimientos que utilizan de forma exclusiva técnicas de consistencia y de propagación local, son incompletos. Además, y como muestra el ejemplo expuesto en Barták (1998), se puede obtener un grafo de restricciones arco-consistente con los dominios de las variables no vacíos, pero que no contengan ninguna solución que satisfaga todas las restricciones:



Grafo de restricciones arco-consistente pero sin ninguna solución que satisfaga todas las restricciones, de Barták (1998).

c) K-consistencia (consistencia de camino). Como se ha comprobado, la consistencia de arcos no es suficiente para eliminar la necesidad de retroceso, aunque si se extiende la consistencia a dos o más arcos se pueden eliminar más valores inconsistentes: es la denominada K-consistencia.

Un grafo es K-consistente según describen Kumar (1992) y Barták (1998), si fijadas K-1 variables a unos valores que satisfacen todas las restricciones entre dichas variables y seleccionada cualquier otra K-ésima variable, existe un valor para esa variable que satisface todas las restricciones entre dichas K variables; un grafo es fuertemente K-consistente si es J-consistente para toda $J \leq K$. La consistencia de vértice es equivalente a una 1-consistencia fuerte y la arco-consistencia es equivalente a una 2-consistencia fuerte. Existen algoritmos para hacer un grafo de restricciones fuertemente K-consistente para $K > 2$, pero en la práctica son raramente utilizados debido a su escasa eficiencia[16].

La excepción es el algoritmo para hacer un grafo de restricciones fuertemente 3-consistente, que es comúnmente denominado consistencia de camino; no obstante, incluso este algoritmo es demasiado lento y se describen formas débiles de consistencia de camino. Un vértice que representa la variable V_i está en un camino consistente, si es arco-consistente (todos los arcos que lo tienen como origen son arco-consistentes) y

[16] En Kumar (1992) se puede encontrar alguna referencia sobre el procedimiento concreto.

para cada valor x del dominio D_i de la variable V_i que acaba de soportar al valor y del dominio de la variable que incide V_j, existe un valor z del dominio de otra variable incidente V_k tal que (x,z) está permitido por la restricción binaria entre V_i y V_k y (z,y) está permitido por la restricción binaria entre V_k y V_j.

Como señala Barták (1998), si un grafo de restricciones que contiene n vértices es fuertemente n-consistente, entonces se puede encontrar una solución del CSP sin ninguna búsqueda; el único inconveniente reside en que la complejidad de estos algoritmos es exponencial. De esta manera, si el grafo es (fuertemente) K-consistente para $K < n$, en general, el retroceso no puede evitarse.

A modo de resumen, se puede comentar que en este apartado se han expuesto diversas técnicas de consistencia concretadas en algoritmos de consistencia para redes de restricciones; de todas formas, hay que tener presente que en el área de la inteligencia artificial se continúa trabajando en este tema[17]. Una consideración a recordar es el hecho de que todas esta técnicas pueden ser consideradas técnicas de reducción, fácilmente incorporables en multitud de procedimientos de búsqueda enumerativos.

3.8.2.3. Propagación de restricciones.

En los apartados anteriores se han presentado dos esquemas diferentes e independientes, para resolver el problema de satisfacción de restricciones: los algoritmos sistemáticos de búsqueda y las técnicas de consistencia. Como exponen diferentes autores (Le Provost & Wallace 1991, Kumar 1992, Le Provost & Wallace 1992, Rich & Knight 1994, Bacchus & Grove 1995, Little & Darby-Dowman 1995, Barták 1998, etc.), existe un tercer esquema consiste en integrar un algoritmo de consistencia dentro de un algoritmo enumerativo de búsqueda. El esquema básico de funcionamiento de estos procedimientos consiste en intentar extender una solución parcial que incorpora valores consistentes hacia una solución completa, seleccionando repetidamente valores para otra variable; después de asignar un valor a una variable se aplica alguna técnica de consistencia al grafo de restricciones, con lo que reduce los dominios de las variables restantes. La búsqueda puede hacerse en forma de árbol, donde cada rama se corresponde con los valores posibles de las variables, cada vértice contiene asignaciones parciales y la propagación de valores se utiliza para ajustar la asignación parcial presente en cada vértice. Como expone Barták (1998), dependiendo del grado de la técnica de consistencia se obtienen diversos algoritmos de propagación de restricciones que varían en coste y calidad del ajuste.

[17] En Kumar (1992) y Barták (1998) se expone un nuevo procedimiento basado en la ordenación de las variables en el momento de ser consideradas; de todas formas tampoco se diseña un algoritmo polinomial para la obtención de la solución óptima.

Freuder & Wallace (1992) denominan a este tipo de procedimientos técnicas prospectivas: los valores son probados sobre los dominios de las variables que todavía no han sido fijadas en la solución parcial, de forma que se pueden descubrir inconsistencias antes que los valores de dichos dominios sean considerados para su inclusión. Estas técnicas miran hacia adelante para establecer formas de consistencia local antes que la búsqueda continúe hacia la solución global, eliminando los valores que no cumplen los criterios de consistencia local: cuando se hace una nueva asignación, se comprueba la consistencia hacia el futuro y se utilizan las implicaciones de dichas pruebas (Bacchus & Grove 1995).

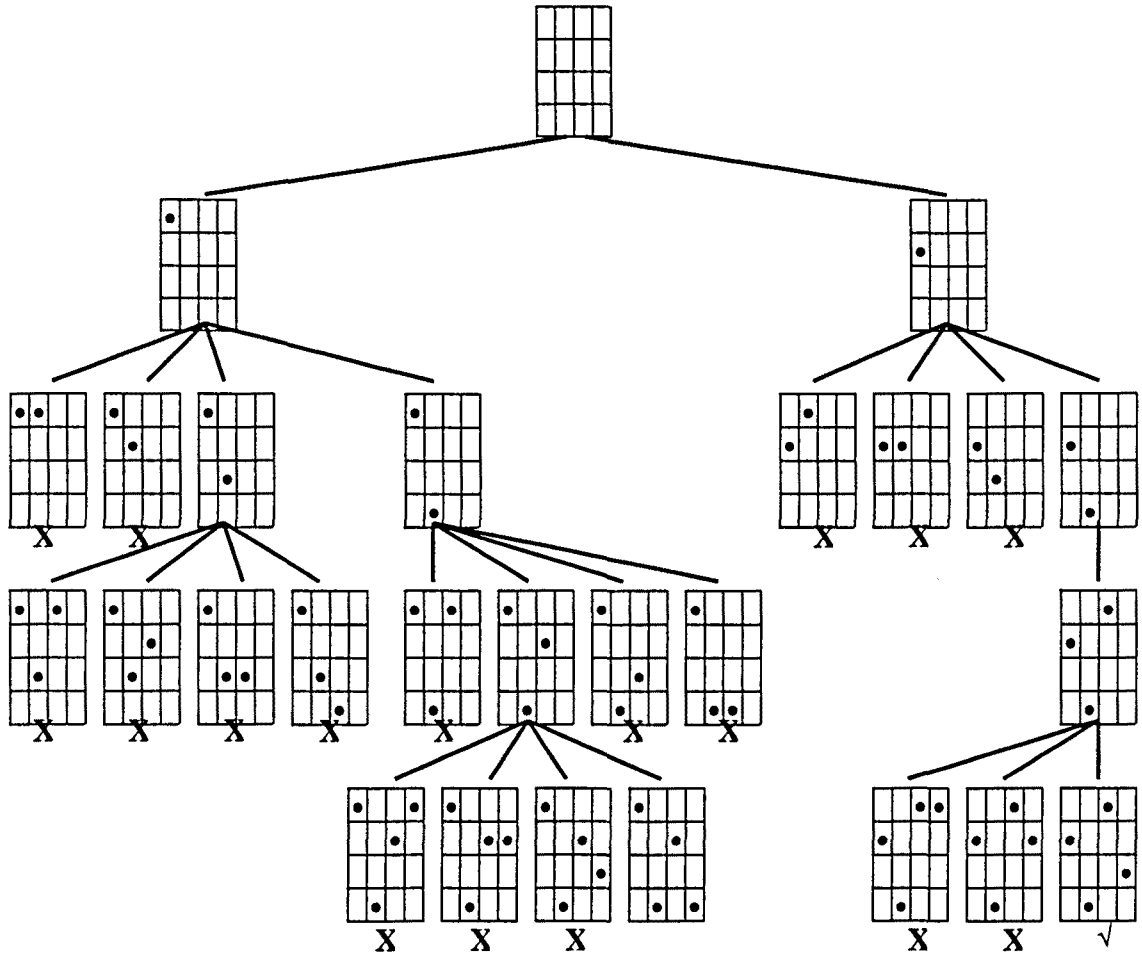
Según una gran parte de los autores referenciados, Freuder & Wallace (1992) y Le Provost & Wallace (1992) entre otros, la forma más comúnmente utilizada de consistencia local es la arco-consistencia y el algoritmo más usado es el *forward checking*; aunque también se han diseñado otros que difieren, según Kumar (1992), en el grado de arco-consistencia que se realiza en los vértices del árbol de búsqueda.

a) *Backtracking* (BT). Para Kumar (1992) y Barták (1998), incluso el *backtracking* simple realiza algún tipo de técnica de consistencia, pudiendo ser visto como una combinación de *generate and test* y una fracción de arco-consistencia. El algoritmo BT prueba arco-consistencia entre las variables asignadas: verifica la validez de las restricciones considerando las asignaciones parciales. Como a las variables se asigna un único valor, es posible verificar únicamente los arcos que contienen la última variable fijada; si su dominio se reduce, entonces la restricción correspondiente no es consistente y el algoritmo retrocede a una nueva asignación. De esta manera, el algoritmo BT detecta las inconsistencias tan pronto como aparecen.

Algoritmo AC-3 para *Backtracking*, según Barták (1998):

```
Procedimiento AC3-BT(cv)
   Q ← {(Vi,Vck) ∈ arcos(G), i<cv};
   Consistente ← cierto;
   Mientras que Q no este vacío & Consistente hacer
      Seleccionar y borrar cualquier arco (Vk,Vm) de Q;
      Consistente ← REVISE(Vk,Vm);
   Fin_mientras;
   Devolver Consistente;
Fin AC3-BT
```

A continuación, se expone el árbol de búsqueda generado en la resolución del problema de las 4-reinas mediante el algoritmo BT:



Problema de las 4-reinas resuelto mediante el algoritmo BT, de Barták (1998).

Barták (1998) comenta que el algoritmo BT puede extenderse para retroceder a la variable conflictiva, incorporando algún tipo de retroceso inteligente como los introducidos en el apartado 3.8.2.1.; no obstante, también señala que este retroceso agrega algún gasto adicional al algoritmo y le parece que la prevención de posible futuros conflictos es más razonable que retroceder desde ellos.

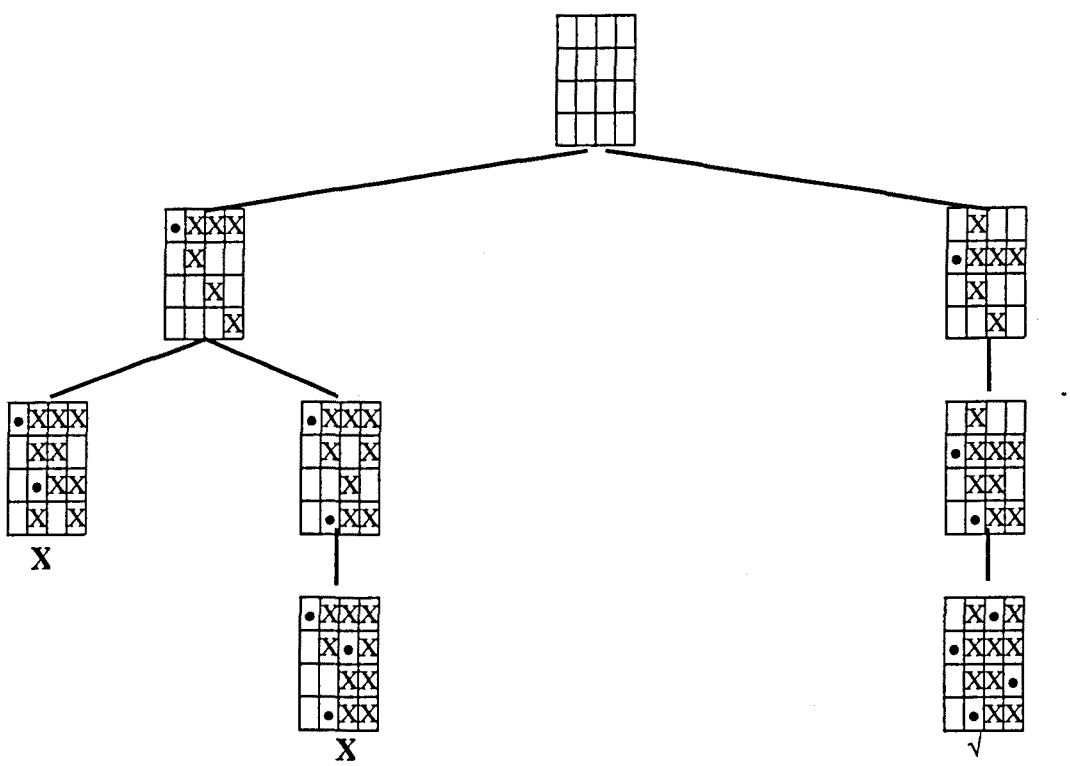
b) *Forward Checking* (FC). Kumar (1992) y Barták (1998) comentan que el algoritmo *forward checking* es la manera más sencilla de prevenir conflictos futuros. En vez de probar arco-consistencia de las variables fijadas, realiza una forma restringida de arco-consistencia con variables todavía no fijadas: entre la variable actual y las variables futuras. Cuando se asigna un valor a la variable actual, cualquier valor del dominio de una variable "futura" que entre en conflicto con esta asignación es (temporalmente) eliminado del dominio. La ventaja reside en que si el dominio de una variable futura llega a ser vacío, se conoce inmediatamente que la solución parcial actual es inconsistente. Cabe destacar que cuando una nueva variable es considerada, se garantiza que todos sus valores restantes son consistentes con las variables ya fijadas, de forma que la comprobación de una nueva asignación con las asignaciones pasadas no es

necesaria. El algoritmo FC detecta inconsistencias antes que BT, lo que reduce más el árbol de búsqueda, y, según Barták (1998), funciona casi siempre mucho mejor que BT.

Algoritmo AC-3 para *Forward Checking*, según Barták (1998):

```
Procedimiento AC3-FC(cv)
   Q ← {(Vi,Vck) ∈ arcos(G), i>cv};
   Consistente ← cierto;
   Mientras que Q no este vacío & Consistente hacer
      Seleccionar y borrar cualquier arco (Vk,Vm) de Q;
      Si REVISE(Vk,Vm) entonces
         Consistente ← Dk no vacío;
      Fin_si;
   Fin_mientras;
   Devolver Consistente;
Fin AC3-FC
```

A continuación, se expone el árbol de búsqueda generado en la resolución del problema de las 4-reinas mediante el algoritmo FC:



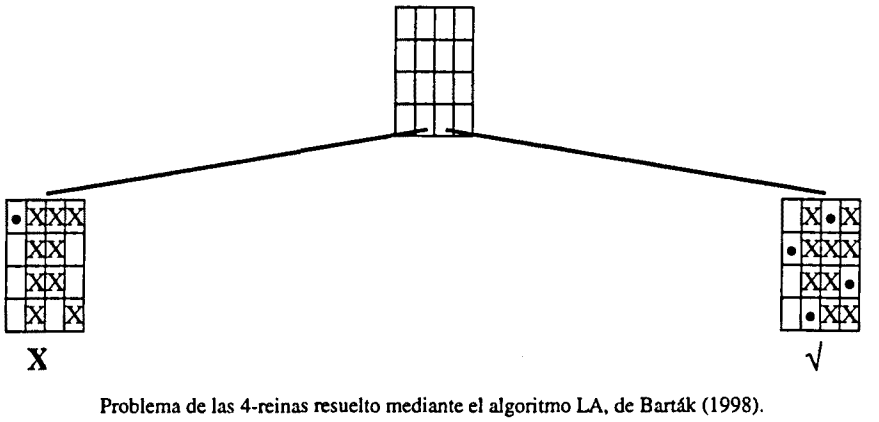
Problema de las 4-reinas resuelto mediante el algoritmo FC, de Barták (1998).

c) *Look Ahead* (LA). Como exponen Kumar (1992) y Barták (1998), el algoritmo FC únicamente comprueba la consistencia entre la variable actual y las variables futuras; pero, ¿por qué no ejecutar una plena arco-consistencia que reduciría los dominios y eliminaría posibles conflictos? Este enfoque se denomina *look ahead* (LA) o mantenimiento de la arco-consistencia (MAC). La ventaja que presenta el algoritmo LA, consiste en que además detecta los conflictos entre las variables futuras y, por tanto, se pueden podar ramas del árbol de búsqueda que conducirían a fracaso antes que con FC. En este caso, la comprobación de una nueva asignación con las asignaciones pasadas tampoco no es necesaria.

Algoritmo AC-3 para *Look Ahead*, según Barták (1998):

```
Procedimiento AC3-LA(cv)
   Q ← {(Vi,Vck) ∈ arcos(G), i>ck};
   Consistente ← cierto;
   Mientras que Q no este vacío & Consistente hacer
      Seleccionar y borrar cualquier arco (Vk,Vm) de Q;
      Si REVISE(Vk,Vm) entonces
         Q ← Q ∪ {(Vi,Vk) tal que (Vi,Vk) ∈ arcos(G),i≠k, i≠m, i>cv};
         Consistente ← Dk no vacío;
      Fin_si;
   Fin_mientras;
   Devolver Consistente;
Fin AC3-LA
```

A continuación, se expone el árbol de búsqueda generado en la resolución del problema de las 4-reinas mediante el algoritmo LA:



Problema de las 4-reinas resuelto mediante el algoritmo LA, de Barták (1998).

Como ya se ha introducido al inicio del presente apartado, dependiendo del grado de la técnica de consistencia se obtienen diversos algoritmos de propagación de restricciones.

Una mayor propagación de restricciones en cada vértice implica que el árbol de búsqueda contiene menos vértices, pero el coste total puede ser más alto ya que el procesamiento en cada vértice es más caro. En un extremo, la obtención de una fuerte n-consistencia para el problema original eliminaría completamente la necesidad de búsqueda, pero esto es comúnmente más caro que un procedimiento más simple; ésta es la razón por la que los procedimientos anteriores todavía se utilizan todavía en aplicaciones (Kumar 1992 y Barták 1998).

En un algoritmo de búsqueda para la resolución de un problema de satisfacción de restricciones, se requiere conocer tanto el orden en el que las variables son consideradas para ser fijadas, como el orden en el que los valores son asignados a la variable en el retroceso. Además, en el área de la inteligencia artificial también es conocido que elegir un correcto orden de variables (y valores) puede mejorar notablemente la eficiencia de la búsqueda.

En cuanto a la ordenación de las variables, Kumar (1992) y Barták (1998) distinguen entre una ordenación estática, en la que el orden de las variables se especifica antes del comienzo de la búsqueda y no se cambia, y una ordenación dinámica, en que la elección de la próxima variable a ser considerada en cualquier momento depende del estado actual de la búsqueda. Concretamente, se han desarrollado varias heurísticas para seleccionar la ordenación de las variables (Kumar 1992, Rich & Knight 1994 y Barták 1998).

- La heurística utilizada más comúnmente se basa en el principio "primero-fracaso": se selecciona de forma dinámica para asignar la variable con menos alternativas restantes. El principio anterior se basa en que si la solución parcial actual no conduce a una solución completa, contra más pronto se descubra, mejor. Según Kumar (1992), al provocar fracasos tempranos, esta heurística debería reducir la profundidad promedio de las ramas en el árbol de búsqueda.

- Otra heurística que se aplica cuando todas las variables tienen el mismo número de valores, consiste en seleccionar la variable que participa en más restricciones.

- Además, se utiliza una heurística que consiste en elegir la variable que tiene relación con el mayor número de restricciones con las variables ya fijadas, con el objetivo de detectar conflictos lo antes posible.

En cuanto al orden en el que los valores son considerados una vez se selecciona la variable a fijar, Kumar (1992) y Barták (1998) comentan que puede tener un impacto considerable sobre el tiempo necesario para encontrar la primera solución; sin embargo,

si se buscan todas las soluciones o no hay soluciones, la ordenación del valor es indiferente. Concretamente, se han desarrollado las siguientes heurísticas:

- Preferir aquellos valores que maximizan el número de opciones disponibles. Es posible contar "lo prometedor" de cada valor, con el producto de los tamaños de los dominios de las variables futuras después de seleccionar ese valor: se selecciona el valor de mayor promesa. También es posible calcular el porcentaje de valores de los dominios futuros que no serán utilizables y elegir el valor de menor porcentaje.

- Preferir el valor que conduce al CSP más fácil de resolver, lo que requiere estimar la dificultad de resolver un CSP. Existe un método que propone convertir un CSP en un árbol estructurado borrando un número mínimo de arcos y entonces encontrar todas las soluciones del CSP resultante (contra más soluciones se encuentran, más fácil es el CSP).

3.8.2.4. Resolución de los problemas de optimización.

La solución de un problema de satisfacción de restricciones consiste en la asignación a cada variable de un valor de su dominio, de forma que todas las restricciones sean satisfechas. Si además se desea la optimización de una cierta función objetivo, se está frente a un problema de optimización y satisfacción de restricciones con variables de dominio finito: CSOP (*constraint satisfaction optimization problem*).

Para la resolución de este tipo de problemas es posible utilizar todas las técnicas expuestas para resolver los problemas de satisfacción de restricciones, si la función objetivo a optimizar es tratada como una restricción más. En este caso se soluciona un problema de optimización resolviendo diversos problemas de existencia. El procedimiento concreto es muy sencillo (Little & Darby-Dowman 1995, Prestwich & Mudambi 1995, Corominas 1996, Vila & Coves 1998, etc.): en el problema de satisfacción de restricciones se incluye una nueva restricción de coste que obliga a que el valor de la solución a obtener sea menor (en caso de minimizar) que el valor de la mejor solución conocida hasta el momento; esta restricción se va modificando cada vez que se obtiene una solución mejor y hasta que no existe ninguna otra solución factible mejor. Sea f(X) la función objetivo a optimizar, con X el conjunto de variables, y $\overline{Z}$ el valor de la solución preferible; en caso de minimizar, la nueva restricción será $f(x) < \overline{Z}$. Una vez se encuentra una solución mejor que la preferible, se puede optar por una de las dos siguientes estrategias:

- comenzar la búsqueda de nuevo desde el inicio,

- continuar la búsqueda una vez actualizada la restricción $f(x) < \overline{Z}$ (en caso de minimizar).

Como exponen Little & Darby-Dowman (1995), este procedimiento puede ser visto como una implementación de las técnicas *branch and bound* ("... *This is an implementation of Branch and Bound technique, ...*", p. 17). Por su parte, Vila & Coves (1998) comentan la posibilidad de utilizar el método *branch and bound* ya que, excepto en la primera iteración, en las otras ya se dispone de una solución factible que permite podar aquellos vértices de peor cota. Otros autores como Beale (1997) resuelven problemas de optimización combinatoria utilizando procedimientos *branch and bound* en los que integran técnicas de propagación de restricciones en cada vértice; concretamente, Beale (1997) resuelve el problema del coloreado de grafos incorporando una rutina de arco-consistencia parecida al algoritmo AC-4 expuesto en el apartado 3.8.2.2.

3.8.2.5. Programación lógica de restricciones (*Constraint Logic Programming* o CLP).

En la programación de restricciones, el proceso de programación se basa en el tratamiento sistemático de las restricciones y en su resolución mediante procedimientos especializados, como las ya comentados. Otros autores, como por ejemplo Frühwirth et al. (1992), exponen un nuevo sistema de programación denominado programación lógica (*logic programming*), cuyo lenguaje de programación más conocido es el Prolog, y que es diferente de los lenguajes convencionales ya que utiliza la lógica para declarar estados del problema y la deducción para su resolución.

En la resolución de problemas de optimización y de satisfacción de restricciones, se comenzó utilizando procedimientos de resolución basados en generación y prueba, pero a partir de los años 80 se han comenzado a aplicar técnicas de manipulación y de propagación de restricciones. Estas técnicas de programación constituyen lo que hoy se conoce como programación lógica de restricciones (*Constraint Logic Programming* o CLP).

Básicamente y como exponen diversos autores: Dincbas et al. (1988), Le Provost & Wallace (1991), Frühwirth et al. (1992), Hentenryck et al. (1992), Little (1993), Jaffar & Maher (1994), Bisdorff & Laurent (1995), Little & Darby-Dowman (1995), Prestwich & Mudambi (1995), Corominas (1996), Darby-Dowman et al. (1997) y Vila & Coves (1998) entre otros, la programación lógica de restricciones, CLP, es una nueva clase de lenguaje que combina la declarativa de un lenguaje de programación lógica como

Prolog, con la eficiencia de la resolución de restricciones mediante técnicas de propagación de restricciones; en este sentido, se ha potenciado el uso de técnicas de consistencia y de propagación local para mejorar la búsqueda, ya que la parte más importante de este tipo de programación la consituye la manipulación de las restricciones. A modo de resumen, estos lenguajes proveen de una sintaxis para las variables, la representación de las restricciones, el proceso de enumeración y el proceso de optimización, junto a mecanismos de búsqueda y de propagación. Además soportan, en un camino declarativo, la resolución de problemas de búsqueda combinatoria utilizando la división recursiva del problema en subproblemas, hasta que éstos son simples y se pueden resolver fácilmente (Hentenryck et al. 1992).

La parte que más se ha desarrollado es la que afecta a las variables de dominio finito, con aplicaciones como CHIP (*Constraint Handling in Prolog*) o ECLiPSe, y el nuevo paradigma exhibido consiste en restringir y generar ("*constrain and generate*"), en oposición a la estrategia original de enumerar y comprobar ("*enumerate and test)*".

Por otro lado, estos sistemas también permiten resolver problemas de optimización al incluir nuevas restricciones de coste cada vez que se encuentra una mejor solución factible. Además, están provistos de técnicas de resolución de restricciones especializadas para cada uno de los tres dominios de computación que tratan: términos de dominios finitos de las variables, términos booleanos (restricciones simbólicas) para trabajar con variables binarias, y términos racionales con variables y números racionales (resueltos con algoritmos como el símplex, un símplex adaptado según Bisdorff & Laurent 1995). Como exponen Dincbas et al. (1988), parte de la originalidad de estos sistemas como CHIP, reside en que trabajan tanto con restricciones numéricas como simbólicas.

Para poder aplicar CLP un problema, éste debe estar definido por una serie de variables con un dominio finito y determinado, un conjunto de restricciones y una función objetivo a optimizar; además, es necesaria una búsqueda y una estrategia para guiarla. De esta manera, existen tres etapas básicas en la construcción de un programa de restricciones, y una cuarta etapa para la optimización:

1. Determinar el tipo de variables y definir lo que éstas representan.
2. Construir las restricciones con las variables definidas anteriormente.
3. Definir la estrategia de la búsqueda: especificar el orden en que se escogen las variables y cómo se elige y asigna un valor u otro de su dominio.
4. Optimización.

Mientras, el proceso de resolución por medio de un esquema de enumeración *branch and bound* sigue concretamente los pasos siguientes:

1. Definición de las variables, sus dominios y la función objetivo.
2. Selección de un variable del problema y asignación de un valor de su dominio.
3. Uso de técnicas de consistencia para ver las consecuencias de esta asignación (reducción de los dominios de las variables), antes de proceder a una nueva elección y asignación.
4. Comprobación de la existencia de alguna contradicción con los valores seleccionados: si existe contradicción, proceder al retroceso de un paso para reanudar la búsqueda con otros valores que cumplan las restricciones, si no es suficiente, retroceder tantos pasos como sea necesario.
5. Repetición del proceso hasta que toda variable tenga asignado un único valor y todas las restricciones se satisfagan.
6. Continuar la búsqueda hasta la optimalidad, en caso necesario.

Los procedimientos de manipulación de restricciones habitualmente no trabajan con todas ellas, se utilizan unas reglas de inferencia que las activan en puntos apropiados del programa para excluir partes del árbol de búsqueda: reglas asociada a cada restricción que indica bajo qué circunstancias una restricción es tratada. Como exponen Little & Darby-Dowman (1995), principalmente existen dos reglas:

- *Lookahead*: esta regla permite el tratamiento de una restricción cuando una de las variables que contiene ve fijado su valor a uno concreto.

- *Forward checking*: esta regla activa una restricción cuando alguna de las variables que contiene ve reducido su dominio.

Como se puede comprobar, los procedimientos de resolución utilizados son muy semejantes a las técnicas descritas en el apartado 3.8.2.3. como de propagación de restricciones.

Frühwirth et al. (1992) matiza que a veces se puede hacer una ramificación con una asignación de un conjunto de valores en vez de uno sólo: se divide el dominio; además, en procesos de optimización se puede mejorar la eficiencia realizando una búsqueda local cuando se encuentra la primera solución factible (una optimización local), pero en 1992 los sistemas disponibles no incorporaban esta posibilidad. Por su parte, Little & Darby-Dowman (1995) comentan que también es posible utilizar una heurística para encontrar una primera solución factible y luego ejecutar el procedimiento, así como realizar algún preproceso en función de las características del problema a resolver. Bisdorff & Laurent (1995) exponen que estos sistemas permiten hacer búsquedas heurísticas, limitando el tiempo de búsqueda y/o finalizando la búsqueda al encontrar una solución un % mejor que la solución de partida.

Entre los sistemas CLP desarrollados cabe destacar, además de CHIP que fue uno de los primeros, otros sistemas como ECLiPSe, CLP($\Re$), Prolog-III, Trilogy, etc. (Frühwirth et al. 1992). Dincbas et al. (1988) muestran un ejemplo de cómo CHIP utiliza la comprobación de consistencias y la propagación de restricciones: sean dos variables X, Y con dominios {0..9}, y el siguiente sistema de restricciones:

(a) $2 \cdot X + 3 \cdot Y \leq 7$,
(b) $X + Y \geq 4$;

de (a) CHIP deduce que $X \leq 3$ e $Y \leq 2$; estas nuevas restricciones son propagadas a (b) y se obtiene que $X \geq 2$ e $Y \geq 1$; por tanto, el nuevo dominio de X es {2, 3} y el de Y es {1, 2}.

Frente a la resolución de problemas combinatorios mediante el uso de las técnicas de modelización y resolución usualmente utilizadas en programación entera, algunos autores (Jaffar & Maher 1994, Little & Darby-Dowman 1995, Prestwich & Mudambi 1995 y Corominas 1996) exponen diversas diferencias entre éstas y la CLP:

- Modelización del problema. CLP trabaja con lenguajes de alto nivel que facilitan y simplifican la modelización de las características del problema: se proporciona una definición del problema más directa y compacta, con menos variables, con restricciones de alto nivel e incluso con restricciones definidas por el usuario (lo que aporta una gran flexibilidad). Por ejemplo, si la solución debe satisfacer: o la restricción $x \leq 4$ o la restricción $x \geq 6$, se puede modelizar como sigue mediante restricciones lineales (básicas para la programación entera):

$x - M \leq 4 - M \cdot y$
$x \geq 6 + M \cdot y$
$y \in \{0, 1\}$
M entero muy grande;

o se puede representar en CLP únicamente con la variable x: $x \leq 4$; $x \geq 6$ donde el operador ";" representa el operador lógico OR.

- Procedimiento de resolución de los problemas de optimización. Directamente (programación entera) frente a la resolución recursiva de una serie de problemas de satisfactibilidad (CLP).

- CLP presenta técnicas de resolución integradas en un lenguaje host (Prolog es el más usual), mientras que la programación entera, aunque también dispone de lenguajes de modelización, necesita programas que la resuelvan.

De todas formas todo no son ventajas. Prestwich & Mudambi (1995) opinan que para problemas muy estudiados, las técnicas de investigación operativa son las mejores. Darby-Dowman et al. (1997) comentan "*The initial domain reductions available within CLP can act as a pre-processor to IP, but was found to make no difference to the times for solving the IP models. This is not surprising since IP pre-processors adopt similar logical constraint reductions at the root node. However, unlike CLP, commercial IP solvers do not continue to do this at each node in the search tree*", p. 261.

Este último comentario facilita la aseveración expuesta por Corominas (1996), según la cual los procedimientos de resolución utilizados en CLP para la resolución de problemas de optimización combinatoria, pueden formularse como un procedimiento *branch and bound* en el se hace preproceso en todos los vértices del árbol de búsqueda, pero sin utilizar cotas ni podar.

En esta misma dirección, Yan (1998) describe un ejemplo de cómo los programas matemáticos se puede plantear incluyendo alguna regla lógica, además de conservar la resolución mediante procedimientos *branch and bound.* Es lo que Hooker & Osorio (1996) denominan programación lineal mixta lógica (*Mixed logical/linear programming* o MLLP) como una extensión de la programación lineal entera mixta (*Mixed integer/linear programming* o MILP). Un problema modelizado como MLLP consta de un conjunto de fórmulas lógicas que representan los elementos discretos del problema y de una parte de restricciones lineales, con lo que se obtiene una forma más natural de modelizar. Como aclaran Hooker & Osorio (1996), estos programas se resuelven con un *branch and bound* al que se le incluye un elemento clave: antes de ejecutar el procedimiento de acotado (la programación lineal en este caso), se aplican algoritmos de proceso lógico a las fórmulas lógicas, que pueden:

a) proporcionar restricciones lógicas (denominadas cortes lógicos),

b) fijar variables o reducir sus dominios,

c) y descubrir inconsistencias, tras lo cual se realiza *backtracking.*[18]

[18] En Hooker & Osorio (1996) se presenta un *survey* sobre MLLP donde se exponen algunos ejemplos, ventajas e inconvenientes, así como diversas fórmulas lógicas para expresar restricciones discretas y algoritmos de proceso lógico.

Concretamente, en el ejemplo expuesto por Yan (1998) se trabaja con la regla lógica denominada regla de cardinalidad. Sean las variables binarias: y, x_j para j = 1, ..., n, y la regla lógica:

$$y \Rightarrow (x_1 \vee x_2 \vee ... \vee x_n)_k,$$

según la cual si y es verdad, entonces al menos k variables de (x_1, x_2, ..., x_n) son verdad.

Para resolver el programa matemático (basándose en programación lineal relajada si se utilizan procedimientos *branch and bound*), Yan (1998) comenta que se pueden seguir tres estrategias (de las cuales y para el problema que resuelve dicho autor, va mejor la tercera):

1ª. Representar las reglas lógicas mediante restricciones lineales que, si es posible, describan completamente el cierre convexo mediante una serie de cortes denominados cortes lógicos; posteriormente, resolver el problema con *branch and bound.*

Según Yan (1998), las representaciones de las reglas lógicas deben ser lo más ajustadas posibles de forma que definan facetas del cierre convexo. Para la regla lógica comentada, este autor cita el siguiente teorema:

Sea S el conjunto de puntos que satisfacen

$$y \Rightarrow (x_1 \vee x_2 \vee ... \vee x_n)_k$$

y conv(S) el cierre convexo de S.

Suponiendo que n > k o que n = k = 1, las facetas de conv(S) son descritas por:

$$-k \cdot y + (x_1 + x_2 + ... + x_n) \geq 0,$$

y por las restricciones que definen facetas de:

$$y \Rightarrow (x_{j1}, ..., x_{jn-1})_{k-1}$$ [se puede suponer que se refiere $y \Rightarrow (x_{j1} \vee x_{j2} \vee ... \vee x_{jn-1})_{k-1}$]

para todo conjunto $\{j_1, j_2, ..., j_{n-1}\} \subset \{1, 2, ..., n\}$.

De esta manera, el conjunto de restricciones lineales es proporcionado de forma recursiva reduciendo el tamaño de la parte derecha de la regla lógica.

A continuación se expone un ejemplo concreto extraído de Yan (1998):

$$y \Rightarrow (x_1 \vee x_2 \vee x_3 \vee x_4 \vee x_5)_3$$

la representación del cierre convexo viene dada por:

Recursión I: $-3 \cdot y + x_1 + x_2 + x_3 + x_4 + x_5 \geq 0$;
Recursión II: $-2 \cdot y + x_i + x_j + x_k + x_l \geq 0$, $\{i, j, k, l\} \subset \{1, 2, 3, 4, 5\}$;
Recursión III: $-y + x_i + x_j + x_k \geq 0$, $\{i, j, k\} \subset \{1, 2, 3, 4, 5\}$.

Entonces, y como se ha comentado, estas nuevas restricciones se añaden al programa matemático y se resuelve con *branch and bound.*

Esta estrategia requiere un gran número de restricciones pero ajusta mucho el programa lineal, de forma que proporciona una cota de gran calidad.

2ª. Trabajar con restricciones lógicas simbólicas. Como expone Yan (1998), se separan las reglas lógicas del modelo y se utilizan como restricciones simbólicas para podar en la enumeración del *branch and bound.* En el momento de la ramificación se comprueba si las soluciones del vértice violan o violarán las reglas lógicas, y, si es así, se podan.

Esta estrategia reduce la computación del programa lineal relajado, pero aumenta la región de soluciones factibles de dicho programa matemático, con lo que empeora la cota que proporciona.

3ª. Estrategia mixta. Utilizar las reglas lógicas como restricciones simbólicas, pero, además, incorporar al programa matemático la restricción lineal principal obtenida de cada regla lógica, para ajustar la región factible del programa lineal; se resuelve con *branch and bound.* En el ejemplo anterior se implementaría en el programa la restricción lineal obtenida de la primera recursión, además de dejar como restricción simbólica la regla lógica.

En este caso se mejoran las cotas obtenidas del programa lineal sin aumentar mucho su tamaño (Yan 1998).

3.9. Procedimientos heurísticos.

Como definen Gass & Harris (1996), la suboptimización consiste en la búsqueda de una solución para un problema de optimización, mediante un procedimiento que no garantiza que la solución encontrada sea óptima. A estas técnicas, que, como señalan

estos mismos autores, usualmente incluyen reglas heurísticas que ayudan a eliminar la generación de soluciones pobres, se les denomina procedimientos heurísticos. De todas formas, y aunque en ocasiones estas técnicas pueden llegar a asegurar la optimalidad de la solución obtenida cuando el valor de ésta coincide con el valor de una cota de la solución óptima (Gangonells & Gómez 1995, entre muchos otros), se debe tener muy presente que los procedimientos heurísticos no garantizan obtener soluciones factibles a problemas específicos (Mishkoff 1988); por este motivo, a menudo se dice que los métodos heurísticos son impredecibles (Pearl 1984). Según Bjorndal et al. (1995), para valorar la calidad de una solución heurística actualmente se trabaja en obtener condiciones de optimalidad, aunque también hay que tener presente que comprobar la optimalidad de una solución dada puede ser más difícil que resolver el problema.

Ibaraki (1988) expone en su excelente obra sobre la resolución de problemas de optimización combinatoria mediante procedimientos enumerativos, una clasificación de los procedimientos heurísticos. En esta clasificación general se describen todo tipo de técnicas, incluyendo las que son objeto de estudio en este apartado: aquéllas que pueden obtenerse a partir de procedimientos de búsqueda basados en la exploración del espacio de estados:

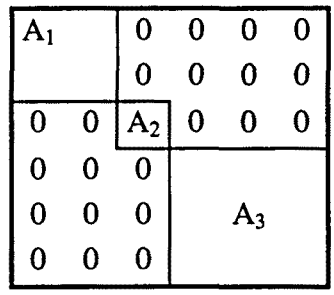
(1) Métodos glotones (*greedy*): son procedimientos que construyen una solución paso a paso con un evaluador local; uno a uno se van seleccionando los candidatos de una lista ordenada. Según Ibaraki (1988), en el problema de la mochila el hecho de ordenar de mayor a menor C_i/a_i proporciona buenos resultados (donde C_i y a_i se corresponden, respectivamente, con el valor y peso del objeto i); por su parte, en el problema del árbol parcial mínimo se obtiene una solución óptima.

(2) Métodos tacaños, mezquinos (*stingy*): estas técnicas parten de tener todos los elementos en la solución y van eliminando los peores hasta obtener una solución factible. En el problema de la mochila se van eliminando los elementos de peor C_i/a_i, hasta tener una solución factible; en el problema de cubrimiento se van eliminando las posiciones más caras hasta tener la última solución factible. En general, las soluciones obtenidas no coinciden con las de los algoritmos *greedy*.

(3) Métodos aleatorios (*random* o de Monte Carlo): consisten en seleccionar soluciones aleatoriamente y quedarse con la mejor; si la construcción de soluciones factibles es fácil, es mejor restringir la búsqueda al espacio de soluciones factibles. En el problema del viajante de comercio cada posible permutación de las ciudades es una solución. De todas formas, estos algoritmos pueden ser muy ineficientes y para mejorarlos se suele intentar dirigir la búsqueda en la dirección de las mejores soluciones.

(4) Métodos de relajación (*relaxation*): se obtienen soluciones, usualmente infactibles, relajando parte de las restricciones originales; posteriormente se van modificando dichas soluciones hasta alcanzar la factibilidad.

(5) Métodos de partición (*partitioning*): consisten en descomponer el dominio de un problema en más de un subdominio y resolver el problema en cada subdominio; posteriormente, las soluciones de los subdominios se agregan en una solución factible del problema original. Un ejemplo es el problema del viajante de comercio y otro el de cubrimiento cuando la matriz es de la forma siguiente:



Matriz de valores de un problema de cubrimiento, de Ibaraki (1988).

(6) Métodos de enumeración parcial (*partial enumeration*) o de reducción del espacio (*space reduction*): cuando no es factible enumerar todos los casos posibles, se pueden enumerar sólo los casos de una pequeña parte del problema y completar el resto de las soluciones de forma heurística; según Ibaraki (1988), esta idea se puede incorporar tanto en procedimientos derivados de la programación dinámica como del *branch and bound.* Para el problema de la mochila se buscan todas las combinaciones posibles de un conjunto de k elementos $\leq$ n (número total de elementos), y aquéllas que no sobrepasan el peso máximo son completadas de forma heurística, conservando la mejor.

(7) Métodos de mejora iterativa (*iterative improvement*) o de búsqueda local (*local search*): técnicas basadas en obtener soluciones aproximadas y repetir la búsqueda entre las vecinas de la mejor solución encontrada. Los algoritmos pueden ser diferentes en función de cómo se definen las vecinas; un algoritmo general de este tipo consiste en generar las soluciones vecinas de la mejor solución en curso: si existe una solución mejor se guarda y se vuelve a iterar, y sino fin.

(8) Métodos de reconocido simulado (*simulated annealing*): cuando el vecindario es grande, una primera modificación de la búsqueda local consiste en explorar aleatoriamente una parte de las vecinas de la mejor solución en curso. Otra modificación consiste en aceptar soluciones peores que la actual con una cierta probabilidad, lo que puede permitir salir de óptimos locales; la probabilidad es $e^{-\Delta/T}$, donde Δ representa el

incremento de la función objetivo y T es una temperatura que va variando de un valor grande a uno pequeño.

(9) Otros métodos: debido a que los algoritmos heurísticos dependen de cómo se explore la estructura del problema, puede haber algoritmos específicos que trabajen muy eficientemente para problemas concretos.

En la literatura que aborda las técnicas de exploración de espacios de estados se han expuesto numerosos procedimientos de búsqueda heurísticos, usualmente como consecuencia de dejar de explorar parte del espacio de soluciones factibles a sabiendas de que en éstos, aunque con menor probabilidad, también se puede encontrar la solución óptima (o, como comenta Sierksma 1996 para los *branch and bound*, al parar el procedimiento antes de finalizar). Se puede pensar que algunos de estos procedimientos heurísticos están basados directamente en técnicas consideradas generales (como el *branch and bound* o algoritmo A*), aunque también existen muchos otros que pueden ser considerados por sí mismos, procedimientos de búsqueda en espacios de estados.

En el apartado 3.1.2. se expone una completa introducción a estas técnicas explicitando diversas formas de obtener procedimientos heurísticos a partir de procedimientos exactos, comentando varios argumentos que para algunos autores justifican el uso de las técnicas heurísticas frente a las exactas y exponiendo su importancia -Ibaraki (1988) y Hoffman & Padberg (1996) consideran que las heurísticas también son importantes ya que pueden proporcionar soluciones en tiempo breve para problemas en los que las técnicas exactas no proporcionan la solución óptima en tiempo razonable, y por que suministran una cota para las técnicas exactas que, como comenta Ibaraki (1988), si es de calidad provoca que el proceso de podado pueda llegar a ser muy potente (Ibaraki 1988, Fischetti et al. 1995, Grötschel & Lovász 1995, Guillén et al. 1995, Hoffman & Padberg 1996, etc.)-. Además, en los apartados anteriores ya se han descrito varios procedimientos de búsqueda que pueden ser considerados heurísticos y se pueden describir muchos otros más, por ejemplo: haciendo terminar a un procedimiento exacto en el instante que obtenga la primera solución factible (o un número deseado de soluciones factibles) e incorporando, si se desea, un procedimiento de optimización local, o haciendo finalizar la búsqueda cuando se haya consumido un tiempo de cálculo prefijado o cuando se hayan explorado un número determinado de vértices, o eliminando la posibilidad de *backtracking* en los diferentes procedimientos que la utilizan; etc.

Otra idea que es interesante tener muy presente es expuesta por Álvarez et al. (1988): los diferentes procesos de selección del vértice, a partir del cual ramificar en cada nivel, permiten diseñar múltiples algoritmos heurísticos a partir de un único procedimiento heurístico basado en la ramificación y exploración del espacio de soluciones factibles;

así, se podría diseñar una heurística tomando como criterio para seleccionar el vértice a expandir aquél de menor cota superior, obtenida ésta mediante una heurística y hasta obtener una solución factible.

A continuación se describe una serie de procedimientos heurísticos que todavía no han sido expuestos completamente en este texto, sin hacer referencia a qué tipo de procedimiento se trata según la clasificación de Ibaraki (1988), sino a sus características concretas como procedimiento de exploración del espacio de estados.

3.9.1. Procedimientos basados en *branch and bound.*

Como ya se ha comentado y Kumar & Kanal (1983b) señalan, es fácil ver que la idea central del *branch and bound* (ramificación y poda) es el corazón de muchas heurísticas de búsqueda en el espacio de estados; así, y como comenta Kumar (1990), muchos procedimientos heurísticos de búsqueda pueden verse como procedimientos derivados de un *branch and bound.* Sean los siguientes ejemplos:

- Ibaraki (1976a) expone que, para un problema de minimización, se puede obtener un algoritmo *branch and bound* heurístico, si se poda un vértice P_i cuando:

$$g(P_i) + \varepsilon(z) \geq \overline{Z},$$

donde $g(P_i)$ es el valor de una cota inferior del vértice P_i, $\varepsilon(z)$ es el valor de una función del error máximo permitido que debe satisfacer $\varepsilon(z) \geq 0$ para todo z, y $z_1 \leq z_2 \rightarrow z_1 - \varepsilon(z_1) \leq z_2 - \varepsilon(z_2)$, y $\overline{Z}$ es el valor de la solución preferible. En este procedimiento heurístico se suelen utilizar dos tipos de funciones $\varepsilon(z)$: por un lado $\varepsilon(z) = \varepsilon$, constante no negativa que garantiza una solución subóptima con un error absoluto $\leq \varepsilon$, y por otro $\varepsilon(z) = r \cdot \varepsilon$, donde r es una constante que satisface $0 \leq r \leq 1$ y que garantiza que la solución óptima esté dentro de un error relativo del r 100% de la solución subóptima final. Este procedimiento definido por Ibaraki (1988) como método de la ε-asignación, también permite utilizar un $\varepsilon(z)$ que varíe en función de los resultados obtenidos y el tiempo empleado, pero siempre cumpliendo que $\varepsilon_1(z) < \varepsilon_2(z) < ...$

- Barr & Feigenbaum (1981) describen el procedimiento heurístico propuesto por Harris (muy semejante al descrito por Ibaraki 1976a), que denomina búsqueda con ancho de banda (*bandwidth search*), y 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, debe cumplir $h(n) \leq h^*(n) + \varepsilon$, y como consecuencia define un algoritmo ε-admisible y encuentra soluciones ε-óptimas.

- Grötschel & Lovász (1995) exponen un procedimiento heurístico basado en la programación lineal: se puede solucionar un programa lineal entero o mixto, resolviendo el programa lineal resultante de no considerar la integridad de las variables, y fijando aquellas variables que son enteras a sus valores y las que no, redondeándolas apropiadamente.

- Hoffman & Padberg (1993) exponen un procedimiento heurístico semejante al anterior para incorporarlo como procedimiento de obtención de soluciones factibles en su procedimiento *branch and cut*. Se basa en la resolución repetitiva del programa lineal resultante de no considerar la integridad de las variables; para ello se intenta dividir el problema en otros más pequeños, redondeando el valor fraccional de un conjunto de variables de valor comprendido entre 0 y 1, y usando las rutinas de preproceso para detectar implicaciones de tales redondeos; posteriormente se lanza el programa lineal y se vuelve a redondear. En cada paso, el programa lineal que se debe resolver disminuye de tamaño (ya que a algunas variables enteras ya se les ha fijado su valor) lo que hace que el algoritmo converja rápidamente. Por otro lado, comentan que el procedimiento *branch and cut* que diseñan también se puede utilizar como un procedimiento heurístico, que, además, proporciona una cota inferior de la solución óptima.

- Johnson et al. (1997) continúa en la tónica de las dos referencias anteriores, exponiendo que se han desarrollado heurísticas efectivas para diversos problemas de *scheduling* basadas en la programación lineal y que recientemente se ha evidenciado que éstas están entre los mejores algoritmos heurísticos. También comentan que las heurísticas de redondeo sucesivo para la resolución de programas lineales enteros o mixtos, a menudo son denominadas heurísticas de inmersión, ya que el redondeo de una variable fraccional puede verse como la ramificación por esa variable y exploración de sólo uno de sus vértices descendientes. Además, matizan que, típicamente, en un algoritmo de redondeo sucesivo para programas lineales entero o mixtos generales, las variables son redondeadas en orden no creciente de sus valores en el programa lineal, aunque desafortunadamente los resultados obtenidos varían enormemente de un ejemplar a otro.

- Como ya se ha comentado en el apartado 3.1.2., Ibaraki (1988) es uno de los autores que exponen, explícitamente, que a partir de procedimientos *branch and bound* se pueden generar procedimientos heurísticos al realizar la búsqueda en un área restringida; concretamente expone tres procedimientos:

1) Método de la ε-asignación (ya definido, sin denominación explícita, en Ibaraki 1976a): procedimiento que consiste, en caso de minimizar, en podar un vértice P_i si $g(P_i) + \varepsilon(z) \geq \overline{Z}$, siendo $g(P_i)$ el valor de una cota inferior del vértice P_i, $\varepsilon(z)$ el valor de una función del error máximo permitido que satisface $\varepsilon(z) \geq 0$ para todo z, y $z_1 \leq z_2 \rightarrow z_1 - \varepsilon(z_1) \leq z_2 - \varepsilon(z_2)$, y $\overline{Z}$ el valor de la solución preferible. Si $\varepsilon(z) = \varepsilon$ (es constante), se

garantiza que la solución obtenida tiene un error $\leq \varepsilon$. Como comenta el mismo autor, un valor alto de $\varepsilon(z)$ relaja mucho este test y reduce el tiempo de búsqueda; también plantea utilizar un $\varepsilon(z)$ que varíe en función de los resultados obtenidos y el tiempo empleado, pero siempre cumpliendo que $\varepsilon_1(z) < \varepsilon_2(z) < ...$

2) Método del T-corte: procedimiento en el que el número de vértices generados está limitado a un valor T; así, cuando se alcanza dicho valor la computación termina, finalizando sin éxito si no se ha obtenido ninguna solución factible.

3) Método del M-corte: técnica en la que el número de vértices activos (vértices generados pero todavía no explorados) está limitado a un valor M. La eliminación de los vértices sobrantes se hace en orden no creciente del valor de una cota o un indicador.

Ibaraki (1988), basándose en su propia experiencia computacional, comenta que comparando los tres procedimientos heurísticos anteriores se pueden proporcionar los siguientes consejos de utilización:

- en el caso de una búsqueda primero el mejor (búsqueda heurística para Ibaraki 1988), si se ordenan los procedimientos heurísticos de mejor a peor se tiene: T-corte, M-corte y ε-asignación;
- en el caso de una búsqueda en profundidad, de mejor a peor son: ε-asignación, M-corte, T-corte;
- parece que cuando el indicador es calidad se debe utilizar el procedimiento T-corte, y cuando no lo es, la técnica ε-asignación;
- el método ε-asignación se puede utilizar para resolver problemas con restricciones duras: primero, y sin incorporar dichas restricciones, se hallan todas las soluciones posibles con ε-asignación; posteriormente se busca si alguna de estas soluciones cumple dichas restricciones.

Ibaraki (1988) también comenta que pueden utilizarse métodos híbridos de los anteriores, y que, empíricamente, se puede decir que ε+T no parece tener ventajas, ε+M puede utilizarse cuando el espacio de memoria no es suficiente para trabajar sólo con ε, T+M parece el más recomendable ya que es el más manejable y controlable (el espacio por M y el tiempo por T), y ε+T+M no parece presentar mejores resultados que los anteriores.

Por último, realiza una caracterización teórica de los algoritmos heurísticos derivados de técnicas *branch and bound*, comentando que ε en general no es monótono, es decir $\varepsilon_1 < \varepsilon_2$ no implica siempre que $T(\varepsilon_1) > T(\varepsilon_2)$ siendo T el número de vértices tratados, y tampoco se puede garantizar que $T(0) \geq T(\varepsilon)$.

- Ibaraki (1988) también describe un procedimiento heurístico derivado de un procedimiento *branch and bound*, en el que se tratan las relaciones de agrupación de vértices (relaciones de dominancias) de forma aproximada. Comenta que el método de la ε-asignación se puede generalizar al test de dominancia, pudiéndose pensar un algoritmo que permita errores en dichas relaciones de dominancia (D): se relaja la condición P_iDP_j (el vértice P_i domina al vértice P_j) $\Rightarrow$ $f(P_i) \leq f(P_j)$ (donde $f(P_i)$ proporciona el valor de la función objetivo de P_i), por $P_iDP_j \Rightarrow f(P_i) - v \leq f(P_j)$, con v = cte. ≥ 0.

3.9.2. Procedimientos basados en A*.

Debido a la gran importancia que ha tenido el procedimiento A* en el área de la inteligencia artificial, algunos autores le otorgan un papel semejante al que ostenta el *branch and bound* en el área de la investigación operativa, en el sentido de considerar al A* como un procedimiento general que ha inspirado diversas heurísticas de búsqueda en el espacio de estados; así, algunas técnicas de búsqueda heurística pueden verse como procedimientos derivados de un A*. Sean los ejemplos siguientes:

- Pearl (1984) comenta que, como lo expuesto en el apartado anterior para el *branch and bound* por Ibaraki (1976a) y Barr & Feigenbaum (1981), se puede pensar en aproximaciones ε-admisibles del algoritmo A*: se puede definir el algoritmo A_ε*, considerando que se pueden admitir soluciones que no estén más lejos del coste de la solución óptima que en un factor $(1 + \varepsilon)$; como consecuencia, el algoritmo no termina con una solución cuyo coste sea más grande que $(1 + \varepsilon) \cdot \bar{Z}$ y naturalmente $\varepsilon = 0$ reduce A_ε* a A*.

- Cortés et al. (1993) definen una propiedad del A*, que denominan degradación gradual, y que, según sus autores, en general se cumple para cualquier algoritmo de búsqueda heurística; la idea consiste en que si el valor de la función de evaluación f(n) rara vez sobrestima el valor real de la distancia recorrida y de la que falta por recorrer en más de un factor α, entonces A* encontrará una solución con un coste que será como máximo K unidades mayor que la solución óptima. Como se puede comprobar es una idea semejante a la expuesta por Pearl (1984).

- En la misma línea argumental, Barr & Feigenbaum (1981) comentan que puede ser mejor encontrar una solución de valor razonable en un tiempo moderado, antes que la óptima con el A*; posiblemente, se refieren al hecho de preferir una función de evaluación que evalúe más ajustadamente aunque a veces sobrestime, con lo que se obtendría un algoritmo que no satisface la condición de admisibilidad.

- Ginsberg (1993) expone el algoritmo A* en tiempo real *(real time A** o RTA*), como un procedimiento A* con un horizonte de búsqueda limitado: se ejecuta una acción cada k segundos, aunque no sea la óptima (siendo k función del horizonte de búsqueda), como resultado de la ejecución de un algoritmo A* con un horizonte de búsqueda limitado. De todas formas este procedimiento no entra en el estudio que se realiza en este texto ya que no cumple las hipótesis de trabajo, al tratarse de problemas de solución on-line en un entorno cambiante (sistemas autónomos en tiempo real, etc.).

3.9.3. Procedimientos basados en programación dinámica: programación dinámica restringida (*restricted dynamic programming*).

Malandraki & Dial (1996) y Hribas & Daskin (1997) describen un procedimiento heurístico de búsqueda basado en programación dinámica, denominado, según Malandraki & Dial (1996), programación dinámica restringida (*restricted dynamic programming*). El procedimiento, muy parecido al algoritmo de búsqueda en haz de amplitud n que se describe en el apartado 3.9.4.2., se diseña con el doble objetivo de evitar la explosión combinatoria típica de la programación dinámica y controlar las necesidades de memoria, a expensas, eso sí, de poder garantizar siempre la optimalidad de la solución obtenida.

El procedimiento es muy sencillo y consiste en restringir el espacio de búsqueda, reteniendo en cada nivel de la arborescencia, únicamente y como máximo, los H estados más prometedores. De esta forma, en esta técnica únicamente se utiliza el podado por dominancias, no se garantizan soluciones óptimas y tampoco se proveen cotas del valor de la solución óptima.

Como señalan Malandraki & Dial (1996) y Hribas & Daskin (1997), si H es infinito se trata de la programación dinámica, y si es igual a 1 se convierte en una heurística *greedy* un paso: en cada etapa se selecciona la mejor solución parcial que se puede obtener a partir del estado actual.

3.9.4. Otros procedimientos.

Existe un conjunto de técnicas heurísticas que, debido a la forma de presentarse en la literatura, pueden considerarse procedimientos heurísticos más generales o son más difíciles de clasifcar; de todas formas, algunas de éstas se pueden llegar a pensar como descendientes de un procedimiento *branch and bound* o de un algoritmo A*.

3.9.4.1. Estrategias de escalada (*hill climbing* y *steepest ascent*).

Las estrategias de escalada o también denominadas de ascenso, son uno de los tipos de procedimientos heurísticos más sencillos y comentados en la literatura, y, como ya ocurría en el caso del procedimiento A* (apartado 3.6.8.), existen algunas discordancias entre las definiciones presentadas en la literatura referenciada: existen autores que definen y diferencian entre los algoritmos de ascenso (*hill climbing*) y de ascenso por la máxima pendiente (*steepest ascent*), mientras que otros sólo consideran el algoritmo de ascenso (*hill climbing*) o consideran que ambos son iguales; también están los que denominan al algoritmo de ascenso por la máxima pendiente como *steepest ascent hill climbing*. De nuevo se pone de manifiesto la poca precisión de algunas definiciones que se han expuesto para las técnicas de exploración de espacios de estados.

Los procedimientos que incorporan la estrategia de escalada han sido definidos por diversos autores (Pearl 1984, Mompín et al. 1987, Shirai & Tsujii 1987, Korf 1990, Cortés et al. 1993, Ginsberg 1993, Rich & Knight 1994 y Winston 1994a, entre otros), y se refieren a la estrategia de seleccionar como próximo vértice a explorar uno de entre los hijos del vértice que acaba de ser expandido; se trata, por tanto, de una búsqueda en profundidad (*depth first search*) pero sin la posibilidad de realizar *backtracking* y con una función de evaluación f(n) que incorpora alguna medición heurística de la distancia que queda por recorrer (no necesariamente en número de pasos). Shirai & Tsujii (1987) consideran que la función de evaluación debe expresar el valor inferido del coste de alcanzar un vértice considerado objetivo desde el vértice n.

Concretamente, Pearl (1984), Mompín et al. (1987), Shirai & Tsujii (1987), Korf (1990), Ginsberg (1993) y Winston (1994a) consideran que esta estrategia consiste en generar todos los sucesores del vértice expandido y entre éstos seleccionar para la nueva expansión aquél de mejor valor de la función f(n); es decir, se restringen los vértices candidatos a ser expandidos entre los descendientes del vértice actual. Este procedimiento es denominado habitualmente *hill climbing*, aunque Ginsberg (1993) lo denomina indistintamente *hill climbing* y *steepest ascent.*

Rich & Knight (1994) señalan dos posibilidades para la estrategia de escalada: por una lado definen la escalada simple (*hill climbing*), que consiste en generar un descendiente del vértice actual y si es mejor seleccionarlo como estado a expandir, o, en caso contrario, en generar un nuevo descendiente; y por otro lado definen la escalada por la máxima pendiente (*steepest ascent hill climbing*) o búsqueda del gradiente (*gradient search*), en la que se consideran todos los descendientes del estado actual y se selecciona el mejor de ellos como nuevo estado a expandir.

Por su parte, Cortés et al. (1993) también definen, aunque parece que de forma diferente, dichas estrategias; así, la estrategia *hill climbing* consiste en seleccionar como nuevo vértice a expandir un vértice mejor al actual o ninguno (según una f(n) que proporciona una cierta estimación de lo prometedor que puede llegar a ser un camino), mientras que la estrategia *steepest ascent* selecciona para expandir el vértice con mejor función de evaluación f(n) (se supone que en ambos casos la elección se realiza entre los descendientes del vértice que se acaba de expandir).

Álvarez et al. (1988) presentan un algoritmo heurístico basado en una estructura de ramificación y acotación sin retroceso, en el que en cada nivel de ramificación se evalúa la capacidad de cada vértice (según estos autores típicamente mediante el cálculo de una cota inferior) y se continúa el proceso a partir del vértice más prometedor; se elige un único camino de bajada hasta completar una solución factible (ya que, en el problema que resuelven, cualquier secuencia que cumpla unos criterios considerados en el momento de la ramificación es válida); como se puede comprobar aunque no lo expliciten sus autores, es un procedimiento heurístico de ascenso. Por su parte, Kusiak et al. (1993) resuelven un problema de tecnología de grupos mediante una heurística basada en *branch and bound* que responde también a los procedimientos descritos en este apartado; y Fuentes et al. (1997) resuelven un problema de navegación de un robot autónomo en tiempo real para alcanzar una posición objetivo utilizando una estrategia *hill climbing*.

Como comentan varios autores, Pearl (1984), Shirai & Tsujii (1987), Korf (1990), Cortés et al. (1993), Rich & Knight (1994) y Winston (1994a) entre otros, los procedimientos descritos pueden presentar varios problemas y desventajas:

1. Usualmente sólo se alcanzan estados que constituyen óptimos locales, aunque también pueden corresponder a soluciones parciales no factibles.
2. Cuando se alcanza un estado que es un óptimo local o una solución no factible, no se puede salir de él y explorar otras regiones del espacio de soluciones.
3. Problema de la meseta: cuando todas las expansiones y las sucesivas en un horizonte de uno o dos niveles tienen el mismo valor, la mejor expansión no puede determinarse únicamente con información local ya que todos los estados presentan la misma medida de evaluación.
4. Problema de la cresta: el área de estados en la que se encuentra el vértice es un conjunto de óptimos locales entre los que existe una cierta pendiente, pero es imposible remontarnos mediante los operadores disponibles.

Por tanto, estos procedimientos presentan la incapacidad de decidir la mejor trayectoria global con criterios estrictamente locales.

Como ya se ha comentado, la escalada no es siempre eficaz ya que los procedimientos descritos pueden no encontrar la solución óptima o incluso no encontrar ninguna solución factible; para evitar este fracaso Rich & Knight (1994) y Winston (1994a) proponen diferentes posibilidades:

- volver hacia atrás en algún vértice anterior e intentar seguir un camino diferente,
- realizar un gran salto en alguna dirección para intentar buscar en una nueva parte del espacio de búsqueda,
- aplicar dos o más reglas antes de realizar la evaluación,
- incluir un ápice de búsqueda no determinista.

3.9.4.2. Búsqueda en haz de amplitud n (*beam search*).

Varios autores, Mompín et al. (1987), Bisiani (1990), Korf (1990), Shih et al. (1992), Cortés et al. (1993), Winston (1994a), etc., han definido el algoritmo de búsqueda en haz de amplitud n (*beam search*), también llamado estrategia selectiva de amplitud n por Cortés et al. (1993).

Este procedimiento consiste en una técnica heurística de búsqueda, que según Cortés et al. (1993) se puede considerar una variante de la búsqueda en anchura, que en cada nivel, y para la siguiente expansión, sólo retiene los n mejores vértices encontrados en ese nivel; es decir, se examina un número determinado de alternativas (el haz) descartándose para siempre todas las demás. Se trata de una técnica heurística ya que se utilizan reglas heurísticas para descartar alternativas poco prometedoras, intentando mantener el tamaño del haz tan pequeño como sea posible; en consecuencia, y como especifica Winston (1994a), el número de vértices explorados se mantiene siempre manejable.

Como señala Bisiani (1990), si el haz es infinito se trata de una búsqueda en anchura, y si no lo es, no garantiza la solución óptima. Cortés et al. (1993) exponen una posible dificultad de este procedimiento: tiene el inconveniente de que se debe seleccionar un valor adecuado de n (tarea para la que no existen reglas generales); mientras, Mompín et al. (1987) proponen un valor dinámico de n: en zonas donde existe un camino muy prometedor respecto a sus competidores la expansión de alternativas debe ser pequeña, mientras que en caso contrario la "anchura del haz" debe aumentar.

Este procedimiento de búsqueda admite claramente el cálculo paralelo: aunque parece claro que si se paraleliza se debe paralelizar la expansión y la evaluación de los vértices, esto provoca demasiado overhead (se supone que de comunicación); para solucionar este problema se han diseñado arquitecturas especiales para este tipo de búsqueda (Bisiani 1990).

Shih et al. (1992) definen un procedimiento heurístico para la resolución de un problema de distribución en planta (un problema de ordenación en el que cualquier orden proporciona una solución factible), que consiste en un algoritmo de búsqueda en haz de amplitud n con un mayor grado de elaboración. En primer lugar se definen tres valores: la amplitud del haz, n, la anchura de filtro, l, y la profundidad de filtro, d. El procedimiento consiste en una búsqueda en anchura en la que en cada nivel y para la siguiente expansión, sólo se retienen los n mejores vértices de dicho nivel descartándose todos las demás. Para decidir cuáles son los n mejores vértices de cada nivel se realiza el siguiente proceso: se seleccionan los l mejores vértices del nivel en curso según la misma función de selección que en el procedimiento *beam search* original, se realiza una búsqueda en profundidad con éstos, y a las soluciones factibles encontradas se les aplica d iteraciones de un procedimiento de mejora basado en un 2-intercambios. Los valores de las soluciones mejoradas proporcionan un muy buen indicador de la calidad potencial de los l vértices preseleccionados; de esta manera, entre los l vértices se seleccionan n con los mejores valores de dicho indicador (Shih et al. 1992).

3.9.4.3. *Fix and relax* (F&R).

Dillenberger et al. (1994) exponen un nuevo procedimiento heurístico que denominan *fix and relax*, como una técnica de exploración del espacio de soluciones factibles basada en un esquema del tipo *branch and bound*, que utiliza como cotas inferiores las obtenidas de la resolución del programa matemático resultante de la relajación de las restricciones de integridad de las variables. De todas formas, la idea principal en *fix and relax* consiste en no dejar de considerar la integridad de las variables todas a la vez, sino sucesivamente: va considerando iterativamente la integridad de subconjuntos de variables.

Brevemente, el procedimiento *fix and relax* se puede resumir como sigue:

Paso 1. Decidir el subconjunto de variables de valor todavía no fijado dentro de las variables enteras, que deben tomar valor entero en esta iteración.

Paso 2. Resolver el programa lineal mixto (PLM) obtenido en el paso 1 mediante un procedimiento de optimización (que los autores no especifican, aunque parece que se refieren a un paquete standard de optimización), con lo que se obtiene un valor entero para el subconjunto de variables que deben ser enteras en esta iteración y una cota del valor de la solución óptima. Fijar el valor de dicho subconjunto de variables enteras e ir a paso 1.

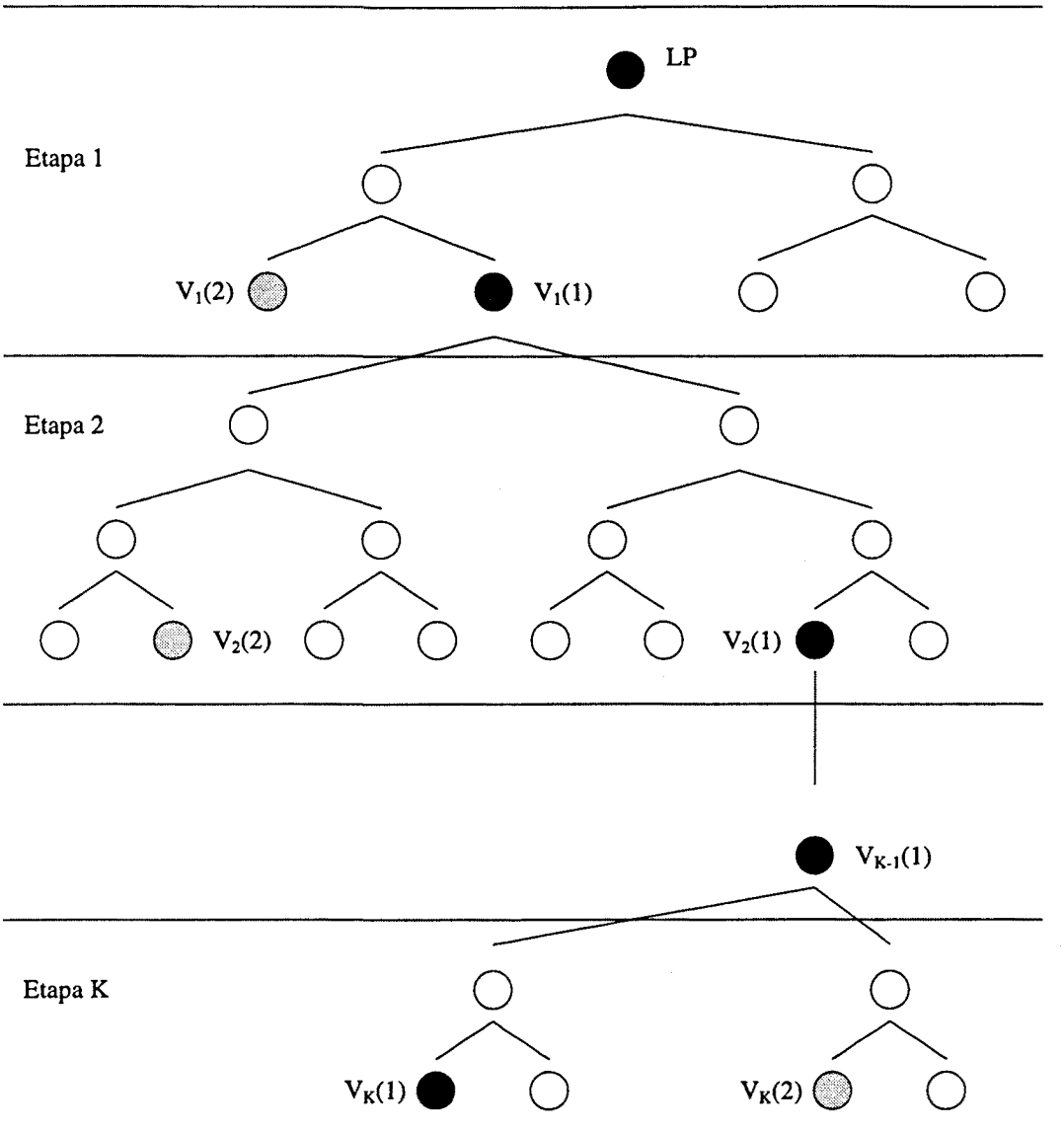
Dillenberger et al. (1994) aplican *fix and relax* para la resolución de un problema de planificación de la producción mediante el planteo y la resolución de un PLM. El problema consiste en planificar la producción en un horizonte de T periodos,

definiéndose para cada periodo $t \leq T$ un conjunto de variables binarias (por ejemplo, X_{igt} = 1 si una parte de los trabajos de tipo i pueden ser asignados a la máquina g en el periodo t, y 0 en caso contrario). Considerando cada periodo desde 1 hasta T como una etapa, se propone el siguiente proceso iterativo:

- En la etapa 1 se resuelve un PLM relajando todas las restricciones de integridad de las variables excepto aquellas definidas expresamente para la etapa 1. Se obtiene el valor óptimo de estas variables enteras para este subproblema y se fijan adoptando dichos valores enteros.

- En la etapa t se resuelve un PLM relajando todas las restricciones de integridad de las variables, excepto aquellas definidas expresamente para la etapa t y excepto aquellas que pertenecen a las etapas anteriores y que, por tanto, ya han sido fijadas. Se obtiene el valor óptimo de estas variables enteras para este subproblema y se fijan adoptando dichos valores enteros.

El procedimiento *fix and relax* puede ser estructurado en un esquema de *branch and bound*, cuyo funcionamiento es el siguiente. Para el primer vértice se resuelve un programa lineal (PL) donde se relajan todas las restricciones de integridad de las variables. Una etapa se refiere a un periodo dentro del esquema multiperiodo. En cada etapa existe una serie de variables que deben tomar valor entero (variables 0-1 y posiblemente variables enteras acotas superior e inferiormente), así se realiza una ramificación completa fijando el valor de dichas variables, y para cada vértice terminal de dicha etapa se resuelve un PL. $V_t(1)$ y $V_t(2)$ son los vértices con el mejor y el segundo mejor valor respectivamente del PL asociado al subproblema de la etapa t. Sea K la última etapa correspondiente al último periodo y $V_K(1)$ el vértice con mejor valor del PL del periodo K, dicho vértice proporciona la solución heurística para el problema original. En el caso de que el subproblema en un estado t sea infactible, se realiza un *backtracking* reemplazando el vértice $V_{t-1}(1)$ por el vértice $V_{t-1}(2)$ y tomando éste como el vértice inicial en la etapa t. Si únicamente se retienen los dos mejores vértices de cada etapa, como parece indicar el procedimiento, puede llegarse al vértice inicial sin haber obtenido ninguna solución factible y existiendo todavía numerosos vértices por explorar.



Evolución de la exploración según el procedimiento *fix and relax*, de Dillenberger et al. (1994).

Dillenberger et al. (1994) comentan que *fix and relax* es un procedimiento que aprovecha la naturaleza multiperiodo del problema entero para encontrar soluciones subóptimas; de todas formas, parece que el esquema *fix and relax* también podría utilizarse en problemas que no fueran multiperiodo, aunque los dos problemas principales serían ¿cómo establecer los grupos de variables y en qué orden hacer enteras dichas variables?.

Fix and relax también ha sido utilizada por Escudero et al. (1995b) para la resolución de un programa matemático que modeliza la determinación de un programa de inversiones, obteniéndose soluciones factible cercanas a la óptima y que constituyen una cota inferior de ésta; y por Plans & Corominas (1998) para la resolución de problemas de líneas de montaje.

3.9.4.4. Procedimientos heurísticos de mejora.

Como comentan Bjorndal et al. (1995) para los problemas de *scheduling* de máquinas, las heurísticas se pueden dividir en dos grandes grupos:

a) Por un lado están las heurísticas constructivas que, como su propio nombre indica, ser basan en ir construyendo la solución mediante reglas "sensatas", pero heurísticas; a este grupo de heurísticas pertenecen la mayoría de procedimientos descritos anteriormente.

b) Por otro lado, existe otro gran grupo de procedimientos heurísticos, los denominados de mejora, que a partir de una solución, factible o no, son capaces de obtener una solución heurística factible, en numerosas ocasiones; para ello utilizan técnicas denominadas algoritmos de búsqueda local o procedimientos de exploración de entornos como la búsqueda tabú, el recocido simulado, los algoritmos genéticos, los k-intercambios, etc. Bjorndal et al. (1995) señalan que debido a que estas heurísticas tienen una estructura muy similar se les ha denominado meta-heurísticas; éstas proporcionan algoritmos con una complejidad que puede ser definida por el usuario debido a la gran flexibilidad y grado de decisión que permiten, además se basan en búsqueda local y no requieren de mucho conocimiento para generar buenas soluciones.

No es difícil imaginar la descripción de una técnica heurística constructiva a la que se le incorpore un procedimiento de mejora (en Mayor & Ruiz 1995, se comenta que las técnicas de búsqueda local permiten obtener soluciones aproximadas a problemas de optimización en condiciones muy generales, especialmente cuando el conjunto factible es numerable, lo que incluye problemas de optimización combinatoria). Este tercer grupo de procedimientos, híbrido entre los dos anteriores, se está desarrollando actualmente y también en el campo de los procedimientos de búsqueda en espacios de estados.

Pastor & Corominas (1998), después de obtener una primera solución pseudofactible en un problema industrial de equilibrado de líneas de montaje, mediante una exploración dirigida en el espacio de estados del problema, mejora su función objetivo, o al menos intenta aumentar su grado de factibilidad, mediante una búsqueda tabú.

Como ya se ha expuesto en el apartado 3.9.4.2., Shih et al. (1992) aplican un procedimiento basado en la búsqueda en haz de amplitud n, en la resolución de un problema de distribución en planta; una parte importante y definitoria de dicha técnica, consiste en la aplicación de un procedimiento de optimización local en un conjunto de vértices terminales.

Fischetti et al. (1995), buscando buenas soluciones iniciales factibles para utilizar en un algoritmo *branch and cut*, proponen un procedimiento heurístico que consta de las dos etapas anteriores: obtener una solución factible a partir de la última solución parcial y aplicar procedimientos de refinado para obtener una solución quasi-óptima (se supone que mediante búsquedas u optimizaciones locales). Por su parte Savelsbergh (1997), también buscando buenas soluciones iniciales factibles para resolver un problema de asignación generalizada mediante un algoritmo *branch and price*, propone un algoritmo heurístico al que incorpora un procedimiento de optimización local.

Guillén et al. (1995) construyen un algoritmo heurístico que se puede utilizar como tal o que se puede implementar en un algoritmo general de *branch and bound*, y que implementa la idea de que soluciones cercanas a la óptima deben tener una estructura similar a la óptima (¿optimización local en los vértices de mejor cota o indicador?).

Nagar et al. (1996) resuelven un problema bicriterio de secuenciación *flow-shop* en 2 máquinas, utilizando una meta-heurística basada en algoritmos genéticos, pero que también incorpora un procedimiento *branch and bound* para generar información que es utilizada posteriormente para guiar la búsqueda del algoritmo genético. A grandes rasgos, las ideas básicas que exponen el funcionamiento de la meta-heurística son las siguientes: (1) el *branch and bound*, utilizando una estrategia *depth first*, explora el espacio de estados hasta que se alcanza cierta profundidad de corte en el árbol; (2) durante el procedimiento, se comparan las cotas inferiores halladas para cada vértice con una cota superior inicial (hallada con una heurística que suele ser bastante buena y que permite podar un gran número de vértices) y se podan los vértices cuya cota inferior es mayor o igual que el valor de la cota superior; (3) si se puede podar un vértice, cualquier secuencia completa obtenida al completar la secuencia parcial de dicho vértice sólo puede producir soluciones subóptimas: 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 una cota superior conocida, 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), ...; (4) utilizando dicha información, el algoritmo genético prohibe la búsqueda en las regiones subóptimas que ya conoce. Se puede comprobar que la eficiencia del algoritmo genético en la meta-heurística depende de la información suministrada por el *branch and bound*, que a su vez depende del nivel del vértice de donde procede; por otro lado, como un incremento en la eficiencia del algoritmo genético implica un mayor coste computacional del *branch and bound*, se necesita determinar cuidadosamente el nivel máximo de búsqueda de éste.

3.9.4.5. Procedimientos heurísticos en inteligencia artificial.

Como especifica Ibaraki (1988), en el área de la inteligencia artificial más que optimizar se busca encontrar un vértice concreto dentro de un árbol, pero sin considerar una

función objetivo, ni un test de cota para podar el espacio de estados, ni relaciones de dominancia para agrupar estados. Dado un árbol, un vértice raíz s y un conjunto de vértices objetivo, se busca obtener un camino desde s hasta uno de los vértices objetivos de la forma más eficiente posible.

Para resolver estos problemas, Ibaraki (1988) comenta que se suele usar el *heuristic path algorithm* o *tree-search algorithm* (HPA) que se basa en seleccionar como vértice a expandir aquél de mejor valor de h(n), una estimación del camino desde s hasta un vértice objetivo pasando por n. Las fases de este procedimiento son las siguientes:

Fase 1. Inicialización.

Incluir el vértice raíz s en el conjunto de vértices activos.

Fase 2. Búsqueda.

Si el conjunto de vértices activos = ∅, entonces el problema no tiene solución y fin, sino seleccionar para ramificar el vértice n de mejor valor de h.

Fase 3. Test de fin.

Si n es un vértice objetivo, entonces fin y éste proporciona la solución de s, sino ir a fase 4.

Fase 4. Ramificación.

Ramificar totalmente el vértice n seleccionado, incorporar sus descendientes a la lista de vértices activos y eliminar n de dicha lista; ir a fase 2.

Como para la mayoría de procedimientos, la eficiencia de HPA viene determinada por la calidad de h. Una forma típica de h(n) es la siguiente:

$$h(n) = d(n) + e(n),$$

donde:

d(n) = profundidad de n,

e(n) = estimación de e*(n),

e*(n) = distancia desde n hasta un vértice objetivo, normalmente el número de arcos.

Ibaraki (1988) también comenta que una forma más general de h(n) es: $h(n) = (1-w) \cdot d(n) + w \cdot e(n)$ siendo w un peso que satisface $0 \leq w \leq 1$.

Los dos procedimientos de búsqueda descritos anteriormente ya han sido definidos en el apartado 3.2. aunque no específicamente como procedimientos heurísticos. Para la mayoría de autores estos procedimientos enumerativos presentan un carácter general que les permite concebir tanto técnicas de búsqueda exactas como heurísticas, en función de las condiciones, características y funciones de evaluación particulares consideradas.

4. PROCEDIMIENTOS DE EXPLORACIÓN EN PARALELO.

La ejecución de programas en paralelo y, en particular, los algoritmos paralelos de búsqueda en grafos de estados, es un tema actualmente en investigación y desarrollo que ha ido variando rápida y continuamente a lo largo de los últimos años y que, sin duda, lo continuará haciendo. La paralelización es un área de investigación con un gran potencial e interés de desarrollo (existen gobiernos incentivando su estudio e implantación práctica -por ejemplo el programa PACOS, dentro del ESPRIT III-), en el que se está comenzando a diseñar hardware *adhoc* e intentando llegar a compromisos para desarrollar modelos de programación, algoritmos y software, apropiados para ordenadores paralelos.

Como puede comprobarse en la literatura existente, el paralelismo es una área de investigación en la que se presentan continuos problemas, soluciones, avances y cambios, por lo que en esta tesis únicamente se realiza una introducción muy breve a los conceptos más importantes implicados en el paralelismo (apartado 4.1.); se comenta cómo se están comenzando a utilizar estas herramientas de hardware y software en la resolución de problemas de optimización combinatoria mediante procedimientos de resolución enumerativos (apartado 4.2.); y se expone cómo se están desarrollando y distribuyendo gratuitamente librerías que permiten ejecutar procedimientos del tipo *branch and bound* en paralelo, viendo la gran dificultad actual de desarrollar compiladores que transformen, de manera eficiente, código secuencial en un programa que se pueda ejecutar en una arquitectura en paralelo.

4.1. Paralelismo.

Según exponen Xu et al. (1995) y Lusa et al. (1997), la paralelización o computación paralela consiste en hacer trabajar a la vez y de manera concurrente sobre un mismo problema (o sobre una tarea simple según Eckstein 1996), a varios ordenadores (sistema denominado multicomputador) o a un ordenador con varios procesadores (sistema denominado multiprocesador): se dividen las tareas en subtareas y se asignan dichas subtareas a los diferentes procesadores que las realizan a la vez. Por tanto, y como exponen estos mismos autores, los objetivos de la paralelización consisten en acelerar la ejecución de una aplicación (*speed-up*) -por ejemplo, para resolver ejemplares de problemas de mayor tamaño (Gendron & Crainic 1994)- y/o ejecutar más aplicaciones por unidad de tiempo (*throughput*).

Como afirman otros investigadores, de Miguel & Angulo (1991), Kumar et al. (1994) y Zenios (1994) entre otros, la paralelización parece necesaria (e imprescindible en

algunas disciplinas: predicción del tiempo, control del tráfico aéreo, etc.) principalmente por dos motivos:

a) por un lado, los procesadores actuales más rápidos están trabajando a velocidades cercanas a los límites establecidos por las leyes de la física, la velocidad de la luz, por lo que para aumentar la velocidad de cálculo se puede pensar en unir el trabajo de varios de ellos;

b) y por otra parte existen factores de coste: el aumento de potencia del ordenador deja de ligarse al coste del mismo mediante una función lineal, en el caso de usar elementos baratos utilizados de forma repetitiva; además, el desarrollo de un nuevo procesador tiene un coste muy elevado.

Grötschel & Lovász (1995) son más tajantes y ya en 1995 opinan que el paralelismo es un aspecto muy importante a considerar en optimización combinatoria, ya que las máquinas del futuro serán paralelas.

Kindervater & Lenstra (1986), Mahanti & Daniels (1993), Gendron & Crainic (1994), Eckstein (1996), Gass & Harris (1996) y Roucairol (1996), consideran que las principales clases de arquitecturas de ordenadores en paralelo con las que se trabaja actualmente son las dos siguientes (clasificación extraída, según Rafiquzzaman & Chandra 1990 y de Miguel & Angulo 1991, de la realizada por Flynn en 1966[19]):

a) SIMD *(Single Instruction, Multiple Data)*: un único flujo de instrucciones controla todos los procesadores; éstos, de forma síncrona, ejecutan los mismos cálculos con diferentes datos, por lo que resulta una organización excelente en aplicaciones de tratamiento de matrices y vectores (Rafiquzzaman & Chandra 1990).

b) MIMD *(Multiple Instruction, Multiple Data)*: cada procesador ejecuta su propio flujo de instrucciones, posiblemente diferente del flujo de instrucciones de otros procesadores; de esta manera, existen varios flujos de instrucciones diferentes, concurrentes y asíncronos.

De todas formas no se debe olvidar la arquitectura SISD *(Single Instruction, Single Data)*, que se corresponde con la arquitectura tradicional de ordenadores secuenciales: una única instrucción es ejecutada en un único dato.

También hay quiénes, como Lusa et al. (1997), distinguen entre otros tipos de estructuras de ordenadores en paralelo:

[19] Para un mayor detalle se recomienda consultar Flynn (1966).

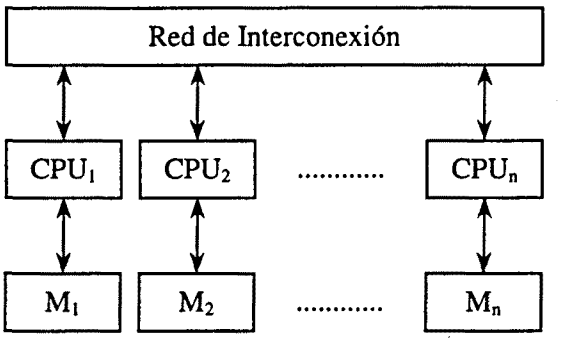
a) Multiprocesadores vectoriales *(Multiprocessor-Vectorcomputers)*: ordenador con varios procesadores vectoriales de altas prestaciones (hasta 16 unidades de proceso), adecuados para operaciones con vectores.

b) Sistemas masivamente paralelos *(Masively Parallel Processors Systems)*: ordenador con varios procesadores standard (hasta cientos o miles).

c) Estaciones de trabajo múltiples *(Multi-Workstations)*: diversos ordenadores o estaciones de trabajo conectadas a través de una red de interconexión, lo que ofrece una gran flexibilidad: los ordenadores pueden ser diferentes, y unos más potentes y paralelos que otros.

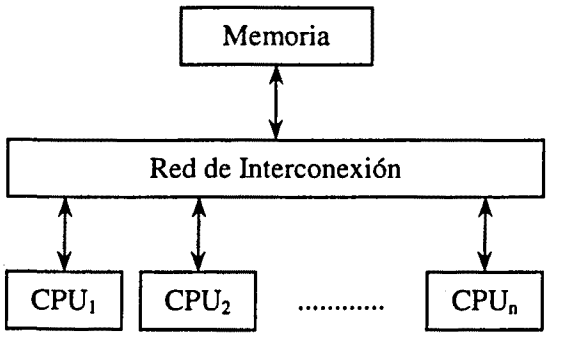
La selección del tipo de memoria a utilizar es otra dicotomía importante a resolver (Zenios 1994, Valero 1995, Eckstein 1996, Lusa et al. 1997, etc.):

a) Memoria local, usualmente denominada distribuida: cada procesador dispone de su propia memoria local y la información únicamente puede ser intercambiada con paso de mensajes a través de una red de interconexión.



Interconexiones en un sistema paralelo con memoria local (distribuida), de Lusa et al. (1997).

b) Memoria global, habitualmente denominada compartida: existe una única memoria global, cuyas posiciones son igualmente accesibles para todos los procesadores a través de la red de interconexión.



Interconexiones en un sistema paralelo con memoria global (compartida), de Lusa et al. (1997).

De todas maneras, y como señala Eckstein (1996), usualmente se utilizan y mezclan todas las clasificaciones, y el uso de uno u otro modelo depende de la velocidad relativa de la comunicación versus la computación.

Si se diseña un código utilizando la programación en paralelo, hay que tener presentes un conjunto de aspectos básicos, entre los que destacan los cuatro siguientes (Gendron & Crainic 1994, Kumar et al. 1994 y Lusa et al. 1997):

1) ¿Cómo realizar la descomposición en tareas?

La paralelización de un algoritmo implica su descomposición en tareas, de forma que cada una de estas tareas indivisibles se pueda ejecutar, si llega a ser necesario, en procesadores diferentes. Esta descomposición se realiza en función de la granularidad deseada de las tareas: tamaño del trabajo que se envía a un procesador, independientemente de los otros, entre dos sincronizaciones; de esta manera se realiza la siguiente distinción:

- Granularidad fina (paralelismo de bajo nivel, según Bruin et al. 1988): operaciones pequeñas a nivel de bit, con las que se obtiene un mayor nivel de paralelismo aunque a costa de una mayor necesidad de comunicación.

- Granularidad gruesa (paralelismo de alto nivel, según Bruin et al. 1988): operaciones a nivel de función, con las que se obtiene un menor nivel de paralelismo aunque requiere una menor necesidad de comunicación.

Valero (1995) propone un mayor grado de precisión y realiza una clasificación más detallada en la que diferencia entre cuatro clases de granularidad:

- Paralelismo de grano grueso: descomposición en procedimientos y funciones.

- Paralelismo de grano medio: descomposición en bucles.

- Paralelismo de grano fino: descomposición en sentencias.

- Paralelismo de grano muy fino: descomposición en opciones dentro de una sentencia.

2) ¿Cuándo realizar la descomposición en tareas?

La descomposición de un algoritmo en las tareas a ejecutar, se puede hacer de forma estática o dinámica:

- Descomposición estática: la descomposición la decide el programador o el compilador, en el momento de realizar el código paralelo.

- Descomposición dinámica: se diseña un programa de forma que la descomposición se decide en el instante de la ejecución.

3) ¿Cuándo y dónde ejecutar las tareas?

El instante y el procesador en el que ejecutar las tareas a realizar, también se puede decidir de forma estática o dinámica:

- Planificación o equilibrado estático de la carga: el orden, así como las tareas que realizará cada procesador, se decide en el momento de la compilación o al inicio de la ejecución; según Lusa et al. (1997), este equilibrio de la carga puede no ser bueno al existir tiempos muertos.

- Planificación o equilibrado dinámico de la carga: el orden, así como las tareas que realizará cada procesador, se decide en tiempo de ejecución; según estos mismos autores, en este caso se obtiene un mejor equilibrio de la carga.

4) ¿Cuántos procesadores considerar?

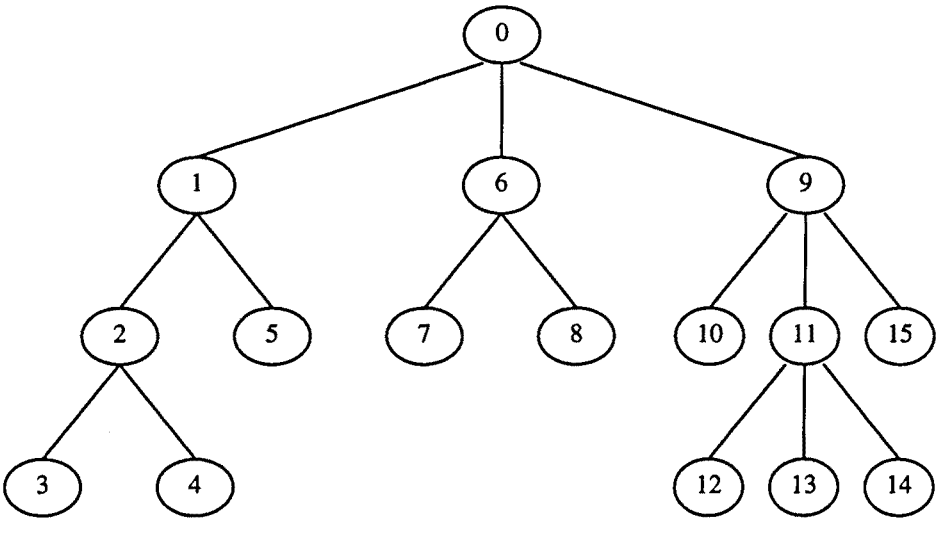
En los sistemas con un gran número de procesadores en paralelo, del orden de miles, usualmente se trabaja con granularidad fina; mientras, la granularidad gruesa es considerada en arquitecturas con menos procesadores, del orden de decenas. De todas formas, y como exponen Gendron & Crainic (1994), la situación cambia rápidamente, de forma que ya en 1994 existen sistemas con miles de procesadores que trabajan con granularidad gruesa. Lo que parece más claro, es que el hecho de que trabajar con memoria compartida implica un número reducido de procesadores para evitar cuellos de botella en el acceso a la memoria.

Otros aspectos a considerar, no por ello menos importantes, son las necesidades y tipología de comunicación y/o sincronización, la facilidad de la codificación, la localización de los datos, el equilibrio en la duración de las tareas, etc. Zenios (1994) y Eckstein (1996) entre otros autores, introducen un nuevo aspecto a tener presente en el momento de resolver un problema mediante paralelización: ¿paralelizar algoritmos ya existentes o probar nuevos algoritmos que busquen ventajas de la estructura del problema: estructuras propicias para ser paralelizadas? De todas formas, parece evidente la necesidad de un cambio en la mentalidad del programador en secuencial, si desea diseñar programas paralelos que sean eficientes.

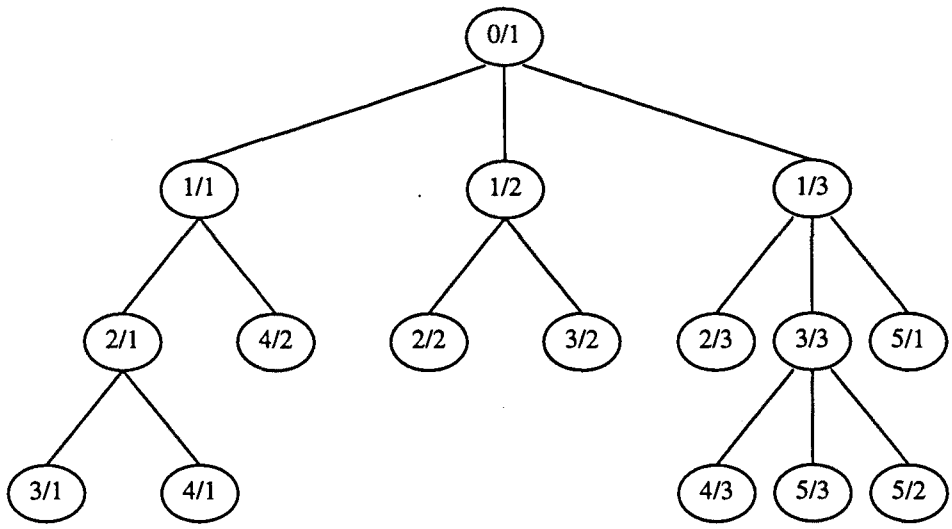
Sea T_p el tiempo necesario para resolver un problema utilizando p procesadores y T_1 el tiempo necesario para resolverlo con único procesador. Entre otros autores, Gendron & Crainic (1994), Kumar et al. (1994), Valero (1995), Eckstein (1996) y Cruz & Mateus (1997), definen las siguientes medidas de rendimiento: la aceleración, S_p, como $S_p = T_1 / T_p$ y la eficiencia, E_p, como $E_p = S_p / p$. Como es lógico, se desea el mayor valor posible de ambos indicadores, y, como señala Ibaraki (1988), lo ideal seria que $T_p \approx T_1/p$.

Aunque puede parecer extraño en un primer momento (con p procesadores se pueden realizar a la vez p trabajos independientes, con lo que teóricamente el tiempo de computación se puede llegar a reducir a 1/p), tanto puede llegar a ocurrir que $T_p > T_1$ -no deseable y denominado por Ibaraki (1988) y Eckstein (1994) daño o desaceleración anómala- o que $T_p < T_1/p$ -totalmente deseable y denominado por estos mismos autores aceleración anómala-, con lo que se pueden llegar a alcanzar aceleraciones mayores que el número de procesadores (Troya & Ortega 1989). Kindervater & Lenstra (1986) comentan que es posible construir ejemplos en los que p procesadores son más lentos que uno solo, o más de p veces más rápidos; mientras, Ibaraki (1988) comenta que se pueden pensar condiciones para tener aceleraciones anómalas en procedimientos *branch and bound* aunque sólo se dan bajo condiciones anómalas, de todas formas no especifica cómo hacerlo.

Kindervater & Lenstra (1986) exponen la aplicación que hizo Pruul en 1975 de un procedimiento *branch and bound* en serie y en paralelo al problema del viajante de comercio, en la que utiliza la búsqueda en profundidad como estrategia de selección del próximo vértice a expandir:



Búsqueda secuencial: el vértice t es seleccionado en el instante t, de Kindervater & Lenstra (1986).



Búsqueda en paralelo con tres procesadores:
el vértice t/p es seleccionado en el instante t por el procesador p, de Kindervater & Lenstra (1986).

Como exponen Kindervater & Lenstra (1986), Pruul alcanzó aceleraciones promedio mayores que el número de procesadores utilizados. Aunque no se especifica cómo, no es difícil de suponer que sea debido al hecho de obtener una poda más efectiva al encontrar mejores valores de la solución preferible en etapas más tempranas de la exploración.

De todas formas hay que tener muy presente que, como advierte Eckstein (1996), la paralelización no es la panacea que lo resuelve todo y puede llegar a ser inapropiada, ya que, como señalan Kumar et al. (1994), los problemas son paralelizables en grados muy diferentes: de nada a mucho. Además, este mismo autor comenta que existen algoritmos paralelos que van mejor o peor que otros en función de, por ejemplo, el número de procesadores disponibles.

Para una mayor profundización sobre paralelismo se recomienda el estudio de la extensa literatura existente sobre el tema o la consulta de revistas especializadas como *Distributed and parallel databases*, *IEEE parallel & distributed technology*, *IEEE transactions on parallel and distributed systems*, *International journal of parallel programming*, *Journal of parallel and distributed computing*, *Neural, parallel & scientific computations*, *Parallel algorithms and applications*, *Parallel computing*, *Parallel processing letters*, etc.

4.2. Paralelismo en problemas de optimización combinatoria.

Zenios (1994) advierte que aunque en el área de la optimización combinatoria existen muchos autores investigando el diseño nuevos algoritmos bajo arquitecturas paralelas,

en este campo el paralelismo es muy difícil de aplicar y se está utilizando mayoritariamente en el desarrollo de procedimientos heurísticos. De todas maneras cabe destacar que, debido a la enorme dificultad de resolución de los problemas combinatorios tratados, es muy grande el interés existente por la paralelización para acelerar (a veces drásticamente según Corrêa 1995) la ejecución de dichas aplicaciones (Kindervater & Lenstra 1986 y Bruin et al. 1988).

Roucairol (1996) comenta que hay varios factores que han contribuido a incrementar la demanda de computación paralela en la resolución de problemas de optimización combinatoria, como vehículo para acelerar los procesos y aumentar la capacidad de memoria:

- se ha incrementado el número de aplicaciones científicas, militares e industriales, que son expresadas como problemas de optimización discreta;

- además, dichas aplicaciones también están incrementando su complejidad: las soluciones deben ser obtenidas cada vez de forma más rápida y a veces en tiempo real (movimiento de robots, etc.).

Con esta herramienta se pueden resolver problemas en menor tiempo y además aumentar el tamaño de los problemas resueltos, ya que, y concretando en los procedimientos de ramificación y acotación, el paralelismo es utilizado para aumentar la construcción del árbol, facilitando más eficientes y extensivos podados gracias a una más rápida adquisición de conocimientos; el precio que se ha de pagar es el de diseñar, analizar e implementar, nuevos y eficientes algoritmos en paralelo, ya que se pueden obtener aceleraciones mayores que p pero también menores que uno (Roucairol 1996).

En la práctica ya se han paralelizado varios algoritmos de resolución de problemas de optimización combinatoria, con los que se han obtenido aceleraciones lineales con el número de procesadores: árbol parcial mínimo, camino más corto entre cualquier pareja de vértices, asignación cuadrática, cubrimiento, problemas de corte bidimensionales, etc. Como indican Kumar et al. (1994) y se ha introducido en el apartado anterior, en todos estos algoritmos la granularidad de las tareas y la forma de asignarlas a los diferentes procesadores puede ser muy variada, dependiendo totalmente de esta división y asignación, los resultados obtenidos de la ejecución del algoritmo paralelo. Además, hay que tener presente que la selección de la estrategia para explotar el paralelismo está grandemente influenciada por la arquitectura de la máquina paralela utilizada y por las características del problema (Gendron & Crainic 1994).

Gendron & Crainic (1994) definen en su extensa y brillante recopilación sobre algoritmos *branch and bound* en paralelo, tres tipos principales de diseño de este tipo de

algoritmos, mientras que la mayoría de autores (Bruin et al. 1988, Ibaraki 1988, Powley et al. 1993, Roucairol 1996, entre otros) únicamente consideran los dos primeros:

a) Paralelismo de tipo 1: paralelismo de bajo nivel o de granularidad fina. Consiste en la introducción del paralelismo en la ejecución de operaciones en los vértices individuales: cálculo de la cota inferior, cota superior, indicador, aplicación de las reglas de podado, etc. Este tipo de paralelismo no tiene influencia en la estructura general de algoritmo *branch and bound*, ya que sólo se paraleliza una parte del algoritmo secuencial y es particular al problema a resolver. De esta forma, el algoritmo paralelo diseñado realiza la misma búsqueda que la versión secuencial: se ramifican los mimos subproblemas y en el mismo orden. Según Roucairol (1996), con granularidad fina se requiere una gran necesidad de comunicación, por lo que no es muy adecuada cuando el número de procesadores es muy elevado.

b) Paralelismo de tipo 2: paralelismo de alto nivel o de granularidad gruesa. Consiste en la construcción de la arborescencia *branch and bound* en paralelo, ejecutando operaciones en varios subproblemas de forma simultánea. Este tipo de paralelismo puede afectar al diseño del algoritmo; de esta forma, el orden en el que se realiza el trabajo puede ser diferente, siendo posible que parte de la exploración realizada por el algoritmo secuencial no sea realizada por la versión paralela y viceversa. Según Roucairol (1996), con granularidad gruesa se requiere una menor necesidad de comunicación, pero esto puede repercutir en trabajar más e innecesariamente al no saber que hacen los otros procesadores. En Mínguez & Roselló (1997) se presenta una implementación del algoritmo *Lomnicki pendular* (Companys 1995) con granularidad gruesa.

c) Paralelismo de tipo 3. Construcción de varias arborescencias *branch and bound* del mismo problema en paralelo, variando la estrategia de ramificación, de acotado, de poda, o de selección del próximo vértice a explorar. La información generada cuando se construye un árbol puede ser utilizada en la construcción de los otros. Este tipo de paralelismo puede afectar al diseño del algoritmo, y, especialmente adaptado para implementaciones con granularidad gruesa y en arquitecturas MIMD asíncronas, ha sido objeto de pocos estudios[20].

Gendron & Crainic (1994) comentan, mediante la enumeración de diversas referencias, que, de todas formas, los tres tipos de paralelismo pueden llegar a ser combinados. Por su parte Powley et al. (1993), expone que, además, existe una aproximación para paralelizar algoritmos de búsqueda: distribuir el espacio de estados entre los diferentes procesadores, de forma que cada estado tiene una correspondencia fija con un

[20] En Gendron & Crainic (1994) se pueden encontrar referencias sobre dichos estudios.

procesador particular; una motivación para esta distribución es el hecho de intentar detectar estados duplicados.

Como se acaba de describir, las posibles aproximaciones van desde aquéllas que utilizan el paralelismo de alto nivel, hasta aquéllas que lo utilizan en el cálculo de las cotas inferiores, superiores, etc. (paralelismo de bajo nivel). De todas formas, en la gran mayoría de aplicaciones de algoritmos *branch and bound* paralelos, únicamente se utiliza el paralelismo de granularidad gruesa, el de tipo 2. En Gendron & Crainic (1994) se presenta un brillante estado del arte y una extensa recopilación de la literatura publicada sobre este tipo de paralelismo desde un punto de vista histórico y distinguiendo tres periodos: los primeros años (1975-1982), en los que existían pocos sistemas paralelos y por tanto se utilizaban arquitecturas experimentales o se hacían simulaciones de paralelismo; los años ochenta (1983-1986), en los que se estudiaron desde el punto de vista teórico las cualidades de los algoritmos *branch and bound* paralelos; y desde 1987, en los que la investigación se ha focalizado en la implementación de diversos tipos de algoritmos en varias arquitecturas paralelas.

Además de decidir el tipo de paralelismo con el que se va a trabajar, hay que tener presentes otros parámetros de diseño que permiten nuevas clasificaciones (Bruin et al. 1988). A continuación se detallan los principales de estos parámetros que, según Gendron & Crainic (1994), hay que considerar, en especial, para el paralelismo de tipo 2:

a) Generación y localización inicial de las cargas de trabajo. El objetivo es utilizar todos los procesadores tan pronto como sea posible, evitando asignarles subproblemas no prometedores; para ello existen diferentes tácticas (Gendron & Crainic 1994 y Roucairol 1996):

1. Asignar el problema original a un procesador y gradualmente asignar a los demás los subproblemas que son generados.
2. Generar varios subproblemas (generalmente en un número mayor que el de procesadores) utilizando una estrategia de ramificación especial (que evite los poco prometedores).
3. Ejecutar un algoritmo *branch and bound* secuencial en un único procesador, hasta que se hayan generado suficientes subproblemas inexplorados para todos los demás.
4. Todos los procesadores ejecutan una fase secuencial en la que cada procesador construye el mismo árbol (aunque no se especifica, se supone que con reglas de exploración diferentes), y, cuando el número de subproblemas inexplorados es al menos igual al número de procesadores, cada procesador selecciona los subproblemas en los que posteriormente trabajará.

5. Mans et al. (1995) proponen un nuevo procedimiento de distribución de la carga, basado en la noción de árbol de alimentación (*feeding tree*). El árbol de alimentación es una estructura de datos en la que la parte superior del árbol de búsqueda es desarrollada completamente hasta un nivel i; los vértices terminales del árbol de alimentación son los vértices raíz de los subárboles que son asignados a los diferentes procesadores. A cada procesador se le asigna un vértice raíz de un subárbol (vértice terminal del árbol de alimentación), en el que desarrolla una búsqueda en profundidad. Cuando acaba su exploración, accede al árbol de alimentación y toma un nuevo vértice que es la raíz de un nuevo subárbol a explorar.

b) Localización y partición de las cargas de trabajo entre los procesadores. El objetivo consiste en dividir y distribuir equitativamente la carga de trabajo entre los procesadores (todos deben trabajar más o menos lo mismo para aprovechar completamente el paralelismo), y en alimentarlos con subproblemas prometedores (para intentar evitar generar subproblemas poco prometedores). Para alcanzar este doble objetivo, Bruin et al. (1988) y Roucairol (1996) señalan que la estrategia de división debe ser dinámica durante la ejecución, pero teniendo en cuenta que el coste de dividir y transferir un trabajo de un procesador a otro no debe ser excesivo. Los dos extremos son: una única lista central accesible por todos los procesadores y una lista privada propia de cada procesador (Bruin et al. 1988, Gendron & Crainic 1994 y Roucairol 1996); de todas formas, el número de listas, y, por tanto, el carácter más o menos central de las mismas, depende de la frecuencia con que los procesadores acceden a dichas listas, del número de procesadores accediendo a la misma lista, de la eficiencia de las comunicaciones, de la tipología de la red de ordenadores, de la estrategia de distribución de la carga y de la susceptibilidad del ejemplar del problema a tener anomalías[21].

Otra cuestión sobre la que es imprescindible pronunciarse es la siguiente: ¿qué pasa cuando un procesador completa su trabajo? Según Bruin et al. (1988) nuevamente se pueden presentar dos extremos: por un lado, antes de comenzar un nuevo trabajo, un procesador puede esperar a que todos los demás acaben los suyos; y, por otro, el procesador puede comenzar un nuevo trabajo inmediatamente, sin esperar que acaben los otros procesadores. Como comenta Corrêa (1995), se presenta la siguiente dicotomía: sincronismo (perder tiempo esperando) frente a asincronismo (realizar trabajo extra al explorar vértices innecesarios).

El hecho de trabajar con una lista única o con una lista para cada procesador, presenta tanto ventajas como inconvenientes; así, trabajando con una estrategia de selección del próximo vértice a explorar primero el de mejor cota, mientras un algoritmo *branch and*

[21] Eckstein (1994) proporciona varias referencias al respecto.

bound paralelo con lista única garantiza que cada procesador selecciona uno de los p vértices con mejor cota, en los de listas múltiples cada procesador selecciona el vértice de mejor cota de su propia lista, pero éste puede no ser uno de los p mejores vértices en global (Troya & Ortega 1989). A continuación se presentan las estrategias de lista única y multilista, así como varias de sus posibilidades:

1) Lista única. Después de asignarse los primeros subproblemas a los procesadores, el procesador central les va asignando nuevos subproblemas de la lista de vértices activos a medida que van acabando su trabajo. Para su implementación se suele utilizar una estrategia de control estática y asíncrona. Según Roucairol (1996) existe más comunicación, pero puede ser más rápida la poda; además puede haber un cuello de botella por el alto tráfico de comunicaciones.

2) Multilista. En una multilista con una lista privada propia para cada procesador, es usual utilizar estrategias de control dinámicas para intercambiar subproblemas. En una primera clasificación realizada por Gendron & Crainic (1994) y Roucairol (1996), se distinguen tres clases de estrategias dinámicas:

a) estrategia con petición: un procesador con poco trabajo o inactivo, pide trabajo a otro procesador: si éste acepta, le cede una parte de su trabajo (después de decidir qué subproblemas le transfiere y cómo los selecciona), y si lo rechaza, puede decidir enviar una nueva petición a otro procesador;

b) estrategia sin petición: los procesadores deciden enviar trabajo a otros procesadores aunque no lo soliciten, de forma que, el que cede trabajo, debe decidir cada cuánto, cuántos subproblemas transfiere, a quién las envía y cómo los selecciona;

c) estrategia combinada: combina las dos anteriores: los procesadores envían trabajo sin preguntar, aunque también lo demandan cuando su lista se acorta.

Por otro lado, la mayoría de autores prueban diferentes estrategias de distribución de la carga, que, como comentan Xu et al. (1995), es esencial para un uso eficiente de los procesadores ya que el tiempo de computación consumido por los subproblemas no es uniforme. De todas maneras, es difícil establecer qué es una buena función de distribución, aunque un requerimiento básico es el de realizar una buena distribución de los vértices activos sobre todas las listas; además hay que tener muy presente las necesidades de comunicación (Troya & Ortega 1989). A continuación se describen diferentes estrategias descentralizadas de distribución:

1) Equilibrado estático. No hay distribución de carga y cada procesador explora totalmente los subproblemas que se le asignan inicialmente (Cruz & Mateus 1997). Según Roucairol (1996), el único intercambio de información

entre los procesadores es el valor de una posible mejor solución preferible (que puede hacerse tan pronto como es encontrado -algoritmos asíncronos- o en instantes o iteraciones determinadas -algoritmos síncronos-).

2) Equilibrado de Zhang o estrategia de asignación aleatoria (RAND). Cuando un procesador expande un problema, los subproblemas generados son enviados a uno o varios procesadores vecinos seleccionados aleatoriamente. Estrategia muy simple de implementar (Troya & Ortega 1989, Karp & Zhang 1993, Eckstein 1994, Xu et al. 1995 y Cruz & Mateus 1997).

3) Equilibrado de Zhang modificado. Es una modificación de la estrategia anterior, en la que se impide que un procesador envíe subproblemas a otros si él se queda sin carga (Cruz & Mateus 1997).

4) Estrategia de asignación aleatoria priorizada (PRAND). En este caso, la información de las cotas de los subproblemas es considerada en sus procesos de decisión, de forma que es el segundo mejor subproblema de la lista local el que es seleccionado para la migración a un procesador vecino seleccionado aleatoriamente (Xu et al. 1995).

5) Función determinista. Consiste en insertar cíclicamente los subproblemas generados en todas las listas de los procesadores. También existe la versión que inserta los subproblemas generados en un subconjunto fijo de las listas de p procesadores (Troya & Ortega 1989).

6) Estrategia de contracción adaptativa entre el vecindaje (ACWN). Cuando se genera un nuevo subproblema, éste es enviado al vecino que está menos cargado (Xu et al. 1995).

7) Estrategia de contracción adaptativa priorizada entre el vecindaje (PACWN). En este caso, la información de las cotas de los subproblemas también es considerada en sus procesos de decisión, de forma que es el segundo mejor subproblema de la lista local el que es seleccionado para ser enviado al vecino que está menos cargado (Xu et al. 1995).

8) Estrategia de promedio local LADE. Un procesador con necesidad de carga, equilibra su carga de trabajo con la de uno de sus vecinos, teniendo en cuenta el promedio de las cargas de trabajo de dichos vecinos (Xu et al. 1995).

9) Estrategia de promedio local LADF. Un procesador con necesidad de carga, equilibra su carga de trabajo con la de sus vecinos cercanos, teniendo en cuenta el promedio de las cargas de trabajo de dichos vecinos (Xu et al. 1995).

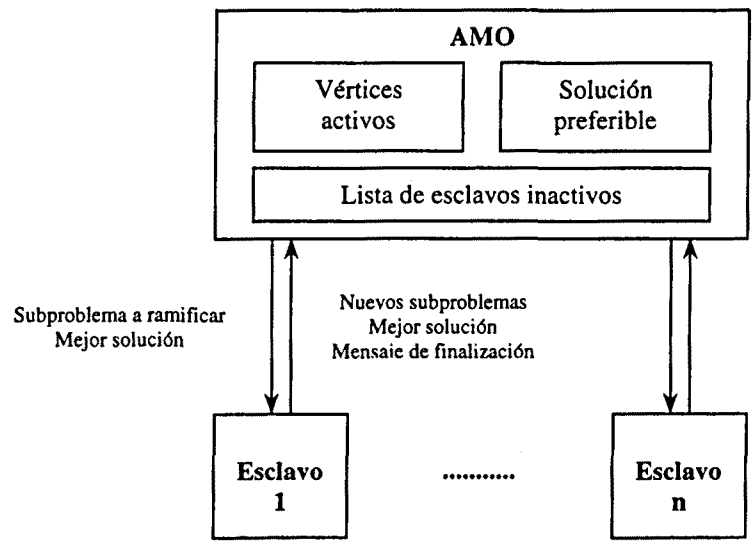
10) Se determina una función peso asociada a la carga local de cada procesador y sólo las diferencias significativas entre pesos de los procesadores adyacentes provocan el envío de subproblemas (Almeida et al. 1995).

11) Roucairol (1996), realiza un intento de síntesis de estas estrategias, de forma que el intercambio de subproblemas se produce tan pronto como el tamaño de la lista local se incrementa o decrementa en un % respecto al último cambio:

- un procesador sólo envía trabajo a otro que lo pide, si su carga local es muy diferente que la del procesador que demanda,
- el trabajo se envía en función de unas reglas preestablecidas de antemano: al vecino más próximo, de forma cíclica a un nodo o a cada uno de los p procesadores, de forma regular o a intervalos, etc.,
- el trabajo se envía de forma aleatoria a otro u otros procesadores y en función o no de una distribución de probabilidad.

c) Intercambio de información: actualización de la solución preferible. Cuando un procesador encuentra un mejor valor de la solución preferible, la puede enviar inmediatamente a todos los demás procesadores (Gendron & Crainic 1994) o puede esperarse. Bruin et al. (1988) comentan, también en referencia al apartado anterior, que como la comunicación tiene un coste, se debe alcanzar un equilibrio entre las ventajas que proporciona el aumento de información y el tiempo de comunicación necesario para dichos intercambios.

El procedimiento más habitual de implantación de un algoritmo *branch and bound* en paralelo consiste en un esquema amo-esclavos, donde el amo tienen la misión de controlar y gestionar, y los esclavos la de trabajar. Este tipo de esquema admite tanto la estrategia de lista única como la multilista, cuyos códigos varían en función de la política utilizada para reducir el desequilibrio de carga entre los procesadores (Almeida et al. 1995). A continuación se muestra un ejemplo de un algoritmo *branch and bound* paralelo asíncrono, con una única lista central de vértices activos, y con un proceso amo y n esclavos (Bruin et al. 1988):



Algoritmo paralelo asíncrono, con una única lista central, y un proceso amo y n esclavos, de Bruin et al. (1988).

En este ejemplo el amo toma todas las decisiones importantes y los esclavos las ejecutan. El amo mantiene la lista de vértices activos y decide qué subproblemas son ramificados por quién y cuándo. Los esclavos realizan la ramificación y calculan las cotas de los subproblemas generados, y, cuando acaban, se lo notifican al amo quien recopila toda la información generada por los esclavos, la interpreta y realiza el mantenimiento de la lista de vértices activos y la solución preferible. Si la solución preferible es actualizada, el amo envía esta información a todos los esclavos, por lo que son ellos quienes hacen el test de cota inferior en los subproblemas generados.

De una manera más concreta, Cruz & Mateus (1997) describen el pseudo-codigo de un procedimiento *branch and bound* secuencial, así como de un *branch and bound* paralelo con lista única y otro con multilista. Sea un problema de minimización, donde, en la terminología utilizada por Ibaraki (1988) y tomada en este texto como referencia, $\overline{Z}$ es el valor de la solución preferible, P_i es el subproblema i a resolver, $g(P_i)$ es una cota inferior del vértice P_i, $u(P_i)$ es una cota superior del vértice P_i, A es el conjunto de vértices activos y se utiliza la búsqueda en profundidad como estrategia de selección del próximo vértice a expandir.

El pseudo-codigo de la versión secuencial del algoritmo *branch and bound* descrito por Cruz & Mateus (1997) es el siguiente:

```
Algoritmo branch-and-bound
    Z̄ ← +∞
    A ← {P0}
    Mientras A ≠ 0 hacer
            /Selección/
        Seleccionar un problema Pi y eliminarlo de A
            /Acotado/
        Calcular g(Pi) y u(Pi)
        Actualizar Z̄
            /Ramificación/
        Si g(Pi) < Z̄ y Pi no es terminal entonces
            A ← A ∪ {Pij}
        Fin si
    Fin mientras
Fin algoritmo
```

Pseudo-codigo de la versión secuencial de un algoritmo *branch and bound*, de Cruz & Mateus (1997).

Como se ha introducido, Cruz & Mateus (1997) exponen una versión centralizada (de lista única) paralela amo-esclavos de su algoritmo *branch and bound* secuencial. En esta versión, el amo administra la lista A de problemas P_i inexplorados, mientras que la tarea de expandir los subproblemas se asigna a los esclavos: calcular las cotas y generar los vértices descendientes P_{ij}.

```
Z ← +∞; A ← {P0}
Mientras hay trabajo hacer
    Si A ≠ 0 entonces
        Seleccionar un problema Pi de A
        Enviar Pi y Z al esclavo
    Fin si
    Si hay una respuesta entonces
        Recibir resultados del esclavo
        Actualizar Z
        A ← A ∪ A'
    Fin si
Fin mientras
Final de los esclavos
```

Algoritmo del amo

```
Mientras hay trabajo hacer
    Recibir Pi y Z del amo
    Calcular g(Pi) y u(Pi)
    Actualizar Z
    Si g(Pi) < Z y Pi no es terminal entonces
        A' ← {Pij}
    sino
        A' ← ∅
    Fin si
    Enviar resultados al amo
Fin mientras
```

Algoritmo de un esclavo

Pseudo-codigo de la versión paralela centralizada amo-esclavos de un algoritmo *branch and bound*, de Cruz & Mateus (1997).

El amo no finaliza su trabajo mientras existen subproblemas a ser resueltos en la lista A o mientras algún esclavo esté trabajando. Si existe un problema en la lista A, el amo elige el siguiente a explorar según la estrategia de búsqueda en profundidad y lo envía al primer esclavo que esté libre. Cuando una expansión finaliza, el amo recibe de un esclavo las cotas superiores e inferiores y la lista de vértices generados recientemente A'; con esta información actualiza $\overline{Z}$ y la lista A. Como también comentan Troya & Ortega (1989), esta implementación requiere un alto grado de sincronismo y esto puede causar un cuello de botella en el amo y, como resultado, una subutilización de los esclavos; sin embargo, puede ser eficiente si la granularidad de los problemas es suficientemente grande: si las expansiones son computacionalmente más intensas que los costes de comunicación.

Por otro lado, Cruz & Mateus (1997) también exponen una versión distribuida (multilista) paralela amo-esclavos de su algoritmo *branch and bound* secuencial. En esta versión, el amo únicamente es responsable de la carga inicial de los esclavos y del algoritmo de finalización; cada esclavo implementa el algoritmo *branch and bound* secuencial, con débiles modificaciones que permiten el cambio de carga con otros esclavos.

```
Z ← +∞
Generar k problemas Pi
Para todo k hacer
    Enviar Pi^k al esclavo k
Fin para
Mientras alguien esté trabajando hacer
    Recibir el estado actual del esclavo
Fin mientras
Para todo k hacer
    Recibir resultados del esclavo k
    Actualizar Z
Fin para
```

Algoritmo del amo

```
Recibir el problema Pi
Z ← +∞; A ← {Pi}
Mientras no termine hacer
    Si A ≠ 0 hacer
        Seleccionar un Pi de A
        Calcular g(Pi) y u(Pi)
        Actualizar Z
        Si g(Pi) < Z y Pi no es terminal
        entonces
            A ← A ∪ {Pij}
        Fin si
        Cambiar trabajo con otros esclavos
    Fin si
    Recibir información de otros esclavos
    Enviar el estado actual al amo
Fin mientras
Enviar resultados al amo
```

Pseudo-codigo de la versión paralela distribuida amo-esclavos de un algoritmo *branch and bound*, de Cruz & Mateus (1997).

El amo expande P_0 hasta tener tantos subproblemas P_i^k como esclavos, distribuye la carga entre los k esclavos y espera que éstos acaben de explorar completamente el subárbol asignado con el subproblema P_i^k. Posteriormente, el amo recibe todas las mejores soluciones parciales y selecciona la mejor de entre todas ellas, finalizando el algoritmo.

Como ya se ha introducido y comenta Eckstein (1994), la literatura de procedimientos *branch and bound* paralelos presta una particular atención al tema de la búsqueda anómala, debida a que el orden en el que son expandidos los subproblemas puede, a través de los mecanismos de poda, afectar a la cantidad de trabajo que debe ser realizado después. El tamaño y la forma de la arborescencia de búsqueda puede variar con el número de procesadores, provocando aceleraciones o desaceleraciones anómalas[22].

Una vez expuestas diferentes técnicas de paralelización de los procedimientos *branch and bound*, es fácil de imaginar la introducción de la paralelización en otros procedimientos de búsqueda tales como el *branch and bound* con planos de corte, el *branch and cut* (como exponen Eckstein 1994 -que comenta que ya se han realizado dos aplicaciones de estas técnicas: una para el TSP y otra para programación binaria[23]- y Jünger et al. 1994), el IDA* tanto en arquitecturas SIMD (Mahanti & Daniels 1993 y Powley et al. 1993) como anteriormente en arquitecturas MIMD (Rao & Kumar 1987), etc.

Existen algoritmos que poseen un paralelismo intrínseco que facilita su resolución en paralelo. Este es el caso de la búsqueda en haz de amplitud n (*beam search*) descrita en el apartado 3.9.4.2., y que, recuérdese, es una técnica heurística de búsqueda en la que, en cada nivel, se exploran en paralelo los n mejores vértices encontrados en dicho nivel, descartándose todas las demás. Como expone Bisiani (1990), este procedimiento de búsqueda admite claramente el cálculo paralelo y para potenciar sus posibilidades se han diseñado arquitecturas especiales.

Como señala Roucairol (1996), existe una gran dificultad de programación de los *branch and bound* paralelos; de todas formas, comenta que se ha de seguir en este camino ya que los logros obtenidos están en contradicción directa con los argumentos de aquéllos que piensan que resolver problemas NP-completos es sinónimo de procedimiento aproximativo. Por su parte, Gendron & Crainic (1994) comentan muchas posibles extensiones y puntos de investigación, lo que da idea de lo lejos que se está de un conocimiento total del tema.

[22] En Eckstein (1994) se enumeran diversas referencias al respecto.

[23] La aplicación de un algoritmo *branch and cut* paralelo en la resolución de programas binarios se puede consultar en Cannon & Hoffman (1990).

4.3. Librerías *branch and bound* en paralelo.

Valero (1995) comenta que de cara al futuro se diseñarán supercompiladores (capaces de detectar automáticamente la posibilidad de ejecutar de forma paralela partes de un código secuencial, generando instrucciones que utilicen los mecanismos de hardware que ofrece la arquitectura) y se paralelizarán librerías científicas (colecciones de rutinas que implementan operaciones habituales y de fácil portabilidad).

Actualmente ya existen librerías y paquetes de software que paralelizan procedimientos *branch and bound.* A modo de ejemplo cabe citar dos de ellos:

a) PPBB-Library: desarrollada en la *University of Paderborn* por el grupo de trabajo del profesor B. Moniem y accesible en http://www.uni-paderborn.de/fachbereich/AG/monien/SOFTWARE/PPBB/. Esta librería ha sido diseñada para ser utilizada en arquitecturas de memoria distribuida, y, por ejemplo, Xu et al. (1995) la utilizan para probar y comparar las características de varias estrategias de equilibrado de carga entre los procesadores.

b) BOB: desarrollada por el *PRiSM laboratory* de la *Université de Versailles* por el grupo de trabajo de la profesora C. Roucairol y accesible en http://www.prism.uvsq.fr/english/parallel/cr/bob_us.html. Esta librería está diseñada para ser utilizada tanto en arquitecturas de memoria distribuida como compartida.

El objetivo de este tipo de software es el de proporcionar herramientas de trabajo para desarrollar fácilmente aplicaciones *branch and bound* paralelas, que, además, sean independientes de las máquinas y que se sitúen entre el modelo y la máquina: paralelizar algoritmos *branch and bound* secuenciales en varias arquitecturas paralelas, sin que el usuario tenga que tener conocimientos ni de la arquitectura del hardware ni de los mecanismos de paralelización; únicamente es necesario implementar la ramificación y los procedimientos de acotado y evaluación (Le Cun et al. 1995 y Xu et al. 1995).

La librería se encarga de la gestión de los subproblemas que se van creando durante la ejecución del algoritmo *branch and bound* y equilibra su distribución; de todas maneras, estas librerías suelen permitir al usuario implementar y probar sus propias estrategias de equilibrado de la carga, y especificar el número de procesadores, la granularidad de la computación y los resultados a ser visualizados (Le Cun et al. 1995).

Como exponen Le Cun et al. (1995), para trabajar con estas librerías es necesario desarrollar el código de las funciones que dependen directamente de la naturaleza del problema tratado; en el caso de la librería BOB, las funciones que debe implementar el usuario son las siguientes: la inicialización (cotas inferior y superior iniciales, el vértice

raíz, el tamaño del problema y la mejor solución inicial conocida), el procedimiento de finalización, visualización en tiempo real de la solución preferible y generación de los subproblemas (procedimiento de ramificación y criterio de evaluación de los subproblemas). Además, se debe proporcionar la estructura de los vértices del árbol de búsqueda.