TESIS DOCTORAL

ALGORITMOS HEURÍSTICOS PARA LA SELECCIÓN DE SECUENCIAS ÓPTIMAS DE ENSAMBLAJE

por

Carmelo del Valle Sevillano

Presentada en la Escuela Superior de Ingenieros
de la Universidad de Sevilla
para la obtención del
Grado de Doctor Ingeniero Industrial

Sevilla, septiembre de 2001

TESIS DOCTORAL

ALGORITMOS HEURÍSTICOS PARA LA SELECCIÓN DE SECUENCIAS ÓPTIMAS DE ENSAMBLAJE

Autor: Carmelo del Valle Sevillano

Directores: Eduardo Fernández Camacho
Miguel Toro Bonilla

Fdo: CARMELO DEL VALLE SEVILLANO

Fdo: EDUARDO FERNÁNDEZ CAMACHO

Fdo: MIGUEL TORO BONILLA

A mi familia

Agradecimientos

Quiero expresar mi gratitud a mis directores de tesis, Eduardo Fernández Camacho y Miguel Toro Bonilla, quienes, con su apoyo y dedicación, ayudaron a que este trabajo saliera adelante.

A los compañeros de los departamentos de Lenguajes y Sistemas Informáticos y de Ingeniería de Sistemas y Automática por sus críticas y comentarios.

A Paqui, por su constante ayuda durante el desarrollo de esta tesis.

Índice de Contenidos

Índice de Figuras

Capítulo 1

Introducción

Esta tesis se centra en la selección óptima de secuencias de ensamblaje para la fabricación de productos en un entorno con múltiples estaciones de trabajo. El enfoque usado es la resolución de un problema de planificación en el que por un lado deben seleccionarse las operaciones de montaje para formar un plan completo y por otro deben secuenciarse las tareas seleccionadas según un orden óptimo, desde el punto de vista del tiempo total del ensamblaje.

En todas las áreas de la industria, la optimización en el uso de los recursos resulta de gran importancia. En muchas ocasiones, el proceso de producción implica el secuenciamiento de tareas dentro de un rango de posibilidades de carácter combinatorio. Es frecuente que la solución adoptada para resolver estos problemas de secuenciamiento se realice de forma manual a través de los criterios de los expertos en la materia de que se trate, simplificando el dominio del problema y sin analizar el amplio espectro de posibles soluciones que podrían ser tenidas en cuenta. De esta forma, se adoptan secuencias de trabajo que a menudo se alejan de las que podrían considerarse óptimas para el buen funcionamiento de los procesos de producción.

Los problemas de planificación y secuenciamiento se encuentran entre los más complejos de resolver computacionalmente, por un lado por su naturaleza combinatoria y por otro por los factores específicos de la aplicación de los que suele acompañarse. Su resolución determinista ha venido ligada a la utilización de algoritmos de búsqueda sobre el espacio de soluciones que, de una manera más o menos inteligente, tratan de obtener el óptimo. Sin embargo, por la propia naturaleza de estos problemas, no hay garantía de lograr encontrar ese óptimo en un tiempo razonadamente pequeño (polinó-

mico), por lo que en situaciones prácticas, cuando el tiempo de computación es limitado, suele recurrirse a algoritmos que, con algún tipo de heurística, consiguen una buena solución, aunque sin garantía de resultar ser la óptima.

1.1 Motivación

Una aplicación que se corresponde con los condicionantes descritos es la planificación del ensamblaje de productos, que incluye, entre otras tareas, la identificación, selección y secuenciamiento de las operaciones de montaje, desde el punto de vista de su efecto en la construcción de nuevos submontajes. En un nivel inferior de abstracción se encontraría la planificación de los movimientos de los robots para llevar a cabo el programa de trabajo obtenido en la fase previa.

La elección de la secuencia en que las piezas y submontajes intermedios deben unirse en el ensamblaje mecánico de un producto puede afectar drásticamente a la eficiencia del proceso de ensamblaje. Existe la necesidad de encontrar una secuencia de ensamblaje eficiente, que supongan en general menos dispositivos de fijación, menos cambios de herramientas y el uso de operaciones más simples y fiables. La elección de la secuencia de ensamblaje es realizada usualmente por un experto. Hay una creciente necesidad de sistematizar y computerizar la generación de secuencias de ensamblaje por varias razones: evitar que se pasen por alto buenas secuencias de ensamblaje, disminuir costes y tiempos de planificación y programación, generar secuencias de ensamblaje y desensamblaje (sustitución de partes defectuosas) en sistemas autónomos para aplicaciones espaciales o de exploración marina, adaptar el proceso de ensamblaje a diferentes máquinas en entornos poco estructurados...

La identificación de las operaciones de ensamblaje conduce normalmente al conjunto de todos los planes de montaje factibles, y existe un amplio consenso en la metodología para su obtención, que involucra el estudio de los aspectos geométricos (dimensiones, conectividad y accesibilidad) de las piezas individuales y de los submontajes, así como de la estabilidad de los mismos [Homem de Mello and Sanderson, 1991b] [Ames, *et al.*, 1995] [Romney, *et al.*, 1995].

La siguiente consideración en la aplicación anterior es la obtención de un plan de montaje óptimo, seleccionado del conjunto de todos los planes factibles. El problema de la selección óptima de secuencias de ensamblaje continúa siendo un tema muy abierto, existiendo una fuerte investigación sobre él. Se han usado una gran diversidad de criterios para escoger un plan óptimo. La mayoría de ellos tratan de simplificar el plan de ensamblaje, por ejemplo evitando movimientos complicados y estados delicados o inestables, minimizando las tareas poco productivas, como las que implican cambios de dispositivos de fijación o reorientación de piezas. Todos estos criterios son usados para lograr un plan previo sobre el que trabajar posteriormente para obtener el secuenciamiento de las tareas que lo componen. Por tanto, tienen poco en cuenta la ejecución real del plan de ensamblaje, al no considerar cómo pueden seleccionarse y ordenarse las tareas para obtener un programa de montaje óptimo.

En este trabajo se ha usado como criterio la evaluación de las secuencias en base a su ejecución en el entorno real de trabajo, a través de su simulación y la obtención de parámetros de interés, como suele ser típicamente el tiempo total de la ejecución del plan. Se ha considerado un sistema de ensamblaje con múltiples máquinas de ensamblaje, pudiendo por tanto realizarse varias operaciones de ensamblaje asociadas al mismo producto de forma simultánea.

Existen múltiples referencias en las que se abordan típicos problemas de secuenciamiento (*Job Shop Scheduling*, viajante de comercio, etc.) usando diferentes técnicas de optimización. Es preciso señalar que el problema aquí planteado supone una mayor complejidad, debido a que no sólo está involucrado el secuenciamiento de las tareas de montaje, sino también la propia selección de las mismas de entre todas las posibles, y de manera que en su conjunto formen un plan de ensamblaje válido, capaz de realizar el ensamblaje completo del producto en cuestión. En este sentido, el problema viene definido a través de restricciones de tipo *And/Or*, representadas a través de un grafo *And/Or* [Homem de Mello and Sanderson, 1990], que recoge de manera implícita el conjunto de todos los planes de ensamblaje factibles. Con este tipo de restricciones, en [Gillies and Liu, 1995] se demuestra que el problema de planificación resultante es NP-completo cuando se intenta minimizar el tiempo total del proceso. Debe añadirse que existen pocas referencias que traten de problemas de planificación en donde se consideren operaciones alternativas para construir la solución [Ahn, *et al.*, 1993] [Beck and Fox, 2000], lo cual ha supuesto una motivación adicional.

1.2 Aportaciones

La primera aportación del trabajo realizado en esta tesis es la definición del modelo del problema sobre el que se van a construir las soluciones. La principal consecuencia de dicho modelo es que los resultados pueden usarse en distintas etapas de la planificación, y teniéndose en cuenta la ordenación final de las secuencias en la ejecución del programa de montaje. En una primera fase los algoritmos mantendrían todas las posibilidades (tareas, recursos...), e irían orientados a eliminar aquellas que, bien por su coste económico (necesidad de recursos adicionales) o por su poca efectividad (tiempo total de ensamblaje elevado), no son deseables. En esta primera fase no suele perseguirse el óptimo, entre otras causas porque el número de soluciones alternativas puede ser extraordinariamente alto. En etapas posteriores, una vez que el problema se ha simplificado, el objetivo de encontrar el óptimo podría resultar más viable. Además, los algoritmos desarrollados pueden representar el núcleo principal de algoritmos que operen en línea durante la propia ejecución del plan de ensamblaje.

Como se ha indicado, el criterio escogido para la determinación de secuencias óptimas de ensamblaje es el tiempo total del ensamblaje del producto. Para ello, se ha supuesto el caso general de la realización en un sistema con múltiples estaciones de trabajo, lo que permitiría la ejecución simultánea de operaciones de ensamblaje correspondientes al mismo producto. En este sentido, el uso de grafos *And/Or* para la

representación implícita de todos los planes de ensamblaje factibles es muy adecuada. Aparte de las duraciones asociadas a las tareas, se consideran otros aspectos que influyen en el tiempo total del ensamblaje, como son los retardos provocados por los cambios de configuración de las máquinas que pueden necesitarse entre la ejecución de tareas, y los retardos asociados al transporte de los submontajes intermedios. Estas restricciones representan otro factor de complejidad adicional. En [Lawler, *et al.*, 1985] se muestra la correspondencia de la planificación de tareas donde se tienen retardos producidos por cambios en las configuraciones en las máquinas dependientes de la secuencia con el problema del viajante de comercio, problema bien conocido por su NP-completitud.

Para la resolución del problema se han utilizado distintas técnicas algorítmicas de optimización, tanto deterministas como no deterministas. En la parte no determinista, se han empleado algoritmos genéticos, una técnica ampliamente usada para resolver problemas de tipo combinatorio como el que nos ocupa. En el lado determinista, se utilizan algoritmos A*, para los que se han definido distintas funciones heurísticas, usadas para mejorar la eficiencia de los algoritmos en la búsqueda de la solución óptima. Por último, se propone el uso de la Programación con Restricciones para la resolución del problema, considerándolo como un problema de satisfacción de restricciones.

En el uso de los algoritmos genéticos se proponen dos modelos para la representación sintáctica de los individuos. El primero de ellos se corresponde con una representación binaria, lo que ha permitido utilizar los operadores genéticos estándar, aunque su efectividad queda condicionada por la calidad de la información genética asociada a esa representación, que podría resultar deficiente. La segunda de las representaciones propuestas, la representación simbólica, está más cercana a la estructura de las soluciones del problema. Sin embargo, han debido definirse para ella operadores genéticos específicos.

En la aplicación algoritmos A*, se han definido varias heurísticas admisibles, partiendo de dos modelos relajados del problema. Dos funciones heurísticas básicas muestran la naturaleza de las restricciones tenidas en cuenta. En la primera de ellas únicamente se consideran las restricciones de precedencia entre tareas, lo cual nos permite obtener una cota inferior del tiempo restante necesario para la ejecución de todas las tareas incluidas en cada subárbol de precedencias. En la segunda, sólo se consideran los tiempos totales de uso de cada recurso para la estimación del tiempo necesario para la ejecución de las tareas aún no introducidas en la solución, ignorando las relaciones de precedencia entre tareas. Ambas heurísticas son mejoradas por otras que incorporan información asociada a las restricciones no consideradas inicialmente. En esas mejoras ha jugado un papel importante la definición de modelos basados en multiconjuntos parcialmente ordenados (*pomsets*) para la estimación del número de cambios de herramientas.

El problema ha sido también abordado mediante el uso del paradigma de la programación con restricciones. Los principales beneficios que se obtienen son una fácil formulación del problema y una mayor expresividad de las soluciones obtenidas, que

incluyen las holguras existentes para las variables temporales asociadas a la ejecución de las operaciones de ensamblaje, lo que permite un análisis más directo de las soluciones. La forma natural en que se expresan las restricciones que definen el problema permite una fácil modificación de las aplicaciones. Los primeros resultados se han obtenido usando el lenguaje CLP(R). Sin embargo, este lenguaje, basado en la Programación Lógica, parece no ser el más adecuado para abordar problemas altamente combinatorios como el que estamos tratando. Los recientes y continuos avances que se están produciendo en esta relativamente nueva tecnología permiten vislumbran mejores expectativas.

1.3 Organización de la tesis

El resto de la tesis se organiza de la siguiente manera: en el Capítulo 2 se presenta el problema que se quiere resolver en el marco del problema general del ensamblaje de productos, junto con una definición más precisa del modelo adoptado para la resolución del problema. En el Capítulo 3 se aborda el problema propuesto a través de la aplicación de algoritmos genético. En el Capítulo 4 se estudia la utilización de algoritmos A* para la resolución del mismo problema. En el Capítulo 5 se plantea la Programación con Restricciones como otra técnica de interés para resolver el mismo problema. En los capítulos anteriores (3, 4 y 5) se ofrecen distintos resultados de la aplicación de las técnicas correspondientes, y en el capítulo 6 se realiza una comparación de los mismos. Asimismo, en el Capítulo 6 se resume el trabajo realizado, y se presentan distintas líneas de trabajo futuro.

Capítulo 2

Planificación de Secuencias de Ensamblaje

2.1 Introducción

El ensamblaje de un producto representa una de las fases en todo el proceso de fabricación del mismo. En el proceso conjunto se trata en general de obtener la mejor calidad del producto con el mínimo coste. En ese objetivo influyen multitud de factores, de manera que existe una mayor o menor dependencia de unos frente a otros. De esta forma, el análisis de un determinado aspecto quedaría incompleto si no se hace referencia a los factores que no se están considerando directamente, bien porque están asociados a parámetros fijados en etapas previas, bien porque se asumen como poco influyentes.

Una clasificación de las etapas que conforman todo el proceso de fabricación de productos viene dada en [Kusiak, 1990]:

- Diseño de componentes y productos
- Diseño de herramientas y dispositivos de fijación
- Planificación de procesos
- Programación de dispositivos
- Planificación de la producción: requerimiento de material, carga y secuenciación de máquinas

- Mecanización: torneado, perforación, soldadura...
- Ensamblaje
- Mantenimiento
- Control de calidad
- Inspección
- Almacenaje y recuperación

En la planificación del proceso de ensamblaje se distinguen varias etapas, desde la propia identificación de las operaciones de ensamblaje, hasta el secuenciamiento de las tareas seleccionadas para efectuar el ensamblaje. Este último aspecto es un factor que influye de manera determinante en el coste final del producto.

Sin entrar en un estudio exhaustivo, son claras las dependencias del ensamblaje con las primeras etapas indicadas. Por un lado, el equipamiento disponible en la planta suele ser un factor poco modificable, al estar compuesto por maquinaria cara. La planificación de procesos debe tener en cuenta tales restricciones para una configuración óptima de la planta. Sin embargo, no deben despreciarse alternativas que, suponiendo mayores inversiones incorporando nuevos equipos, faciliten el proceso mejorándolo en su conjunto. En el mismo sentido, el diseño del producto y sus componentes puede venir condicionado por el equipamiento disponible, a veces intentando aprovechar los dispositivos para la manipulación de piezas y submontajes. En otros casos, será necesario diseñar herramientas y otros dispositivos específicos para el nuevo producto a fabricar, teniendo en cuenta además el resto del equipamiento disponible.

El diseño de productos para su ensamblaje automático (DFA – *Design for Assembly*) ofrece una metodología para asistir al diseñador en la reducción del coste del producto mejorando su *ensamblabilidad*. En esa metodología se tienen en cuenta fundamentalmente aspectos geométricos y físicos de los componentes del producto, como por ejemplo la forma que deben tener las piezas para facilitar su alineamiento o su cogida por las garras de los robots, o los sistemas de unión usados para ensamblar las distintas piezas (a presión, soldadura, atornillado, pegado, etc.). El efecto sobre los procesos de ensamblaje es la obtención de una mayor eficiencia, tanto en el aspecto de la seguridad, evitando posibles fallos en las operaciones de montaje, como en la consecución de tiempos de ensamblaje más cortos.

Asimismo, la planificación de secuencias de ensamblaje juega un papel central en el diseño del producto y del propio sistema de ensamblaje. La elección de la secuencia de montaje puede afectar a la eficiencia y coste final del proceso. Así, el orden en que las piezas son ensambladas puede influir, por ejemplo, en el número de cambios de orientaciones de los submontajes y de herramientas en los robots, el número y complejidad de los dispositivos de fijación, y la complejidad de las operaciones de ensamblaje. Por otro lado, la elección de la secuencia de montaje puede afectar, y verse afectada por diversos factores, tales como la elección y disposición del equipamiento, por las estrategias para realización de tests y reparaciones, y el método de ensamblado, automático o manual.

Sin embargo, las técnicas convencionales de DFA ignoran importantes aspectos de tipo combinatorio en montajes complejos [De Fazio, *et al.*, 1999], principalmente en las particiones asociadas a los posibles submontajes y la elección de las secuencias de ensamblaje. Así pues, es importante que en la etapa de diseño se disponga de un estudio de la ejecución de las posibles secuencias de montaje posibles, que permita detectar cuáles son los factores de diseño que más influyen en la consecución de un plan de trabajo óptimo.

El presente trabajo se centra en la elección de las secuencias óptimas de ensamblaje de un producto. Aparte de su utilización como solución para ser tomada como referencia en la ejecución del propio ensamblaje, podrá servir en el propio proceso de diseño para mejorar las posibles deficiencias que pudieran detectarse. De esta forma, se intentarían mejorar las operaciones críticas, o reasignar diferentes recursos a las actividades. Dentro de las estrategias de planificación sería útil disponer de un estudio sobre la influencia en el coste final del uso de diferentes conjuntos y configuraciones de recursos, donde se tendría en cuenta la posibilidad de incorporar nuevo equipamiento en la planta.

Es importante señalar las diferencias más relevantes del ensamblaje con respecto a los procesos de mecanizado [Boneschanscher, 1993]:

- En los procesos de mecanizado sólo se necesita la pieza sobre la que se realiza el tratamiento. Se necesitan simultáneamente dos piezas (o submontajes) en una estación de trabajo para el caso del ensamblaje, lo que implica un problema de sincronización. Esto conllevará en general mayores requerimientos en los equipos de manipulación de piezas.

- En el mecanizado de piezas, el equipamiento es en su mayor medida específico del tipo de operación a realizar. En el ensamblaje de piezas, es la forma de éstas la que condiciona el equipamiento necesario.

- La relación entre la duración de las operaciones de mecanizado y los tiempos de puesta a punto de las máquinas es en general mucho mayor que 1, lo que permite despreciar en muchas ocasiones las operaciones de puesta a punto. En cuanto a las operaciones de ensamblaje, su duración es del mismo orden que los tiempos necesarios para realizar un cambio de herramientas. De esta forma, el efecto de los cambios de herramientas en los procesos de ensamblaje es mayor que en los procesos de mecanizado.

Debido a los factores señalados, la automatización de las operaciones de mecanizado se encuentra más desarrollada. Por un lado, resulta más fácil justificar económicamente el uso de máquinas que, aunque realicen operaciones especificas, sirven para tratar un amplio conjunto de piezas. Por contra, el equipamiento necesario para las operaciones de ensamblaje en muchas ocasiones debe ser reemplazado casi en su totalidad para distintas familias de productos.

Por otro lado, la programación de robots resulta más compleja en las operaciones de ensamblaje, ya que deben especificarse todos los movimientos a realizar. Cualquier incertidumbre puede llevar al fracaso del programa. Serán necesarios distintos sensores para poder percibir el entorno y poder interactuar adecuadamente con él, complicando por tanto la programación.

[Gottschlich, *et al.*, 1994] recoge las conclusiones del *IEEE Technical Committee on Assembly and Task Planning* sobre el estado de la planificación del ensamblaje hasta la fecha. Se recoge como objetivo global la sintetización automática de un programa detallado para ser ejecutado en un sistema de ensamblaje, a partir de una descripción en alto nivel del producto a ser montado. Se distinguen dos partes principales en la planificación. Por un lado, la planificación del ensamblaje (*Assembly Planning*) considera la determinación del conjunto de operaciones necesarias para ensamblar el producto a partir de sus componentes. En este nivel, las operaciones de ensamblaje vienen especificadas a partir de las piezas y submontajes involucrados. Por otro lado, la planificación a nivel de tareas (*Task Planning*) se encarga de la traducción a instrucciones de robot para llevar a cabo dichas operaciones. Como se ha comentado, se deberán tener en cuenta para ello diversos factores asociados al entorno de fabricación, incluyendo la capacidad para reaccionar ante las incertidumbres relacionadas con las tolerancias.

Para la automatización de la planificación de los procesos de ensamblaje, es necesario abordar una serie de aspectos técnicos [Homem de Mello and Lee, 1991], :

- Representación de productos: se deben obtener modelos para representar las formas y relaciones geométricas entre las distintas partes que forman el producto, así como otros aspectos no geométricos (ligaduras entre partes, tratamientos químicos...). Tales modelos deben permitir razonar sobre las posibles operaciones de montaje que pueden obtenerse.

- Representación de planes de montaje: de gran importancia en el desarrollo de los planificadores. Se han seguido diferentes metodologías: grafos dirigidos, grafos *And/Or*, condiciones de establecimiento de conexiones y relaciones de precedencia entre otras.

- Corrección y completitud del proceso de planificación: en la generación de los planes de montaje sólo deben aparecer aquellos que sean correctos, y no debe faltar ninguno factible.

- Eficiencia del proceso de planificación: al resultar la planificación muy costosa en general, se suelen buscar enfoques que reduzcan la computación necesaria. En muchos casos se sacrifica la completitud. Es importante que en ese proceso no se pierdan los mejores planes.

- Selección de planes de ensamblaje: debido al gran número de planes posibles que se pueden dar, incluso para un número pequeño de piezas, es necesario encontrar

técnicas que simplifiquen su tratamiento, ya que una enumeración de todos los planes posibles resulta prohibitiva. Para ello se han usado dos enfoques, uno cualitativo, considerando reglas para eliminar planes que incluyan tareas difíciles o submontajes delicados; y el otro cuantitativo, usando funciones de mérito (coste de recursos, tiempo real necesario, dificultad de ejecución...) que permitan su ordenación según los criterios elegidos.

- Integración con programas CAD: dado que son cada vez más usados para diseñar la geometría de los productos, deben ser tenidos en cuenta como entrada a los generadores de modelos de ensamblajes.

- Integración con planificadores de tareas y movimientos: para facilitar la programación de manipuladores a partir de los planes de ensamblaje.

En el resto del capítulo se ofrecen más detalles de aquellos elementos que más influyen en la planificación del ensamblaje. Por un lado, se describen métodos para representar tanto a los productos a ensamblar, con vistas a su análisis para la generación de planes de montaje mostrando distintas técnicas usadas para resolver los distintos aspectos involucrados. Se realiza una clasificación de los tipos de planes de montaje que pueden ofrecerse. A continuación se trata la representación de secuencias de ensamblaje, que es uno de los aspectos que pueden influir más en la construcción de un buen planificador, debido a la repercusión en el coste computacional. Seguidamente se ofrece una muestra de la generación de secuencias de ensamblaje, donde se identifican por un lado las operaciones de montaje teniendo en cuenta las restricciones internas de entre las piezas que forman el producto, y se definen las restricciones entre las operaciones de montaje, de precedencia y de completitud. A continuación, se exponen distintos criterios utilizados para la selección de secuencias de ensamblaje. Finalmente, en la última sección se describe el modelo propuesto en esta tesis para la selección de secuencias de ensamblaje, basado en la resolución de un problema de planificación que incorpora los principales factores que influyen en el tiempo total del ensamblaje del producto en un sistema genérico con múltiples máquinas de ensamblaje.

2.2 Representación de productos

Tal como se indicó antes, es necesario disponer de una representación del producto a ensamblar. El modelo del producto debe reflejar todos los detalles esenciales requeridos por el planificador para poder obtener todas las secuencias de ensamblaje factibles. Nótese que ello incluye toda la información necesaria para poder razonar sobre la factibilidad de los planes de ensamblaje. Así, de la geometría de los distintos componentes del producto deben extraerse las relaciones existentes entre las partes.

[Bourjault, 1984] usó dos tipos de grafos en su modelo relacional: el grafo de contactos y el grafo de conexiones. Los nodos del grafo de contactos se corresponden con cada pieza del producto, y existe una arista por cada contacto existente entre cada

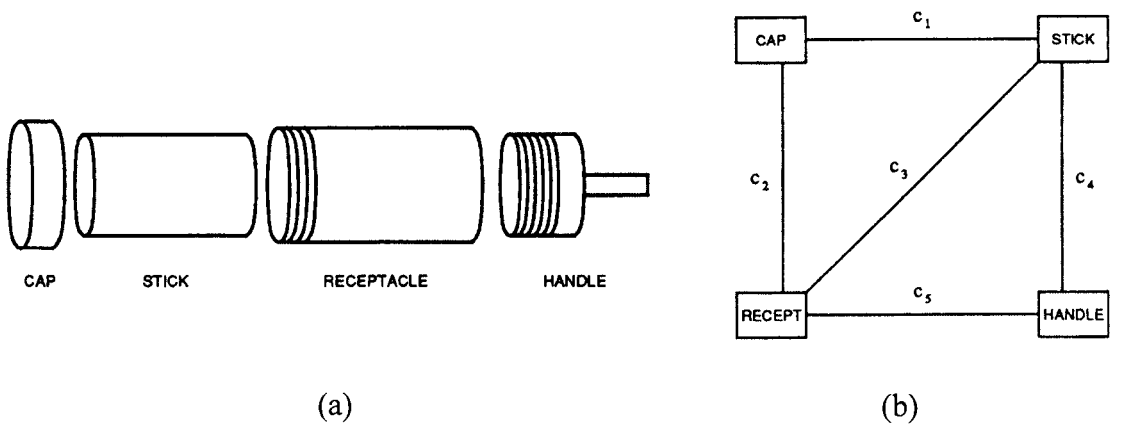


Figura 2.1: Producto ejemplo de Homem de Mello y su grafo de conexiones.

par de piezas. En el grafo de conexiones, las aristas se corresponden con lo que Bourjault llamó "*liaisons fonctionelles*", en donde los contactos pueden ser reales o *virtuales*. Los contactos virtuales informan de la imposibilidad de desplazar una pieza en una dirección debido a la presencia de otra en el camino, aunque directamente no estén en contacto. El mismo tipo de representación ha venido siendo usado por diferentes autores. En la Figura 2.1 se representa el grafo de conexiones para un ejemplo de producto tomado de [Homem de Mello and Sanderson, 1991a]. El modelo relacional diseñado por estos autores incluye distintos tipos de relaciones entre componentes que no se muestran en la figura, y que son usadas para la generación automática de las secuencias de ensamblaje factibles.

Otros autores utilizan representaciones de tipo jerárquico [Boneschanscher, 1993] [Chakrabarty and Wolter, 1997] [Moradi, *et al.*, 1997], de forma que permiten una reducción en la complejidad al estudiar los posibles planes de ensamblaje.

En los modelos anteriores, se consideran piezas sólidas rígidas. Con piezas flexibles, el tratamiento de las operaciones de ensamblaje debe ser diferente [Wolter and Kroll, 1996]. Por un lado, puede haber operaciones en piezas individuales. Con piezas rígidas, la mayoría de las representaciones de planes identifican las operaciones como relaciones entre dos piezas diferentes. Con piezas flexibles es habitual manipular las piezas en diferentes etapas. Para la mayoría de planificadores, no existen estados intermedios en las operaciones de montaje, sino que éstas se identifican mediante sus estados inicial y final.

2.3 Clasificación de secuencias de ensamblaje

La mayoría de los planificadores tienen limitaciones en cuanto al tipo de planes que pueden producir. [Wolter, 1992] ofrece una clasificación de los mismos:

Un plan se dice que es *secuencial* si puede ser descompuesto en un conjunto de operaciones tales que cada operación involucra el movimiento de un conjunto de piezas

a lo largo de una trayectoria común. Un plan secuencial podría ser ejecutado por un robot *a una mano*. El producto de la Figura 2.2 (a) requiere para su ensamblaje el movimiento de dos de sus piezas a lo largo de dos trayectorias diferentes, por lo que se dice que se necesita un plan de montaje no secuencial.

Un plan es *monótono* si ninguna operación separa ningún par de piezas que estuvieran en sus posiciones relativas finales entre sí, y todas las operaciones dejan todas las piezas movidas en sus posiciones relativas finales a alguna pieza no movida. De esta manera, se excluyen en los planes monótonos las operaciones que dejan las piezas en posiciones temporales, tal como ocurriría en ensamblaje de la Figura 2.2 (b).

Un plan es *coherente* si durante el proceso de ensamblaje, todas las operaciones añaden una pieza o submontaje a otro componente estableciendo al menos una conexión entre ambos. El producto de la Figura 2.2 (c) muestra un producto que necesita un plan no coherente para su montaje. Para realizar tal tipo de operaciones suele utilizarse un dispositivo de fijación que mantenga las posiciones relativas de los dos componentes a unir con el tercero.

Un plan es *lineal* si sólo puede moverse una pieza en cada instante, es decir, cada operación añade una sola pieza a un submontaje. Al no formarse más de un submontaje intermedio, la ejecución de las operaciones de los planes lineales es estrictamente

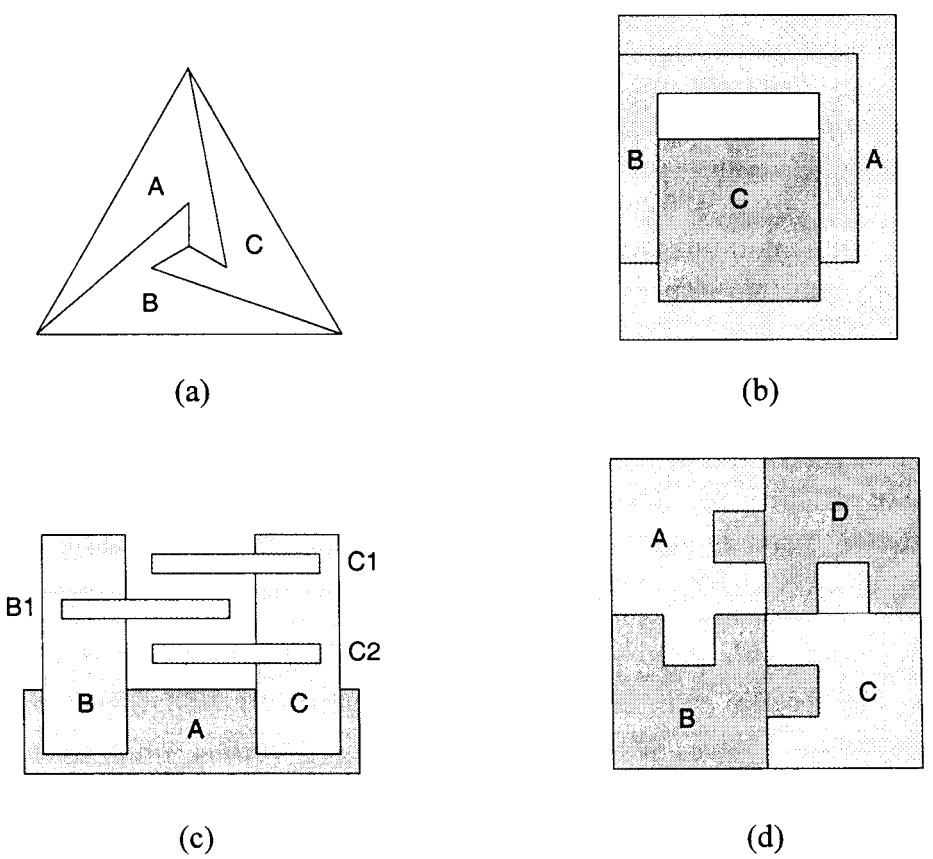


Figura 2.2: Ensamblajes bidimensionales que no pueden ser construidos por un plan (a) secuencial, (b) monótono, (c) coherente, (d) lineal. (Wolter)

secuencial, es decir, no pude haber más de una operación ejecutándose en paralelo. Aparte de que muchos planificadores sólo consideran la generación de planes secuenciales, para algunos productos, como el de la Figura 2.2 (d), se necesitan planes no lineales para su ensamblaje.

La mayoría de planificadores generan planes coherentes, monótonos y no lineales, como por ejemplo los descritos en [Homem de Mello and Sanderson, 1991b] y en [Baldwin, *et al.*, 1991]. [Wilson and Rit, 1991] y [Wolter, 1988] presentan planes lineales, y en [Tsao and Wolter, 1993] se describe un planificador de secuencias no lineales y no monótonas.

2.4 Representación de secuencias de ensamblaje

Una representación de las secuencias de ensamblaje factibles para un producto debe ser completa y correcta, es decir, sólo debe incluir las secuencias factibles, y no debe olvidarse ninguna valida. La elección de tal representación para un planificador puede ser crucial, debido entre otros factores a los distintos grados de complejidad que pueden obtenerse, la cantidad de memoria y tiempo necesario para su generación y almacenamiento, y al coste de la evaluación de las secuencias de ensamblaje a partir de la representación. Además, en tales representaciones suele incluirse información adicional asociada al proceso de ensamblaje, típicamente restricciones de tipo temporal o espacial.

Una secuencia de ensamblaje puede representarse a partir de una lista ordenada de operaciones de montaje o de estados de ensamblaje. Por tanto, el conjunto de secuencias de ensamblaje factibles para un producto podría representarse como un conjunto de listas ordenadas, correspondiendo cada una de ellas a una secuencia de montaje. Sin embargo, debido a que muchas secuencias comparten subsecuencias comunes, es conveniente la utilización de representaciones más compactas.

A continuación se detallan las representaciones básicas más usadas. Pueden englobarse en dos tipos de representación: explícita, a través de distintos tipos de grafos, e implícita, indicando las condiciones que debe satisfacer cualquier secuencia de ensamblaje factible. [Homem de Mello and Sanderson, 1991a] realiza un estudio detallado de tales representaciones, incluyendo procedimientos para realizar la transformación de unas representaciones a otras. Otros autores añaden información específica del proceso de ensamblaje. Por ejemplo, [Wolter, 1988] incluye restricciones sobre las trayectorias que pueden ser usadas en las operaciones de ensamblaje, y [Wilson, 1992] introduce restricciones de bloqueo entre las piezas en su representación NDGB (*non-directional blocking graph*).

2.4.1 Grafos de estados

[Bourjault, 1984] y [De Fazio and Whitney, 1987] usaron grafos de estado para la representación de secuencias en sus planificadores. Un estado venía definido por el

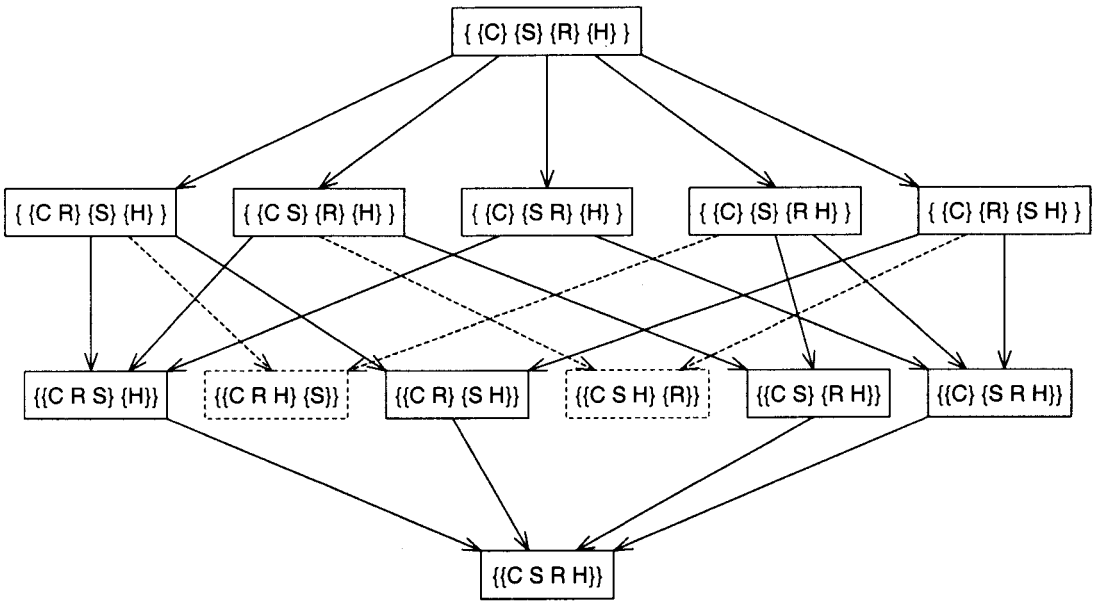


Figura 2.3: Grafo de estados para el producto de la Figura 2.1.

conjunto de conexiones que habían sido establecidas hasta el momento entre las diferentes piezas que forman el producto. Esto equivale también al conjunto de submontajes formados en cada momento, suponiendo que en los nodos del grafo se representan las situaciones iniciales y finales de cada operación de ensamblaje. Las operaciones se corresponden entonces con los arcos que unen a los nodos correspondientes a los estados inicial y final. La Figura 2.3 muestra el grafo de estados correspondiente al producto de la Figura 2.1.

Se asume en la representación anterior que las operaciones de ensamblaje son ejecutadas secuencialmente, y que en cada operación de montaje son ensamblados exactamente dos componentes, sean piezas o submontajes. Se asume también que, cuando se forma un submontaje, todas las conexiones entre sus piezas son establecidas, y que así permanecerán durante el resto del proceso de ensamblaje.

Una secuencia válida equivale a un camino desde el nodo raíz del grafo de estados hasta el nodo objetivo, correspondiente al producto completamente ensamblado. En la representación final han sido eliminados aquellos estados que, correspondiendo a estados estables, no pueden dar lugar a secuencias de montaje válidas (en trazo discontinuo en la Figura 2.3).

Los grafos de estado pueden ser usados también para representar secuencias no monótonas, de manera que un estado vendría representado por las conexiones establecidas. Una conexión ya establecida podría ser rota por una operación posterior.

2.4.2 Grafos *And/Or*

El grafo *And/Or* es la estructura estándar para representar problemas que pueden descomponerse en subproblemas independientes o con pocas interacciones entre ellos

[Nilsson, 1980]. [Homem de Mello, 1989] fue el primero en utilizarlo para representar planes de ensamblaje, de manera que las descomposiciones se refieren a los distintos submontajes que forman el producto completo.

En esta representación se asumen las mismas condiciones que para los grafos de estado: cada operación de ensamblaje realiza la unión de dos componentes, y en la unión de las piezas correspondientes a un submontaje, todos los contactos entre ellas quedan establecidos.

En la representación mediante grafos *And/Or*, cada nodo *Or* se corresponde con un submontaje estable, siendo el nodo raíz el que hace referencia al producto completo, y los nodos hoja a las piezas individuales. Cada nodo *And* se corresponde con la tarea de montaje que une los submontajes de sus nodos hijos produciendo el submontaje de su nodo padre. En el grafo *And/Or* vienen representadas únicamente las tareas factibles, de manera que tampoco aparecerán aquellos submontajes que, aun siendo estables, no pueden dar lugar a una secuencia de ensamblaje válida, manteniendo las conexiones entre sus piezas.

Una solución particular en el grafo *And/Or*, llamada por Homem de Mello *árbol de ensamblaje*, es un árbol en el grafo *And/Or* cuya raíz es el nodo correspondiente al producto completo y cuyas hojas son los nodos correspondientes a las piezas. Un árbol de ensamblaje representa pues un *plan de ensamblaje*, en donde se especifican todas las operaciones necesarias para ensamblar el producto completo, así como las restricciones de precedencia entre ellas. A partir de un plan de ensamblaje podrán obtenerse distintas secuencias de ensamblaje, para lo cual deberán tenerse en cuenta las restricciones asociadas a los recursos compartidos que se utilicen. La Figura 2.4 ilustra un ejemplo de esta representación, correspondiente al producto ejemplo de la Figura 2.1 [Homem de Mello, 1989], en la que los nodos *And* se han omitido. En cualquier caso, y tal como originalmente fue representado el grafo *And/Or* por sus autores, las operaciones de ensamblaje quedan claramente identificadas por los *hiperarcos And*, que conectan a los submontajes resultantes de la unión con los de partida. Cuando de un nodo salen varios hiperarcos, cada uno de éstos representa un conjunto diferente de dos componentes que pueden ser unidos para formar el submontaje asociado con el nodo en cuestión, y por extensión, una operación de ensamblaje alternativa para formar el submontaje.

Esta representación ofrece varias ventajas. Por un lado, el número total de nodos es considerablemente menor cuando el número de piezas de los productos se hace mayor. Cada operación de ensamblaje y cada submontaje vienen representados una sola vez, a diferencia de lo que ocurría en los grafos de estado. De hecho, en [Homem de Mello and Sanderson, 1990] y [Wolter, 1992] se muestra que esta estructura es más eficiente en la mayoría de los casos que otras estructuras enumerativas. Otra ventaja importante es que muestra de forma explícita la posibilidad de que las operaciones de ensamblaje puedan ejecutarse en paralelo. Tal circunstancia no es observable en la representación mediante grafos de estados.

El uso de grafos *And/Or* se ha convertido en un referente para la representación de todos los planes de ensamblaje factibles. Puede ser obtenido estudiando el problema opuesto, el desensamblado, pero suponiendo las restricciones del ensamblado para las

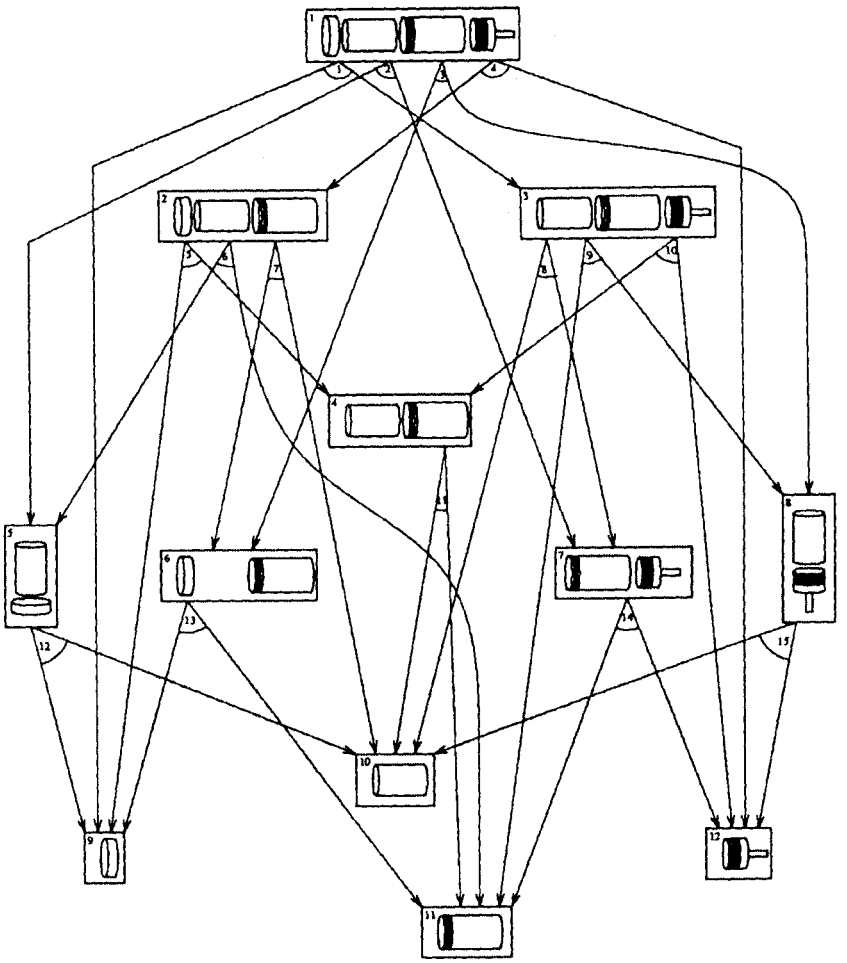


Figura 2.4: Grafo *And/Or* para el producto de la Figura 2.1.

operaciones. La mayoría de los planificadores automáticos trabajan con esta estrategia. El resultado es una representación adecuada para un enfoque dirigido hacia el objetivo.

2.4.3 Representaciones implícitas

El conjunto de todas las secuencias de ensamblaje factibles puede venir detallado a partir del conjunto de condiciones que cualquier secuencia factible debe satisfacer, dando lugar a representaciones implícitas. En muchos casos puede obtenerse una representación más compacta que mediante las representaciones explícitas, aunque puede ser a costa de sacrificar la corrección, de manera que pueden existir secuencias no factibles que satisfagan las condiciones.

[Bourjault, 1984] introdujo las *condiciones de establecimiento* de conexiones como un conjunto de funciones lógicas que indican los estados desde los cuales puede establecerse cada una de las conexiones sin impedir el montaje final del producto. Representando un estado a partir de un vector binario[1] L-dimensional, siendo L el

[1] o lógico, indicando el valor *true* que la conexión correspondiente está establecida y *false* que no lo está

número total de conexiones entre las piezas, sea $\Xi_i = \{\underline{x}_1, \underline{x}_2, ..., \underline{x}_{K_i}\}$ el conjunto de estados desde los que la conexión *i-ésima* puede establecerse. La condición de establecimiento de la conexión *i-ésima* viene dada por la función lógica

$$F_i(\underline{x}) = F_i(x_1, x_2, ..., x_L) = \sum_{k=1}^{K_i} \prod_{l=1}^{L} \gamma_{kl} \tag{2.1}$$

donde la suma y el producto son las operaciones lógicas *OR* y *AND* respectivamente, y γ_{kl} es x_l si la componente *l-ésima* de $\underline{x}_k$ es *true* o $\overline{x}_l$ si la componente *l-ésima* de $\underline{x}_k$ es *false*. Las expresiones resultantes pueden ser simplificadas utilizando las reglas del álgebra de Boole, y considerando los vectores que no representan estados de ensamblaje y a aquellos estados que no pueden darse en un plan de ensamblaje factible. De esta forma, las condiciones de establecimiento para el ejemplo de la Figura 2.1 vendrían dadas por las expresiones

$$\begin{aligned}
F_1(x_1, x_2, x_3, x_4, x_5) &= \overline{x}_1 \cdot \overline{x}_4 + \overline{x}_1 \cdot x_2 + \overline{x}_1 \cdot x_5 \\
F_2(x_1, x_2, x_3, x_4, x_5) &= \overline{x}_2 \cdot \overline{x}_5 + x_1 \cdot \overline{x}_2 + \overline{x}_2 \cdot x_4 \\
F_3(x_1, x_2, x_3, x_4, x_5) &= \overline{x}_3 \\
F_4(x_1, x_2, x_3, x_4, x_5) &= \overline{x}_1 \cdot \overline{x}_4 + \overline{x}_4 \cdot x_5 + x_2 \cdot \overline{x}_4 \\
F_5(x_1, x_2, x_3, x_4, x_5) &= \overline{x}_2 \cdot \overline{x}_5 + x_4 \cdot \overline{x}_5 + x_1 \cdot \overline{x}_5
\end{aligned}$$

Por otro lado, [De Fazio and Whitney, 1987] utilizaron *relaciones de precedencia* como otra forma de representación implícita del conjunto de secuencias de ensamblaje factibles. Dos tipos de relaciones de precedencia pueden definirse: las relaciones de precedencia entre el establecimiento de una conexión y los estados del proceso de ensamblaje, y las relaciones de precedencia entre el establecimiento de una conexión y el de otras conexiones.

En el primer tipo, la notación $c_i \rightarrow S(\underline{x})$ indica que el establecimiento de la conexión *i-ésima* debe preceder a cualquier estado que satisfaga la función lógica $S(\underline{x})$. Para representar combinaciones lógicas de relaciones de precedencias se puede usar una notación más compacta. Así, $c_i + c_j \rightarrow S(\underline{x})$ significa $(c_i \rightarrow S(\underline{x})) \vee (c_j \rightarrow S(\underline{x}))$. Realizando el mismo tipo de simplificaciones que para las condiciones de establecimiento, las relaciones de precedencia entre el establecimiento de una conexión y los estados del proceso de ensamblaje para el ejemplo de la Figura 2.1 vendrían dadas por las expresiones

$$\begin{aligned}
&c_1 \rightarrow x_2 \cdot x_5 \qquad c_1 \rightarrow x_2 \cdot x_3 \\
&c_2 \rightarrow x_1 \cdot x_4 \qquad c_2 \rightarrow x_1 \cdot x_3 \\
&c_3 \rightarrow x_1 \cdot x_2 \qquad c_3 \rightarrow x_4 \cdot x_5 \\
&c_4 \rightarrow x_3 \cdot x_5 \qquad c_5 \rightarrow x_3 \cdot x_4
\end{aligned}$$

En las relaciones de precedencia entre el establecimiento de una conexión y el de otras, se usa la notación $c_i < c_j$ para indicar que la conexión *i-ésima* debe realizarse antes que la conexión *j-ésima*, y $c_i \leq c_j$ para indicar que el establecimiento de la conexión *i-ésima* debe preceder o ser simultáneo al establecimiento de la conexión *j-ésima*. También puede usarse una notación más compacta para expresar la combinación de varias relaciones de precedencia. Así, $c_i < c_j \cdot c_k$ equivale a $(c_i < c_j) \wedge (c_i < c_k)$, mientras que $c_i + c_j < c_k$ equivale a $(c_i < c_k) \vee (c_j < c_k)$. Para el ejemplo de la Figura 2.1, las relaciones de precedencia entre el establecimiento de distintas conexiones vendrían dadas por la expresión

$$\begin{aligned}&\left((c_1 \leq c_2 + c_5) \vee (c_3 \leq c_2 + c_5) \vee (c_4 \leq c_2 + c_5)\right) \wedge \\ &\left((c_2 \leq c_1 + c_4) \vee (c_3 \leq c_1 + c_4) \vee (c_5 \leq c_2 + c_5)\right)\end{aligned}$$

2.5 Generación de secuencias de ensamblaje

Para la obtención del conjunto de todos los planes de ensamblaje factibles se han seguido distintas estrategias. Inicialmente, a través de planificadores interactivos en los que un experto introducía la introducción necesaria, y últimamente a través de planificadores automáticos que extraen la información a partir de modelos geométricos y relacionales del producto.

La identificación de las operaciones de ensamblaje se acomete mediante el análisis de la estructura del producto, y puede conducir al conjunto de todos los planes de ensamblaje factibles. El número de ellos crece exponencialmente con el número de piezas, y depende de otros factores, tales como la forma en que las partes individuales están interconectadas en el ensamblaje completo, es decir, la estructura del grafo de conexiones. De hecho, este problema ha sido probado como NP-completo tanto en el caso bidimensional [Kavraki and Kolountzakis, 1995] como en el tridimensional [Kavraki, *et al.*, 1993] [Wilson, *et al.*, 1995].

[Bourjault, 1984] usó el grafo de conexiones para obtener las condiciones de establecimiento a través de las respuestas del diseñador acerca de la posibilidad de establecer cada conexión en función de si las demás conexiones han sido o no establecidas previamente. [De Fazio and Whitney, 1987] propusieron un conjunto de preguntas alternativas de manera que el número de ellas se reduce considerablemente, permitiendo así su aplicabilidad a problemas con mayor número de piezas. Sin embargo, la reducción en el número de preguntas venía acompañada de un incremento en la dificultad de responder a esas cuestiones.

Uno de los primeros planificadores automáticos fue XAP/1 [Wolter, 1988], pero únicamente consideraba planes lineales. Este sistema tampoco estaba enlazado con ningún modelo geométrico del producto. Tenía como entrada una lista de piezas, una lista de posibles trayectorias de inserción para cada pieza y una lista de restricciones de precedencia de las operaciones de ensamblaje en función de la trayectoria de inserción. Además, el enfoque estaba orientado hacia la optimización del plan de montaje, con lo

que no generaba todas las secuencias factibles. Las soluciones no sólo incluía el orden en el que las piezas eran unidas, sino también las trayectorias de inserción de cada pieza.

[Homem de Mello and Sanderson, 1991b] presentan un planificador automático que genera el conjunto de todas las secuencias de ensamblaje factibles. Partiendo del grafo de conexiones, utilizaron un método basado en considerar el proceso opuesto al ensamblaje, la descomposición de cada submontaje en dos componentes, procediendo de forma recursiva hasta llegar a las piezas individuales. Para cada submontaje, se consideran todas las particiones posibles en el grafo de conexiones. En la Figura 2.5 aparecen todas las particiones del grafo de conexiones correspondiente al ejemplo de la Figura 2.1. Para cada descomposición, se verifica si la operación de ensamblaje correspondiente es factible desde el punto de vista geométrico, mecánico y de estabilidad de los submontajes intermedios. En tal análisis, se usa un modelo relacional del producto en el que se recogen las características geométricas de las piezas, así como el tipo de contacto entre las piezas. El resultado es el grafo *And/Or* correspondiente a todos los planes de ensamblaje factibles para el producto considerado. El enfoque correspondiente a considerar el problema del ensamblaje a partir del problema inverso, el desmontaje, será usado en la mayoría de los planificadores posteriores, ya que facilita el razonamiento automático sobre el conjunto de operaciones factibles para obtener un plan de ensamblaje completo.

[Henrioud and Bourjault, 1991] describen su planificador LEGA, escrito en PROLOG, el cual guarda similitudes con el método desarrollado por Homem de Mello. El enfoque es interactivo, pero introducen nuevas restricciones, de tipo estratégico, que reducen drásticamente el número de árboles de ensamblaje generados en la etapa del modelado del producto.

[Wilson, 1992] describe el planificador GRASP, un sistema que genera secuencias de ensamblaje directamente de un modelo geométrico del producto a ensamblar. Para ello, detecta el conjunto de contactos entre las piezas y construye el grafo *And/Or*

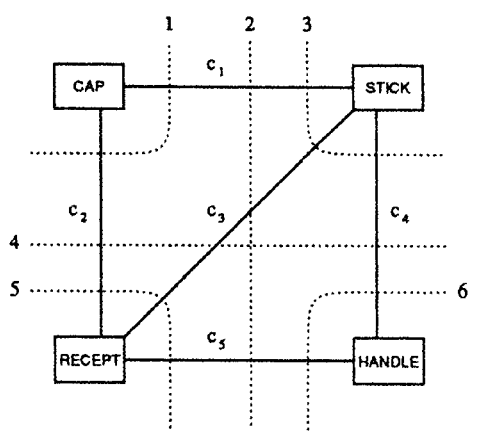


Figura 2.5: Particiones posibles para el producto de la Figura 2.1.

para el montaje del producto a partir del análisis de los posibles movimientos a nivel local y global de sus piezas y de la estabilidad de los submontajes resultantes. Su modelo de representación de las restricciones de bloqueo entre las piezas, NDBG, es usado por varios planificadores posteriores, ya que permite recoger de manera muy eficiente las propiedades geométricas del producto.

Enfoques parecidos han sido usados en planificadores más recientes. Uno de los planificadores a destacar es el sistema *Archimedes*, construido en los laboratorios Sandia. En [Ames, *et al.*, 1995] y [Kaufman, *et al.*, 1996] se realiza una descripción del mismo en su segunda versión. Incluye distintos módulos que permiten interactuar con el diseñador. Por un lado, facilitan al usuario la introducción de distintos tipos de restricciones asociadas al proceso. [Jones and Wilson, 1996] y [Jones, *et al.*, 1997] recogen una amplia relación de los diferentes tipos de restricciones que se suelen definir en los procesos de ensamblaje. Por otro lado, el sistema *Archimedes* permite mostrar una simulación del plan de montaje obtenido, tomando como entrada una célula de ensamblaje y los dispositivos de fijación necesarios. En [Calton, 1999] se describen los cambios que se han llevado a cabo en el sistema para mejorar algunas de las deficiencias observadas.

Otro planificador basado en los mismos principios es STAAT, realizado en la Universidad de Stanford [Romney, *et al.*, 1995]. Se trata de un sistema automático para generar a partir de un modelo geométrico del producto bien el conjunto de todas las soluciones, o bien una secuencia que cumpla un determinado criterio, para lo cual ofrecen distintas medidas de complejidad de las soluciones.

2.6 Selección de secuencias de ensamblaje

Una vez obtenido el conjunto de todas las secuencias de ensamblaje factibles, la siguiente consideración es la selección del plan óptimo de montaje, seleccionado del conjunto de todos los planes de ensamblaje factibles. Se han usado diversidad de criterios para escoger un plan óptimo, que pueden encuadrarse en dos enfoques alternativos, uno cualitativo, en el que se usan reglas para eliminar planes que incluyan tareas difíciles de ejecutar, y otro cuantitativo, en el que se usan funciones de mérito para evaluar la bondad de los planes de ensamblaje. Varias de estas propuestas se recogen en distintos capítulos de [Homem de Mello and Lee, 1991], dedicado monográficamente al tema.

La mayoría de los criterios planteados tienen como objetivo la simplificación del plan de ensamblaje, por ejemplo evitando movimientos complicados y estados delicados o inestables, minimizando las tareas poco productivas, como cambios de dispositivos de fijación o reorientación de piezas.

Por ejemplo, en [Wolter, 1988] se ponderan diferentes criterios: la manipulabilidad de los submontajes, evitando la realización de operaciones complejas con piezas de difícil manipulación; la complejidad de los mecanismos de fijación, evitando llevar muchas piezas pesadas que necesiten dispositivos de fijación complejos; y la direccio-

nabilidad, minimizando el número de direcciones diferentes desde las que las operaciones pueden efectuarse.

En [De Fazio, *et al.*, 1990] se introdujeron criterios que incluían la minimización de reorientaciones en los submontajes intermedios y del requerimiento de dispositivos de fijación. En un trabajo más reciente [De Fazio, *et al.*, 1999], utilizan como medida de la complejidad de las operaciones de montaje el número de grados de libertad cinemáticos completados en cada por cada movimiento de ensamblaje y el número de contactos fijados por cada movimiento de ensamblaje.

En [Henrioud, 1989] y [Henrioud and Bourjault, 1991] se propone la ayuda de un experto para evaluar la complejidad operativa y logística (geométrica, de estabilidad y requerimiento de materiales), y utilizar restricciones de tipo estratégico para seleccionar el mejor árbol de ensamblaje, por ejemplo imponiendo submontajes intermedios o agrupando piezas de formas y tamaños similares que puedan ser manipuladas por el mismo dispositivo de agarre, disminuyendo así el número de cambios de herramientas.

[Homem de Mello and Sanderson, 1990] propusieron asignar pesos a los hiperarcos del grafo *And/Or* dependientes de la complejidad de las tareas y de la estabilidad de los submontajes intermedios, y usar algoritmos genéricos de búsqueda tales como el AO* para obtener el mejor plan. Por otro lado, en [Homem de Mello and Sanderson, 1991c] propusieron algunos criterios de optimización basados en la maximización del número de diferentes secuencias de ensamblaje asociadas al plan de montaje y en la maximización del posible paralelismo (simultaneidad) en la ejecución de las tareas de montaje.

En otro ámbito, en [Holland, *et al.*, 1992] se utiliza como criterio de minimización el tiempo de ensamblaje para una célula de ensamblaje flexible específica, y para pequeños lotes de productos. El algoritmo propuesto está dirigido a obtener en línea una secuencia de montaje de manera que para cada ciclo va intentando mejorar la solución, teniendo en cuenta además los posibles errores que van produciéndose.

El sistema STAAT [Romney, *et al.*, 1995] permite optimizar algunas medidas como linealidad, apilabilidad, facilidad de extracción de piezas individuales y facilidad en la planificación de movimientos como la inserción de piezas.

Por otro lado, el sistema *Archimedes* [Kaufman, *et al.*, 1996] [Calton, 1999] permite especificar al usuario la medida sobre la que quiere optimizar la solución. Una amplia colección de tales medidas se recoge en [Jones and Wilson, 1996]. Además, sobre el mismo sistema, se ha integrado un módulo para la estimación de costes [Calton and Peters, 1999] que facilita los procesos de decisión entre distintas estrategias de fabricación.

Muchas de las medidas de complejidad usadas para juzgar la calidad de las secuencias de ensamblaje son discutidas en [Goldwasser and Motwani, 1999]. Además, se demuestra que para muchas de las medidas consideradas el problema es NP-completo, incluyendo el caso de restricciones de precedencia de tipo *And/Or*.

2.7 Modelo propuesto

El presente trabajo se centra en la selección óptima de secuencias de ensamblaje. Para ello, se parte del conjunto de secuencias de ensamblaje factibles que ha debido generarse previamente. Concretamente, se supone definido el grafo *And/Or* correspondiente al producto en cuestión. En esta representación, cada nodo *Or* se corresponde con un submontaje, siendo el nodo raíz el que hace referencia al producto completo, y los nodos hoja a las piezas individuales. Cada nodo *And* se corresponde con la tarea de montaje que une los submontajes de sus nodos hijos produciendo el submontaje de su nodo padre. Normalmente, se suelen omitir los nodos *And* en las representaciones gráficas, viniendo representadas las operaciones de montaje a través de los hiperarcos correspondientes (véase la Figura 2.6).

Por mayor claridad en la representación, el problema puede ser estudiado a través del problema inverso, el desmontaje del producto completo en sus piezas individuales. Esto facilita su resolución mediante un enfoque dirigido hacia el objetivo. Evidentemente, si queremos obtener el plan de ensamblaje, las tareas deben referirse a la operación de ensamblaje correspondiente, con lo cual, para obtener la solución correspondiente al proceso de ensamblaje sólo bastará con invertir la solución obtenida. Según todo lo anterior, en adelante, al referirnos a la precedencia de tareas y el secuenciamiento de las mismas, lo haremos en base al desmontaje del producto.

La representación mediante grafos *And/Or*, como se indicó en el apartado 2.4.2, expresa no sólo el conjunto de operaciones de ensamblaje que pueden conducir al montaje completo del producto, sino también dos tipos de restricciones entre esas operaciones.

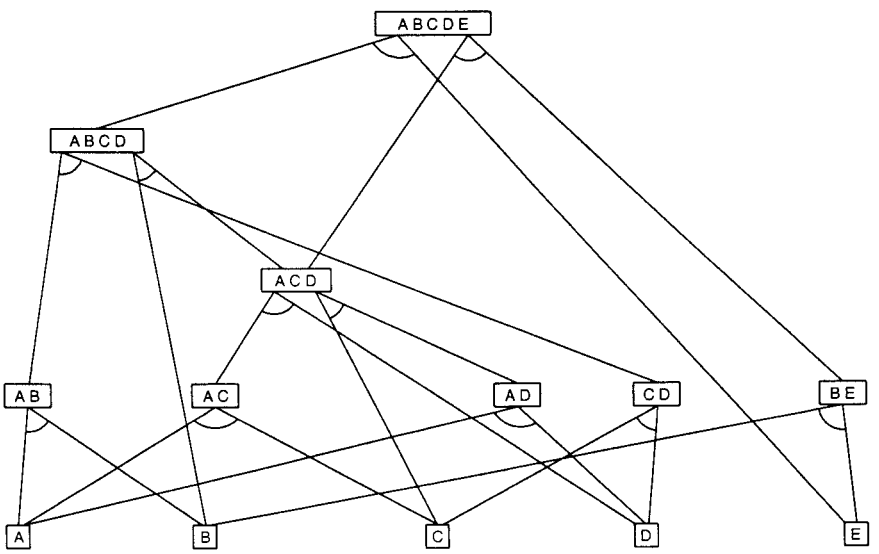


Figura 2.6: Grafo *And/Or* para el producto ABCDE.

Por un lado, una solución completa viene dada por un árbol de ensamblaje, por lo que para obtener una solución correcta, las operaciones de ensamblaje deben pertenecer todas a un mismo árbol dentro del grafo *And/Or*. A las restricciones de este tipo las podríamos llamar de integridad o completitud. Obsérvese que el número de operaciones de ensamblaje necesarias para completar el montaje de un producto de N piezas es $N-1$, dado que cada operación de montaje une dos componentes, sean submontajes o piezas individuales.

Por otro lado, de la estructura acíclica del grafo *And/Or* se extraen las relaciones de precedencia entre las tareas que pueden formar parte de un mismo plan de ensamblaje. Así, la operación de ensamblaje asociada a un nodo *And* determinado se deberá ejecutar con anterioridad a aquellas otras que se correspondan con los nodos *And* por debajo de aquél. Por esta razón, llamaremos también árbol de precedencias (de tareas) a un árbol de ensamblaje del grafo *And/Or*.

La representación mediante grafos *And/Or* refleja también de manera muy clara la posibilidad de que varias operaciones de ensamblaje se ejecuten en paralelo, otro aspecto que será muy tenido en cuenta en el presente trabajo. Así, las tareas correspondientes a la formación de un submontaje (todas las que estén por debajo del nodo *Or* correspondiente) podrían ejecutarse en paralelo con aquellas otras involucradas en la formación de otro submontaje disjunto a aquél, es decir, sin ninguna pieza en común.

La mayoría de criterios recogidos en el apartado anterior buscaban la minimización de elementos problemáticos para la ejecución del proceso de ensamblaje, como movimientos complicados e inestables, y tareas no productivas como los cambios de dispositivos de fijación y la reorientación de piezas. Incluso en algunos casos, forzaban la consecución de planes lineales. Al no considerar todo el conjunto de secuencias posibles, pueden pasarse por alto algunas que podrían ser interesantes desde el punto de vista del rendimiento global del proceso, en las que la posibilidad de ejecutar en paralelo el ensamblaje de distintas partes del producto tendrían especial importancia.

El criterio escogido en este trabajo es la *minimización del tiempo total del ensamblaje* del producto. Para ello se van a considerar a continuación aquellos aspectos que influyen directamente en él. Se va a suponer un sistema con múltiples estaciones de trabajo, capaces de realizar cada una de ellas una operación de ensamblaje en cada momento. A ellas nos referiremos habitualmente como máquinas de ensamblaje o simplemente máquinas. De esta forma, el proceso de ensamblaje podrá completarse ejecutándose varias operaciones de ensamblaje de forma simultánea.

El problema resultante a resolver va a ser un problema de planificación que vendrá definido, aparte de por el propio grafo *And/Or*, por la información asociada a cada tarea de ensamblaje en el grafo *And/Or*. Esa información se compone de una duración estimada para la tarea de montaje, supuesto además que se realiza en una máquina concreta y con una configuración determinada de la misma. Tal configuración se va a representar a través del uso de una *herramienta* de agarre para la manipulación de piezas. Se utilizará la siguiente notación: para una tarea de ensamblaje T, su duración se denotará mediante *dur* (T), $M(T)$ representará la máquina usada para realizar la tarea T, y $H(T)$ la herramienta utilizada en $M(T)$ para ejecutar T.

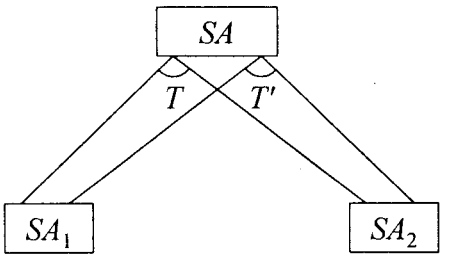


Figura 2.7: Las tareas *T* y *T'* sólo se diferencian en los recursos que utilizan.

La estimación de la duración y recursos necesarios para cada tarea deberá haberse realizado bien durante el proceso de construcción del grafo *And/Or*, o bien con posterioridad, mediante un análisis en el que se habrán debido considerar tanto las características geométricas de las piezas y submontajes como las propiedades de los componentes del sistema de ensamblaje, fundamentalmente los dispositivos de agarre en los robots y dispositivos de fijación para los submontajes.

En este trabajo se ha considerado una sola combinación $< dur\,(T), M\,(T), H\,(T) >$ para cada tarea de ensamblaje *T*. Sin embargo, la ampliación a más opciones para poder ejecutar una determinada tarea no ofrece dificultad. Basta con considerar que se trata de otra posible tarea para construir el mismo submontaje a partir también de los mismos componentes de partida. Se trataría pues de ampliar el grafo *And/Or* con nodos *And* adicionales cuando se tengan distintas posibilidades (máquina-herramienta) para construir un submontaje determinado a partir de dos componentes dados (véase la Figura 2.7).

Aparte de la propia duración de las tareas, hay otros factores que pueden influir en el tiempo total del proceso de ensamblaje. Por un lado, las duraciones de los cambios de herramientas, que como se indicó en la sección 2.1, son del mismo orden que las de las operaciones de ensamblaje. Obsérvese que la influencia de los cambios de herramientas en el tiempo total del ensamblaje no depende de las ejecuciones de las tareas, sino del orden de ejecución de las mismas. Así, una buena solución deberá buscar que las operaciones que usen una misma herramienta aparezcan contiguas en la secuencia, ahorrando así los cambios de herramientas que se necesitarían en caso contrario. El tiempo necesario para instalar en la máquina *M* la herramienta *H* estando previamente instalada *H'* se denotará como $\Delta_{cht}(M,H,H')$. Este es el retardo mínimo que deberá existir entre la finalización de una operación de ensamblaje que utilice la herramienta *H'* en la máquina *M* y el comienzo de otra que se ejecute con posterioridad en la misma máquina utilizando la herramienta *H*.

Debe señalarse que cualquier cambio en la configuración de una máquina entre la ejecución de una tarea y otra puede modelarse de esta forma. Podría pensarse incluso que, para la ejecución de una determinada tarea, inicialmente se necesite una determinada herramienta y acabe la misma con otra. El tratamiento que se daría a esta situación es la creación de una determinada configuración virtual (con una herramienta imaginaria), y definir como nulos los tiempos de cambio de herramienta desde la inicial a la virtual y

desde la virtual a la final, y englobar el tiempo necesario para el cambio de herramienta en la duración de la tarea de ensamblaje.

Otro factor que puede influir de forma directa en el tiempo total del ensamblaje es el tiempo necesario para transportar las distintas piezas y submontajes hacia las máquinas de ensamblaje. Es preciso observar que este factor puede ser muy dependiente de la configuración del sistema de ensamblaje. A pesar de que éste puede estar compuesto por distintos dispositivos, como alimentadores de piezas, cintas transportadoras o robots móviles, de distinta naturaleza y posiblemente de forma combinada, pueden realizarse algunas consideraciones útiles. Si el sistema de transporte está bien dimensionado y se tiene una perfecta planificación del mismo a la hora de ejecutar el plan de ensamblaje, podría esperarse que cuando se necesite un componente en una determinada máquina para realizar una operación de ensamblaje, el componente esté allí presente. Esto no debería suponer ningún inconveniente para las piezas individuales, ya que, previamente a su primera utilización en la formación del producto, saldría de su lugar de almacenamiento en el momento adecuado y llegaría a su lugar de destino con el tiempo suficiente para poder ser utilizado sin provocar ningún retraso en el sistema. En el presente modelo se supone un sistema de transporte de piezas óptimo, por lo que el tiempo correspondiente al transporte de las piezas no es tenido en cuenta por no influir en el tiempo total de ensamblaje del producto.

Sin embargo, no puede decirse lo mismo para los submontajes intermedios. Téngase en cuenta que un submontaje intermedio se forma durante el proceso de ensamblaje, y en una máquina determinada. Si a continuación ése submontaje es requerido en otra máquina para realizar otra operación, tal operación deberá esperar a que el submontaje esté presente, para lo cual deberá trasladarse el submontaje desde la máquina en que se formó. Este retardo sí será tenido en cuenta en el modelo propuesto. El tiempo necesario para trasladar el submontaje SA desde la máquina M hasta la máquina M' se denotará como $\Delta_{mov}(SA, M, M')$. Ése será el retardo mínimo entre las operaciones que involucran a un submontaje dado. Para el ejemplo de la Figura 2.8, entre la finalización de la tarea T y el comienzo de T' existirá un retardo mínimo igual a $\Delta_{mov}(SA, M(T'), M(T))$ (cero si $M(T) = M(T')$). Obsérvese que ese retardo tendrá influencia en el tiempo total de

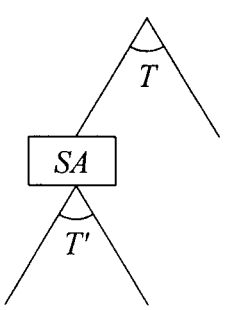


Figura 2.8: Si T y T' se realizan en distintas máquinas, SA deberá ser trasladado desde una a otra.

ensamblaje dependiendo del orden de ejecución de las tareas de montaje. Así, una buena secuencia de ensamblaje debería intentar aprovechar el transporte de un submontaje hacia la máquina que lo requiere para realizar otra operación en la misma.

Aparte de los datos reseñados para cada tarea, denotaremos como *prec*(T) a aquella tarea que aparece como predecesora inmediata a la tarea T en el árbol de precedencias. Refiriéndonos al grafo *And/Or* completo, representaría al conjunto de tareas predecesoras inmediatas, que necesariamente deben pertenecer a distintos árboles de ensamblaje. Del mismo modo, *suc*(T) se corresponde con las tareas sucesoras inmediatas en el árbol de precedencias. Por otro lado, *sa*(T) denotará al submontaje que se forma al realizar la tarea de ensamblaje T. Para el ejemplo mostrado en la Figura 2.8, $sa(T') = SA$.

El problema que resulta del modelo propuesto es la planificación de las operaciones de ensamblaje necesarias para realizar el montaje completo del producto, de acuerdo con las restricciones que se han indicado, y teniendo en cuenta los retardos que pueden producirse debido a cambios de configuración en las máquinas y al transporte de submontajes intermedios entre las máquinas del sistema. Así, deberá decidirse cuál es el conjunto de tareas que se debe ejecutar (árbol de ensamblaje), y los tiempos de comienzo y finalización de las distintas tareas que formen parte del plan de montaje. A esos tiempos los denotaremos como *begin*(T) y *end*(T) respectivamente, para una tarea genérica T.

La complejidad del problema resultante puede apreciarse a partir de las fuertes restricciones que posee. Por un lado, [Gillies and Liu, 1995] demostraron que la planificación de tareas con restricciones de tipo *And/Or* para la minimización del tiempo total es NP-completo. Por otro lado, en [Lawler, *et al.*, 1985] se muestra la correspondencia de la planificación de tareas donde se tienen retardos producidos por cambios en las configuraciones en las máquinas dependientes de la secuencia con el problema del viajante de comercio, problema bien conocido por su NP-completitud.

Capítulo 3

Resolución mediante Algoritmos Genéticos

3.1 Introducción

En este capítulo se estudia la resolución del problema propuesto mediante algoritmos genéticos, una técnica de optimización que se engloba dentro de un conjunto de técnicas no deterministas que han venido a denominarse con el nombre de *metaheurísticas*. Estas técnicas están orientadas principalmente hacia la resolución de problemas combinatorios y con múltiples extremos locales (problemas multimodales), como el problema que nos ocupa, utilizando estrategias de tipo probabilista. Dentro de estas técnicas también se encuentran, aparte de los algoritmos genéticos, otras muy utilizadas para resolver estos problemas, como los algoritmos de recocido simulado (*simulated annealing*) [Kirkpatrick, *et al.*, 1983] y los métodos de búsqueda tabú [Glover, 1989] [Glover, 1990].

A diferencia de los algoritmos genéticos, los métodos anteriores se basan en los métodos de búsqueda local manteniendo un único punto de búsqueda. Para conseguir una buena efectividad, utilizan distintas estrategias para no quedar atrapados en los óptimos locales. Así, los algoritmos de recocido simulado permiten la aceptación de soluciones peores que la actual durante el proceso de búsqueda, de acuerdo con una función de probabilidad. El nivel de aceptación decrece con el número de iteraciones del algoritmo y depende, además, de la magnitud del empeoramiento de la función objetivo.

La búsqueda tabú está caracterizada por utilizar estructuras de memoria que permiten almacenar los últimos movimientos realizados en la búsqueda, de manera que puedan recordarse para evitar caer de nuevo en los mismos extremos locales. Permitiendo además la realización de movimientos con un empeoramiento de la función objetivo, se pueden explorar otras regiones que puedan llevar hacia otros extremos locales o hacia el óptimo global.

Por el contrario, los algoritmos genéticos mantienen un conjunto de puntos sobre los que se permite continuar la búsqueda, de manera que, de forma simultánea, se pueden explorar distintas regiones prometedoras del espacio de búsqueda. La utilización de distintos tipos de operadores permite que la búsqueda se diversifique, desde lo local hasta lo global.

Los algoritmos genéticos, como otros algoritmos de tipo evolutivo, están basados en los principios biológicos de la evolución de las especies. Dentro de esta metodología, cada solución del problema es considerada como un individuo de la población, la cual va evolucionando a partir de ciertos operadores genéticos, que van creando nuevos individuos a partir de los existentes, con una tendencia a mejorar la población anterior.

Así como los componentes de esta metodología son comunes, existen aspectos dependientes del problema concreto que se quiere resolver que complican su aplicabilidad. Dentro de esos problemas se encuentran fundamentalmente el de la representación sintáctica de los individuos y la búsqueda de los operadores genéticos apropiados, ambos íntimamente relacionados.

Aquí serían factibles distintos enfoques, unos más desplazados hacia la fácil aplicación de operadores genéticos comunes, aunque aprovechando quizá poco conocimiento del problema, y otros más desplazados al punto de vista inverso, es decir, aprovechar todo lo posible la información reflejada en una solución, lo cual requerirá normalmente la utilización de operadores genéticos muy complejos.

Como se indicó en el capítulo anterior, el problema que se plantea resolver es el de la obtención de secuencias óptimas de ensamblaje de productos, partiendo de una representación implícita de todas las secuencias factibles mediante un grafo *And/Or*, el cual ha debido obtenerse previamente mediante el análisis de las piezas que componen el producto a ensamblar, su geometría y relaciones con el resto de piezas dentro del producto. El criterio de optimización tomado es el del tiempo total de ensamblaje, para lo cual se toman en consideración los recursos necesarios para la realización de cada tarea de ensamblaje, así como una estimación del tiempo requerido para la ejecución de las distintas operaciones de ensamblaje y los retardos asociados a los cambios de herramientas en las máquinas y al transporte de submontajes intermedios.

Parecidos problemas han sido abordados ya con cierto éxito con algoritmos genéticos, entre los que cabe destacar el problema del viajante de comercio (*Traveling Salesman Problem*, o TSP), y la secuenciación de trabajos en varias máquinas (*Job Shop Scheduling Problem*, o JSSP). El conjunto de soluciones a esos problemas resulta ser el conjunto de todas las permutaciones de los componentes individuales de que consta una solución (las ciudades en el problema del viajante y las operaciones en el problema de secuenciación de trabajos). La aplicación de estrategias de optimización

evolutivas a los problemas de secuenciamiento se ha convertido en un tema de investigación de gran interés. Existen múltiples referencias en las que se tratan problemas típicos de secuenciamiento (JSSP, TSP...) mediante algoritmos genéticos [Nakano and Yamada, 1991] [Starkwheather, *et al.*, 1991] [Syswerda, 1991] [Whitley, *et al.*, 1991] [Mattfeld, 1996].

El problema que aquí se trata resulta ser de una mayor complejidad, pues hay que añadir al problema de la decisión del orden en que se deben ejecutar las operaciones de ensamblaje la propia selección de operaciones entre todas las alternativas posibles. Así, un grafo *And/Or* incluye todos los planes posibles para realizar el ensamblaje de un producto, en donde un plan resulta ser un camino del grafo (árbol), y especifica un orden parcial de las operaciones que deben realizarse. Por otro lado, una operación de ensamblaje, que especifica los submontajes que debe unir para obtener otro submontaje, puede aparecer en más de un plan de ensamblaje. Así, para la obtención de una solución al problema (secuencia de operaciones) es necesaria la obtención de un plan de montaje que determine qué operaciones forman parte de la solución.

En este trabajo se proponen dos alternativas para la representación sintáctica de los individuos. La primera de ellas, una representación binaria, permite utilizar los operadores genéticos estándar, aunque su efectividad queda condicionada por la calidad de la información genética asociada a esa representación, que podría resultar deficiente. La segunda de las representaciones propuestas, la representación simbólica, está más cercana a la estructura de las soluciones del problema. En ambos casos, un individuo representa una secuencia de las tareas que forman la solución correspondiente, la cual deberá ser construida completamente mediante un algoritmo, el constructor de programas de ensamblaje, que tenga en cuenta las restricciones de precedencia entre las tareas y la utilización de recursos compartidos.

En la representación binaria, se enumeran implícitamente todas las posibles secuencias de tareas que conforman soluciones correctas, en el sentido de satisfacer todas las restricciones de precedencia impuestas por el grafo *And/Or*. Así, cada secuencia de tareas correcta tendrá asociada un identificador numérico, comprendido en el rango desde cero al número de secuencias correctas menos uno. Mediante una codificación binaria de los identificadores numéricos asociados a cada solución posible, pueden utilizarse los operadores genéticos típicos binarios, dando lugar a nuevos individuos correspondientes a soluciones siempre correctas. Es evidente que en la utilización de estos operadores no se aprovecha toda la información recogida en los individuos (que por otro lado no es toda la que puede obtenerse), ya que la secuencia de tareas ha sido transformada en un número entero que luego es tratado mediante una cadena de bits. Sin embargo tampoco puede descartarse una mínima influencia de la correlación entre secuencias de tareas e identificadores numéricos asociados. Así, dos individuos con representación numérica parecida (los bits más significativos son comunes) normalmente pertenecerán a una zona común del grafo *And/Or*, lo cual indicará que tendrán algunas tareas comunes, concretamente las más cercanas a la raíz. Teniendo en cuenta lo anterior, los resultados que se obtienen son relativamente satisfactorios.

Una representación de los individuos más natural es la propia secuencia de tareas (representación simbólica). Todavía no se refleja toda la información de la solución en el individuo, pero sí se utiliza más información que con la representación binaria. El problema está ahora en que los operadores genéticos a usar pueden no obtener individuos correctos si son operadores relativamente triviales. Por un lado, los operadores que conducen a nuevas permutaciones, similares a los usados en la literatura para los problemas del viajante y la secuenciación de tareas, deben incorporar algoritmos de reparación para hacer cumplir las restricciones de precedencia entre tareas. Por otro lado, la incorporación de una nueva tarea a un individuo, por ejemplo sustituyendo a otra, conllevaría el incumplimiento de solución válida, teniendo en cuenta que cada tarea se corresponde biunívocamente con los submontajes que ensambla y el submontaje obtenido en la operación de ensamblaje.

Para la representación simbólica se han considerado dos familias de operadores: una formada por algoritmos encaminados a cambiar el orden de las tareas de los individuos para formar otros, y otra familia de operadores para modificar tanto el orden como la composición de tareas en los individuos. La primera familia incluye operadores que buscan nuevas secuencias localmente en un plan de montaje predeterminado, el correspondiente a los cromosomas padre. Estos operadores son similares a los usados para resolver los problemas de TSP y JSSP en la literatura. La otra familia de operadores introduce nuevas tareas en la solución, reemplazando a otras para mantener la validez de los cromosomas. Estos operadores tienen el propósito de buscar secuencias en otros planes de montaje. Estos últimos operadores, los más novedosos, deben involucrar necesariamente al grafo *And/Or* para la construcción de soluciones correctas. Se estudian distintas alternativas, desde aquellos algoritmos que, al introducir una nueva tarea en el plan intenta minimizar el número de tareas adicionales que deben introducirse para obtener un plan correcto, hasta aquellos otros que, a partir de una nueva tarea a introducir, intenta, con el menor coste computacional posible, construir un plan nuevo basándose mínimamente en el anterior.

Como era de esperar, los resultados que se obtienen con el modelo de representación simbólica son mejores que con la representación binaria, producto de que los operadores genéticos definidos aprovechan mejor la información genética que almacenan los individuos.

El resto del capítulo se estructura de la siguiente forma. En la sección 3.2 se presenta el esquema general de los algoritmos genéticos, así como los conceptos más importantes asociados a la técnica. En la sección 3.3 se detallan los aspectos de los modelos de representación escogidos en el trabajo. En la sección 3.4 se presentan los operadores genéticos utilizados para los dos modelos de representación. En la sección 3.5 se incluyen resultados sobre distintos problemas, describiendo la influencia sobre los mismos de los modelos y operadores genéticos usados. Por último, en la sección 3.6 se señalan las conclusiones y posibles trabajos futuros a desarrollar basados en el trabajo realizado.

3.2 Esquema general de los algoritmos genéticos

3.2.1 Principios básicos

La idea de los algoritmos genéticos está basada en imitar determinados procesos observados en la naturaleza asociados a la evolución de las especies biológicas. Según éstos, las poblaciones evolucionan según los principios de selección natural y de supervivencia de lo mejor. Los individuos con mayor capacidad de adaptarse a su entorno tienen mayores posibilidades de sobrevivir y de reproducirse, mientras que los menos adaptados tenderán a desaparecer. Las características de adaptación suelen estar recogidas en los genes de los individuos, de manera que la combinación de buenas características de individuos bien adaptados puede producir una descendencia incluso mejor adaptada.

Un algoritmo genético simula estos procesos tomando una población inicial de individuos y aplicando operadores genéticos en cada reproducción. En términos de optimización, cada individuo de la población es codificado mediante una cadena o *cromosoma* y representa una posible solución a un problema dado. La adaptación (*fitness*) de un individuo es evaluada con respecto a la función objetivo dada para el problema. A los individuos o soluciones con altos valores de adaptación se les otorga mayores probabilidades para reproducirse, intercambiando con otros parte de su información genética en un mecanismo de reproducción denominado *cruce*. Esto produce nueva descendencia con características comunes a ambos padres. Otro mecanismo básico empleado en la reproducción es la *mutación*, mediante la cual se alteran algunos genes en el individuo. Los nuevos individuos generados reemplazarán a algunos miembros de la población actual. Este ciclo consistente en la evaluación, selección y reproducción de individuos se hace repetir hasta que se encuentre una solución satisfactoria al problema o se alcance algún criterio de terminación del algoritmo (ver Figura 3.1).

3.2.2 Componentes de un algoritmo genético

Un algoritmo genético es un algoritmo probabilista adaptativo que contiene los siguientes elementos:

- Una representación genética para las soluciones del problema a resolver, que será referida como cromosomas o individuos.
- Una forma de generar una población inicial de individuos.
- Una función de adaptación que permita evaluar a los individuos según la medida a optimizar

```
Algoritmo genético básico
  generar una población inicial
  evaluar la adaptación de los individuos de la población
  mientras no se alcance el criterio de terminación
    seleccionar los individuos para la recombinación
    obtener nuevos individuos a partir de los padres seleccionados
    evaluar la adaptación de los individuos hijos
    reemplazar parte o toda la población con los nuevos individuos
  fmientras
```

Figura 3.1: Algoritmo genético básico.

- Un mecanismo probabilista para seleccionar a los individuos mejor adaptados para su reproducción
- Un conjunto de operadores genéticos que permitan hacer evolucionar a la población creando nuevos individuos que reemplazarán a otros
- Un criterio de terminación del algoritmo
- Unos valores asociados a los distintos parámetros del algoritmo

Nótese que el mecanismo de selección de padres para su reproducción y los operadores genéticos que generan nuevos individuos llevan incorporadas componentes probabilistas, mientras que en otros aspectos el algoritmo trabaja de forma determinista. En los siguientes subapartados se describen con mayor detalle los distintos componentes señalados, así como otros aspectos de interés.

3.2.3 Representación de los cromosomas

Generalmente, cada solución potencial del problema a resolver debe poder ser representada a través de un conjunto de variables con valores en un alfabeto determinado. En los algoritmos genéticos tradicionales [Holland, 1975], el alfabeto es binario, {0, 1}, y los individuos vienen representados por cadenas de bits de longitud fija. Este tipo de codificación puede ser usado para dar una representación de las soluciones a cualquier tipo de problema.

En algunos casos, se trata de establecer cuál debe ser el tamaño de los cromosomas para que todas las soluciones posibles puedan venir representadas. Si algún parámetro sólo puede tomar un número exacto de valores en un conjunto finito, la codificación puede complicarse. Si el número de valores posibles no es potencia de 2, el dimensionamiento de los cromosomas se hará con holguras, de manera que si estaba reservado un código binario para cada valor posible a representar, habrá secuencias binarias que no se correspondan con valores a representar. Ante esto caben en principio

dos soluciones. Por un lado, a tales cromosomas se les puede asignar un valor de la función de adaptación muy malo. Otra opción es que algunas soluciones vengan representadas dos veces, para que los cromosomas siempre representen a alguna solución válida. En cualquier caso, ambas soluciones pueden perturbar el funcionamiento del algoritmo genético. Normalmente, la resolución de tales problemas de codificación se realiza en la evaluación de la función de adaptación.

En otros casos, es necesario realizar una discretización de los posibles valores que pueden tomar las variables en el problema real. La exactitud de esa discretización determinará el tamaño de los cromosomas, y por extensión la complejidad del problema de búsqueda resultante.

Sin embargo, a veces puede ser más conveniente usar otro tipo de codificación, como por ejemplo una representación numérica (mediante valores enteros o reales), más acorde con el problema que quiere resolverse, o una representación simbólica en los problemas en donde la ordenación de elementos forma parte esencial de la estructura de las soluciones.

3.2.4 Función de adaptación (*fitness*)

La relación de los cromosomas con la bondad de las soluciones que representan viene establecida por la función de adaptación, que generalmente será la función objetivo a optimizar. Para realizar el cálculo correspondiente, en general se necesita un proceso de descodificación de los cromosomas hacia una representación más natural de las soluciones del problema. Muchas veces, el proceso de descodificación queda inmerso en el propio cálculo de la función de adaptación.

La función de adaptación debe ser relativamente rápida de calcular, ya que está inmersa en un proceso en el que deben evaluarse los individuos creados en cada generación, lo cual puede suponer un gran número de evaluaciones, del orden de decenas de miles o incluso superior. Así, si el cálculo exacto de la función objetivo supone un gasto computacional importante, puede ser útil disponer de alguna medida aproximada que atenúe ese efecto y permita obtener unos tiempos de ejecución razonables.

3.2.5 Mecanismos de selección de padres

Uno de los factores que más pueden influir en el proceso de búsqueda de un algoritmo genético es el mecanismo de selección de padres para su reproducción. Se trata de determinar por un lado las probabilidades de selección de cada individuo presente en la población, según las cuales se determinarán cuáles de ellos pasan a la fase reproductiva. Por otro lado, se aplican técnicas de muestreo sobre la población para obtener los citados individuos. En la literatura se han propuesto tres mecanismos básicos para efectuar la selección:

Selección proporcional

Propuesto por [Holland, 1975], este mecanismo asocia a cada individuo una probabilidad de ser seleccionado para la reproducción proporcional al valor de la función de adaptación. Así, en una población con N individuos, donde el individuo *i-ésimo* tiene un valor de la función de adecuación f_i, su probabilidad para ser seleccionado se calcularía como

$$p_i = \frac{f_i}{\sum_{j=1}^{N} f_j} \tag{3.1}$$

Nótese que se está asumiendo la maximización de la función de adaptación. En el caso de problemas de minimización, debe buscarse una función inversa a la función objetivo para poder ser usada como función de adaptación. Tradicionalmente, se usaba el *método de la ruleta* [Goldberg, 1989] para realizar la selección de los individuos, consistente en generar una ruleta en la que a cada individuo se le asigna una fracción igual a su probabilidad de selección. Se generarán tantos números aleatorios, tomados de la distribución uniforme $U(0,1)$, como nuevos individuos deban crearse para la nueva generación, simulando el funcionamiento de la ruleta para la selección de los individuos. Obsérvese que un individuo podrá ser seleccionado varias veces. Idealmente, el número de veces que debería ser seleccionado cada individuo debería ser $n_i = N \cdot p_i$. [Baker, 1987] propone un algoritmo que garantiza que el número de ejemplares seleccionados de cada individuo se encuentra acotado entre $\lfloor n_i \rfloor$ y $\lceil n_i \rceil$.

La selección proporcional presenta algunos problemas. Por un lado, puede favorecer la aparición de *superindividuos*, con un valor de la función de adaptación muy superior al del resto de individuos, pudiendo hacerse dominadores en la población. Esto puede hacer que el algoritmo converja de forma prematura hacia un extremo local. Por otro lado, cuando los valores de la función de adaptación son muy similares entre sí, el algoritmo no es capaz de discriminar adecuadamente entre los diferentes individuos. Para resolver ambos problemas, suelen utilizarse técnicas de *escalado* de manera que se obtienen unos valores para la función de adaptación más apropiados para realizar la selección. [Michalevicz, 1992] recoge los métodos más usuales:

- *Escalado lineal*: la función de adaptación se reajusta a partir de una expresión lineal de la forma $f_i' = a \cdot f_i + b$, siendo a y b dos constantes elegidas de manera apropiada. Los valores de f' pueden resultar negativos, con lo que habría que realizar un ajuste adicional.

- *Truncado sigma*: la función de adaptación se reajusta de forma lineal considerando la desviación estándar σ de los valores de adaptación de toda la población. Los nuevos valores se calculan mediante la expresión $f_i' = f_i - \left(\overline{f} - c \cdot \sigma\right)$. Típicamente se toma para c el valor 1. Nuevamente pueden resultar valores negativos para f', lo que requeriría un ajuste adicional.

- *Escalado exponencial*: la función de adaptación se recalcula mediante la expresión $f_i' = (f_i)^k$. El valor de k puede ser ajustado de forma dinámica, tal como propone [Michalevicz, 1992], de manera que sea mayor conforme avance el algoritmo. Esto conlleva una mayor presión selectiva para los individuos mejor adaptados a medida que el algoritmo evoluciona.

Los métodos anteriores resuelven el problema del escalado. Sin embargo, la presión selectiva que ejercen a lo largo del algoritmo no es constante, lo que mantiene los riesgos de una convergencia prematura del algoritmo. Otros métodos de selección permiten mantener una presión constante, como se verá a continuación.

Selección por *ranking*

Propuesto por [Whitley, 1987], la probabilidad de seleccionar a cada individuo es ahora proporcional a la posición que ocupa en la población, ordenados los individuos de manera ascendente según la función de adaptación. La probabilidad de ser seleccionado el individuo que ocupa la posición *i-ésima* suele tomarse a partir de la expresión

$$p_i = \frac{1}{N}\left(\eta^- + \left(\eta^+ - \eta^-\right)\cdot\frac{i-1}{N-1}\right) \tag{3.2}$$

siendo $\eta^+ + \eta^- = 2$. La forma de seleccionar a los individuos para su reproducción puede hacerse de manera similar a como se indicó para la selección proporcional.

La presión selectiva sobre los individuos en la población se controla ahora de manera más consistente, ya que la probabilidad de seleccionar al *i-ésimo* mejor individuo es siempre la misma, independientemente del valor que tenga la función de adaptación.

Selección por torneo

El método de selección por torneo selecciona el mejor individuo de entre k elegidos de manera aleatoria. Este proceso se repite hasta completar el número de individuos deseado para la reproducción. De acuerdo a este proceso, la probabilidad de seleccionar al individuo que ocupa la posición *i-ésima* en la población, ordenados de forma ascendente según la función de adaptación sería

$$p_i = \frac{i^k - (i-1)^k}{N^k} \tag{3.3}$$

Un tamaño de torneo típico es con $k = 2$, llamado torneo binario. Conforme mayor sea el valor de k, mayor presión selectiva se realiza sobre los mejores individuos. Nuevamente, un individuo podría ser seleccionado varias veces. El comportamiento es parecido a la selección por *ranking*, pero es más eficiente ya que no se necesita ordenar

los individuos en la población. Obsérvese además que las probabilidades de selección no deben ser calculadas.

3.2.6 Operadores genéticos

En un algoritmo genético se distinguen básicamente dos tipos de operadores: cruce y mutación. A continuación se describen los detalles más importantes de ambos, junto con una discusión sobre el efecto que realizan en el funcionamiento del algoritmo.

Operadores de cruce

El operador de cruce juega un papel fundamental en el funcionamiento del algoritmo genético. Su efecto es combinar información genética de dos individuos, llamados padres, para obtener nuevos individuos, hijos, que podrían mejorar su adaptación respecto a la de sus padres. Se distinguen varios tipos de operadores de cruce para la representación binaria:

- *Cruce simple o de un punto*: se selecciona aleatoriamente una posición interior de cruce, y se intercambian los segmentos situados al mismo lado entre los dos padres. En la Figura 3.2 se muestra un ejemplo de uso del operador.

- *Cruce múltiple o de n puntos*: representa una generalización del cruce simple, en el que se obtienen *n* puntos de cruce. Alternativamente se van intercambiando los segmentos de ambos padres para la construcción de los hijos. La Figura 3.3 muestra este efecto.

- *Cruce uniforme*: propuesto por [Syswerda, 1989], cada elemento del cromosoma hijo se escoge aleatoriamente de uno u otro padre.

Una característica común a los operadores de cruce es que los genes comunes en ambos padres son pasados directamente a los hijos. Así, no introducen diferencias en los genes comunes. La actuación de los operadores de cruce en solitario puede favorecer la convergencia prematura por la desaparición de variedad en los valores de los genes, por lo que será necesario el concurso de otros operadores que favorezcan la diversidad

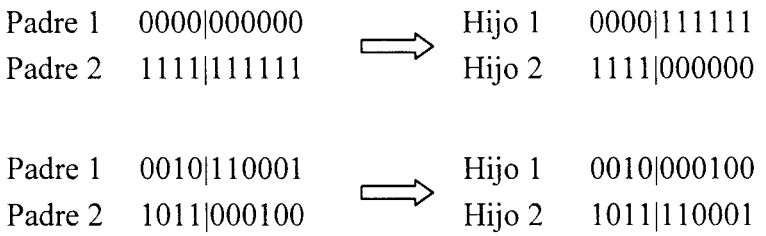


Figura 3.2: Operador de cruce simple.

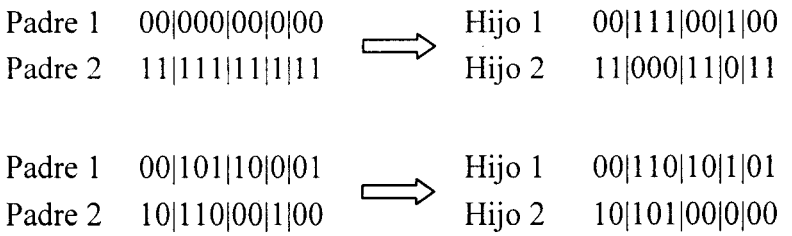


Figura 3.3: Operador de cruce múltiple.

genética en la población.

Operadores de mutación

El operador de mutación es usado para asegurar que todos los posibles cromosomas sean alcanzables en el proceso de búsqueda. Holland lo introdujo como un operador que realizaba una pequeña porción de búsqueda aleatoria en el algoritmo. El modo de actuar es cambiar alguno de los genes de algunos de los individuos presentes en la población. Normalmente, la probabilidad que se toma para que actúe el operador de mutación es muy pequeña. A diferencia de la búsqueda local que efectúan los operadores de cruce, la mutación permite ampliar la búsqueda hacia todo el espacio de soluciones del problema. Del mismo modo, puede observarse también que el operador de mutación permite reintroducir la información que ha podido perderse por la actuación del operador de cruce.

Otros tipos de operadores han sido propuestos en la literatura, aunque de menor relevancia. En muchos casos podrían verse como casos especiales de los operadores de cruce y mutación.

3.2.7 Esquemas de reemplazo

Conforme se van creando nuevos individuos, debe utilizarse algún mecanismo que mantenga el tamaño de la población, de forma que algunos o todos los individuos de la población sean sustituidos por los nuevos individuos generados. Fundamentalmente se han usado dos enfoques para realizar esta sustitución: el reemplazo *generacional*, en donde la población completa es renovada, y el reemplazo de estado estacionario (*steady-state*), en el que sólo unos pocos individuos (típicamente uno o dos) son sustituidos. Un tercer enfoque, el reemplazo *gradual*, generaliza los anteriores, de manera que se entiende como que una fracción de la población (*gap*) es renovada. En general, cuanto menor sea la fracción de la población que se renueva, más rápida puede ser la convergencia del algoritmo. Aunque esto puede resultar ventajoso, es necesario cuidar que no se trate de una convergencia prematura del algoritmo.

En el reemplazo generacional, no se garantiza que el mejor individuo permanezca en la población. Por esta razón, en problemas de optimización se suele utilizar un

enfoque *elitista*, de manera que los k mejores individuos (*k-elitismo*) se mantengan de una generación a la siguiente en la población. Habitualmente, se suele tomar $k = 1$. Con esto, se consigue que el mejor individuo generado durante toda la ejecución del algoritmo genético esté presente en la población final, siendo además el mejor individuo de ésta. Sin embargo, el mantenimiento del mejor individuo no implica necesariamente que el algoritmo se comporte mejor. Por un lado, sus hijos no necesariamente van a ser individuos con buenos valores de la función de adaptación. Por otro, se favorece la aparición de individuos dominantes en la población, lo que podría llevar a una convergencia prematura del algoritmo. A pesar de estos inconvenientes, los resultados parecen indicar una mejora en el comportamiento global de los algoritmos genéticos al aplicar la estrategia elitista.

En el reemplazo de estado estacionario o incremental, al sustituirse únicamente uno o dos individuos, es posible mantener a los mejores individuos en la población. Éste sería el caso en que los nuevos individuos generados reemplacen los peores de la población. Otras opciones de sustitución son el reemplazo aleatorio, en el que los individuos a sustituir son seleccionados de forma aleatoria, el reemplazo directo, en el que los nuevos individuos sustituyen a sus progenitores, o el reemplazo por torneo, en donde, seleccionados al azar un número de individuos, se escoge para ser sustituido al peor de ellos. En este último caso, también se garantiza la supervivencia del mejor individuo de la población.

Aparte de la ventaja de mantener a los mejores individuos, el reemplazo incremental también ofrece otra: los nuevos individuos generados están disponibles de inmediato para ser seleccionados para la reproducción. Para la resolución de problemas de optimización, donde el interés fundamental está en la obtención de la mejor solución posible, las propiedades anteriores pueden resultar útiles.

El reemplazo incremental puede provocar un coste computacional más elevado, si las probabilidades de selección de los individuos para su reproducción han de ser recalculadas cada vez que un nuevo individuo entra en la población. Sin embargo, este coste adicional no sería necesario si se utiliza el método de selección por torneo en lugar de la selección proporcional o por *ranking*.

Otro aspecto que es necesario cuidar en el uso del reemplazo de estado estacionario es la mayor posibilidad de que aparezcan individuos duplicados, lo que redundaría en un peor comportamiento del algoritmo genético al descender el grado de diversidad de la población, pudiéndose llegar a una convergencia prematura del algoritmo. Deberán pues ponerse medios para evitar que se introduzcan en la población individuos ya presentes en la misma.

3.2.8 Otros parámetros

Otros elementos de los reseñados en los apartados anteriores pueden también influir en la efectividad de los algoritmos genéticos.

Tamaño de la población

El número de individuos de la población debe ser escogido atendiendo fundamentalmente a dos criterios, el grado de diversidad genética de la población y la velocidad de convergencia del algoritmo.

Por un lado, la población debe contener un grado de diversidad genética lo suficientemente grande como para permitir la generación de cualquier solución al problema. Es fácil ver que con una población pequeña, el algoritmo podría converger de forma prematura al no cubrirse suficientemente los distintos aspectos genéticos de las posibles soluciones. Tal diversidad viene también influenciada por el tamaño del espacio de búsqueda de soluciones, ya que cuanto mayor sea el espacio de soluciones, mayor deberá ser el tamaño de los cromosomas, y por lo tanto, la información genética que deben albergar. Por tanto, se necesitará un tamaño de la población suficientemente grande como para satisfacer ese objetivo.

Por otro lado, la velocidad de convergencia del algoritmo puede verse reducida si la población es demasiado grande, ya que al ser necesario procesar muchos individuos en cada generación, el coste computacional necesario en la ejecución total del algoritmo puede verse incrementado de forma notable.

Aunque se han realizado distintos estudios teóricos para determinar el tamaño óptimo de la población, se ha comprobado que para la mayoría de los problemas los algoritmos genéticos se comportan bien con un tamaño de población entre 50 y 200 individuos.

Población inicial

En la mayoría de los casos, suele utilizarse una población inicial formada por individuos escogidos de manera aleatoria, con lo que se suele obtener un grado de diversidad adecuado. Sin embargo, otras alternativas pueden resultar interesantes. Por un lado, se podrían usar métodos que garanticen que todos los genes se encuentren presentes en la población. Por otro lado, se pueden utilizar métodos heurísticos que incorporen buenas soluciones al problema en la solución inicial, con el objetivo de ayudar a que el algoritmo genético encuentre mejores soluciones de forma más rápida.

Criterio de terminación

Para determinar cuándo debe finalizar la ejecución del algoritmo genético pueden usarse distintos criterios. Entre ellos podemos enumerar:

- Fijar un número máximo de evaluaciones de la función objetivo, o bien un máximo número de generaciones obtenidas.

- Finalizar la ejecución después de un número determinado de iteraciones sin que haya mejorado la solución.

- Finalizar la ejecución si no ha habido ninguna evaluación de individuos después de un número determinado de generaciones.

Los criterios anteriores pueden combinarse según las necesidades de la aplicación, pudiéndose incorporar otros que estimen la convergencia del algoritmo a través de alguna medida sobre la población.

3.2.9 Hibridación

Para muchas aplicaciones, la utilización de los algoritmos genéticos clásicos no resulta ser la mejor alternativa. De hecho, recientes resultados (teorema de *No Free Lunch* [Wolpert and Macready, 1995]) han mostrado que ningún algoritmo de búsqueda prevalece sobre la búsqueda aleatoria si no se utiliza ningún tipo de conocimiento del problema en la búsqueda. De esta forma, los algoritmos genéticos, que habían sido propuestos como un método de búsqueda mediante caja negra, pierden efectividad si la representación de los individuos y los operadores genéticos tienen poco que ver con la estructura de las soluciones del problema. [Davis, 1991] recoge los principios fundamentales que deben guiar la *hibridación* de los algoritmos genéticos, esto es, la incorporación del conocimiento que se tenga del problema a resolver en el algoritmo genético:

- Utilizar la representación más natural del problema a resolver, normalmente la que usa el experto en la definición del problema o en su resolución mediante otros métodos.

- Hibridizar donde sea posible: se deben incorporar al algoritmo genético aquellos aspectos positivos de los algoritmos conocidos. Si se conoce algún algoritmo determinista que ofrezca buenas soluciones al problema de manera rápida, puede utilizarse para generar la población inicial. Si además se utiliza una estrategia elitista, se asegura que el algoritmo genético no ofrecerá una solución peor que los algoritmos conocidos. Si se conocen algoritmos capaces de encontrar mejores soluciones a partir de una dada, normalmente mediante búsqueda local, podrían ser usados para introducir nuevos individuos que mejoren la población.

- Adaptar los operadores genéticos. Al tener una codificación propia para el problema a resolver, no pueden ser usados los operadores genéticos que fueron diseñados para una codificación binaria. Deben buscarse operadores específicos que funcionen de manera análoga a los convencionales. Así, deben buscarse operadores de cruce que sean capaces de mezclar información genética proveniente de dos individuos, y operadores de mutación que permitan introducir nuevo material genético en la población. En muchos casos, la tarea planteada no resulta fácil. Puede ser útil también incorporar como operadores específicos a aquellos métodos heurísticos que sean capaces de generar buenas soluciones de manera rápida. Es necesario observar que el uso abusivo de métodos de búsqueda local (*hill-*

climbing) para mejorar los individuos existentes podría acarrear una pérdida de información genética del algoritmo y la convergencia prematura del mismo.

3.2.10 Algoritmos genéticos para problemas con restricciones

Los algoritmos genéticos se aplican generalmente como técnicas de búsqueda no restringida, asumiendo que todos los individuos generados son factibles. Sin embargo, los problemas de optimización suelen venir acompañados de restricciones que deben ser satisfechas por las soluciones a los mismos. De acuerdo con la representación escogida para los individuos y los operadores genéticos asociados, podrían ser generados individuos no factibles, por lo tanto no asociados a ninguna solución válida del problema. El espacio de búsqueda podría pues dividirse en un subespacio conteniendo las soluciones factibles y otro correspondiente a los individuos no factibles. La proporción entre uno y otro subespacios determinará la facilidad con que el algoritmo genético podrá responder de forma efectiva ante los individuos no factibles generados. Así, una proporción grande de individuos no factibles puede limitar seriamente la operatividad de los algoritmos genéticos.

Una de las cuestiones a resolver es si en la población deben o no convivir individuos factibles y no factibles. Por un lado, el mantenimiento de individuos no factibles puede dificultar la búsqueda si se asume que es más probable obtener mejores individuos a partir de individuos factibles. Por otro, si se eliminan los individuos no factibles podría perderse diversidad en la población.

Una de las estrategias más habituales para responder ante la aparición de individuos no factibles es la *penalización* de los mismos a través de la función de evaluación, de manera que la penalización tenga en cuenta la magnitud de la violación de restricciones. En este sentido, resulta difícil la determinación de unos coeficientes de penalización adecuados. Esta estrategia permite mantener a los individuos no factibles en la población.

Otra opción es utilizar métodos de *reparación* de individuos no factibles, de manera que los individuos generados que no cumplan todas las restricciones son modificados para obtener una solución válida. Tampoco está claro el mecanismo que debería usarse para ello, ya que podrían ser posibles distintas soluciones factibles a partir de una no factible dada. Si no se tiene cuidado, se puede caer en una pérdida de diversidad genética en la población. Los métodos de reparación suelen ser efectivos cuando se combinan con métodos de representación de los individuos que tienen muy en cuenta las restricciones asociadas al problema a resolver. Otra ventaja de los métodos de reparación es que las funciones de evaluación son mucho más sencillas que cuando se incorporan funciones de penalización. El inconveniente con el que es posible encontrarse es que los operadores de reparación pueden resultar costosos computacionalmente, asunto que dependerá fuertemente del problema específico.

Por otro lado, si se quiere mantener a los individuos no factibles en la población, se podrían usar los métodos de reparación únicamente para el cálculo de la función de

adaptación del individuo, es decir, un individuo no factible tendría asociado otro factible que sería el que determine su adaptación en la población.

Ninguna de las estrategias anteriores es del todo satisfactoria, y otro enfoque que suele usarse es que los operadores genéticos sean diseñados de manera que produzcan individuos correspondientes a soluciones correctas, para lo cual se deberá disponer de una representación adecuada que permita la definición de tales operadores.

3.2.11 Fundamentos teóricos de los algoritmos genéticos

La mayoría de los trabajos sobre algoritmos genéticos se han concentrado en la búsqueda de reglas empíricas para obtener un buen funcionamiento del algoritmo. Existe una falta de teoría general que explique exactamente por qué funcionan estos algoritmos y cuál sería su rendimiento esperado.

Entre los estudios teóricos más desarrollados se encuentra la teoría sobre los *esquemas* propuesta por [Holland, 1975]. Los esquemas se corresponden con conjuntos de individuos en el espacio de soluciones, de manera que definen un patrón de similitud entre ellos.

Admitiendo una representación binaria de los cromosomas, si la longitud de los cromosomas es L, cada individuo pertenecería a 2^L esquemas. Un esquema podría definirse como una cadena de longitud L formada a partir del alfabeto {0, 1, #}, donde el símbolo # indicaría que el valor correspondiente a la posición que ocupe puede ser un 1 o un 0. Por ejemplo, el esquema 1#011 cubre a los cromosomas 10011 y 11011.

Si se asume que los individuos pertenecientes a un mismo esquema comparten características genéticas comunes, al evaluar a un individuo se estaría obteniendo también información relativa a los demás individuos que pertenecen a los mismos esquemas. De esta forma, el algoritmo no sólo procesa individuos sino también esquemas. En una población de N individuos, el número de esquemas que se procesan por cada generación es del orden $O(N^3)$ [Goldberg, 1989]. A esta propiedad se le conoce como *paralelismo implícito*.

La teoría de los esquemas, basada en la representación binaria, utiliza los conceptos de longitud y orden de un esquema. La *longitud de un esquema*, $\delta(H)$, se define como la distancia entre la primera y la última posiciones definidas. El *orden de un esquema*, $o(H)$, es el número de posiciones definidas en el esquema. Suponiendo que se utiliza el método de selección proporcional, reemplazo generacional y los operadores de cruce de un punto y mutación de un sólo gen, el teorema de los esquemas realiza una estimación del número de ejemplares que existirán de cada esquema en una generación a partir de los presentes en la generación anterior:

$$E(H,t+1) \geq E(H,t) \cdot \frac{f(H,t)}{\bar{f}(t)} \cdot \left(1 - p_c \cdot \frac{\delta(H)}{L-1}\right) \cdot (1-p_m)^{o(H)} \qquad (3.4)$$

En la expresión anterior, $E(H, t)$ representa el número de ejemplares correspondientes al esquema H en la población en la generación t, $f(H, t)$ es el valor medio de la función de

adaptación para los individuos del esquema H en la generación t, $\overline{f}(t)$ es el valor medio de la función de adaptación para toda la población en la generación t, y p_c y p_m representan las probabilidades de cruce y mutación respectivamente.

La primera conclusión que se extrae de la ecuación (3.4) es que aquellos esquemas cuyo valor medio de la función de adaptación en la población sea mayor tendrán más posibilidades de aumentar el número de ejemplares en la siguiente generación. Por otro lado, los operadores de cruce y mutación tienden a destruir los esquemas presentes en la población, pero son necesarios para obtener otros mejores. El grado de destrucción dependerá del orden y la longitud del esquema. Así, los esquemas de bajo orden, pequeña longitud y adaptación por encima de la media son favorecidos para la siguiente generación. Esto es lo que se conoce como hipótesis de los *bloques de construcción*. De acuerdo con esta hipótesis, el algoritmo genético va obteniendo buenas soluciones a partir de la combinación de información de pequeños trozos de individuos correspondientes a esos bloques de construcción. Así, el algoritmo genético tendrá un buen comportamiento si es posible combinar esquemas cortos para obtener soluciones mejores.

El teorema de los esquemas ha sufrido fuertes críticas [Altenberg, 1995] sobre las hipótesis y conclusiones que se derivan de ellas. Además, el teorema de *No Free Lunch* [Wolpert and Macready, 1995] echa por tierra la conclusión que obtuvo Holland sobre la superioridad de los algoritmos genéticos sobre la búsqueda aleatoria. La clave para que un algoritmo de búsqueda sea eficiente es que incluya algún conocimiento del dominio del problema a resolver. Para el caso de los algoritmos genéticos, éstos no podrían ser entendidos como una caja negra que nos resuelve cualquier problema, sino que para que lo haga de forma adecuada es necesario que el usuario del mismo incorpore en su estructura (representación de los cromosomas, operadores genéticos...) información importante sobre el problema a resolver.

3.3 Modelos de representación de soluciones

La aplicación de los algoritmos genéticos se enfrenta en primer lugar al importante problema de la selección de la representación adecuada para los individuos, que deberán corresponderse con soluciones al problema. Por un lado, una representación clásica binaria permitiría la fácil aplicación de operadores genéticos comunes, aunque la información genética transmitida podría resultar escasa. Por otro lado estaría una representación que se acerque más a lo que representa realmente una solución, en la que aparezcan directamente las tareas de ensamblaje, recursos utilizados, submontajes, tiempos, en definitiva, todo lo que pueda describir a una secuencia de ensamblaje. Este tipo de representación precisa de unos operadores genéticos específicos, pero podrían resultar más efectivos. Nos enfrentamos, por tanto, al problema de encontrar unos operadores genéticos adecuados que permitan obtener nuevos individuos legales, es decir, que cumplan las restricciones impuestas por el grafo *And/Or*. Esto no resulta ser una tarea fácil debido a la alta dependencia de unas tareas respecto a otras. Así, la

sustitución de una tarea por otra en una secuencia es imposible sin que haya que reformar la secuencia, añadiendo otras tareas y eliminando otras tantas. Esto es debido a que una tarea de ensamblaje viene definida biunívocamente por los submontajes iniciales y el submontaje final obtenido tras su ejecución, y su presencia en la solución implica a un número limitado de caminos posibles en el grafo *And/Or*, lo que conlleva el que solamente puedan aparecer unas tareas determinadas.

A continuación se presentan los dos modelos propuestos de representación de individuos. Como se verá, el modelo simbólico es usado como representación intermedia para el modelo binario en el proceso de descodificación. Se presentará pues en primer lugar el modelo de representación simbólica, en el contexto de dicho proceso de descodificación. Para el modelo binario, éste proceso se completará a partir de la descripción específica de la traslación desde la representación binaria a la simbólica.

3.3.1 Representación simbólica: secuencia de tareas

Asociado con la representación cromosómica de los individuos está el proceso de descodificación para obtener la solución. La Figura 3.4 muestra un esquema de cómo es descodificado un cromosoma para producir un programa de ensamblaje (*schedule*). En el caso de la representación binaria, un *constructor de secuencias* (*sequence builder*) transforma la representación binaria en simbólica, que representa una secuencia de tareas, cuyo orden es compatible con las restricciones impuestas por el grafo *And/Or*, es decir, las restricciones de precedencia de tareas y las asociadas a la construcción de planes de montaje factibles (selección de tareas). Así, no todas las secuencias de tareas constituyen una representación simbólica válida. El constructor de secuencias es descrito en la siguiente subsección.

Un constructor de programas (*schedule builder*) transforma la representación simbólica en un programa (*schedule*) de ensamblaje legal, teniendo en cuenta las restricciones de precedencia y los recursos compartidos que deben usarse (máquinas y herramientas). Esta traslación se realiza fácilmente debido a la simplicidad de esta representación. Así, las operaciones son programadas, siguiendo el orden de aparición en la secuencia, para ser ejecutadas en el tiempo más temprano posible, respetando las restricciones de precedencia y de recursos compartidos fijadas por las operaciones precedentes en la secuencia. El resultado podría visualizarse a través de un diagrama de Gantt, y permite calcular la función objetivo (el tiempo total de ensamblaje).

Obsérvese que, dependiendo de la asignación de recursos a tareas y sus duraciones, a diferentes cromosomas simbólicos, y por extensión, también a diferentes cromosomas binarios, podrían corresponderles un mismo programa de ensamblaje. Ello ocurrirá cuando haya tareas que no compartan los mismos recursos y puedan ser ejecutadas de forma simultánea. Así, en el ejemplo de la Figura 3.4, las tareas OP-9 y OP-10 podrían intercambiar sus posiciones en la secuencia sin que se altere la solución correspondiente.

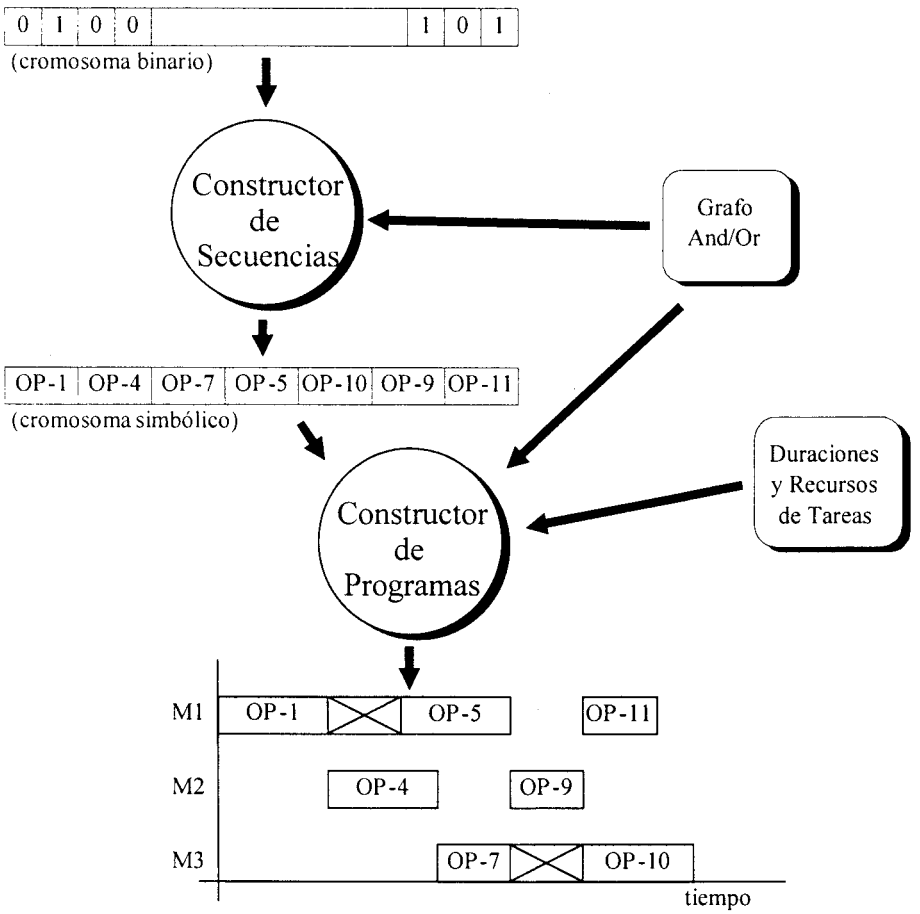


Figura 3.4: Descodificación de los cromosomas

3.3.2 Representación binaria: enumeración (implícita) de soluciones

Para la codificación de las soluciones utilizando cadenas binarias se propone asociar con cada una de las secuencias de tareas posibles un valor numérico distinto y representarlo en forma binaria. Se considerará para ello a los distintos cromosomas simbólicos válidos que podrían ser generados, es decir, secuencias de tareas que satisfacen las restricciones de precedencia y forman un plan de montaje completo. El valor numérico que será asociado con cada cromosoma simbólico atenderá a la enumeración implícita de todas las secuencias posibles.

El número total de individuos posibles se correspondería con el número de soluciones en el caso de que el plan de ensamblaje debiera ejecutarse de manera totalmente secuencial, es decir, que no pudieran efectuarse de forma simultánea más de una operación de ensamblaje. En el caso de ejecución simultánea de tareas (pero sujetas a las restricciones impuestas por el grafo *And/Or*), el número de soluciones estaría limitado superiormente por el número de secuencias de tareas no simultáneas, ya que,

como se mostró anteriormente, varias de estas secuencias podrían dar lugar a una misma distribución de secuencias para múltiples máquinas.

A continuación se muestra el cálculo del número total de individuos, o número de secuencias compatibles de tareas, lo que servirá de base para el proceso de descodificación para obtener la solución asociada.

Para un submontaje dado SA, siendo $T(SA)$ el conjunto de tareas alternativas que pueden dar como resultado SA, y siendo SA_{i1} y SA_{i2} los submontajes de partida para la tarea $T_i \in T(SA)$, el número de secuencias de ensamblaje para obtener SA, $NS(SA)$, viene dado por

$$NS(SA) = \begin{cases} \sum_{T_i \in T(SA)} NS(SA_{i1}) \cdot NS(SA_{i2}) \cdot tmezcla(SA_{i1}, SA_{i2}) & \text{si } T(SA) \neq \varnothing \\ 1 & \text{si } T(SA) = \varnothing \end{cases} \tag{3.5}$$

donde $tmecla(SA_{i1}, SA_{i2})$ representa el número de secuencias que pueden obtenerse a partir de dos secuencias cualesquiera correspondientes a la construcción de los submontajes SA_{i1} y SA_{i2}. Ese número depende del número de componentes de cada una de las dos secuencias, NT_1 y NT_2, respectivamente. En el caso de que los nodos terminales del grafo *And/Or* se correspondan con piezas individuales, el número de tareas necesarias para formar un submontaje coincide con el número de piezas que contiene el submontaje menos 1. Así, un submontaje genérico SA_i, con NP_i piezas, necesitará un número de tareas de ensamblaje NT_i para su construcción:

$$NT_i = NP_i - 1 \tag{3.6}$$

Obsérvese la necesidad de definir como 1 el número de secuencias posibles si el submontaje se corresponde con una pieza individual, de manera que la definición (3.5) sea consistente.

Por otro lado, el número de secuencias que pueden obtenerse al mezclar dos secuencias cualesquiera para formar SA_1 y SA_2, manteniendo el mismo orden relativo de las operaciones que pertenecen a cada secuencia, viene dado por

$$tmezcla(SA_1, SA_2) = \binom{NT_1 + NT_2}{NT_1} = \binom{NP_1 + NP_2 - 2}{NP_1 - 1} \tag{3.7}$$

La enumeración de todas las secuencias permitirá codificar todos los individuos, de forma que a cada secuencia le corresponderá un número entero diferente, cuya codificación en binario resultará ser el cromosoma. El número total de secuencias, $NS(PRODUCTO)$, marcará el tamaño en bits de un cromosoma:

$$n^{o}\ bits = \lceil \log_2 NS(PRODUCTO) \rceil \tag{3.8}$$

Tal enumeración se realiza de la forma siguiente: para cualquier nodo *Or*, correspondiente a un submontaje *SA*, se distribuyen de forma consecutiva todas las secuencias formadas a partir de la misma tarea inicial, es decir, la correspondiente al ensamblaje de *SA*. Siendo *NAND(SA)* el número de nodos *And* que parten del nodo *Or*, o lo que es lo mismo, el número de tareas alternativas que pueden seleccionarse para obtener *SA*, y ordenadas estas tareas como $T_1,...,T_{NAND(SA)}$, la distribución de números enteros para las secuencias que comienzan por la tarea T_i estará en el intervalo

$$\left[\sum_{j<i} NS_j, \sum_{j\le i} NS_j\right) \tag{3.9}$$

siendo

$$NS_j = NS(SA_{j1})\cdot NS(SA_{j2})\cdot\binom{NP_{j1}+NP_{j2}-2}{NP_{j1}-1} \tag{3.10}$$

Para la descodificación de los individuos, la información que es necesario almacenar en el grafo *And/Or* es el límite inferior de cada intervalo de la expresión (3.9), asociada con cada tarea de ensamblaje, y que será denotado por $base(T_i)$. El constructor de secuencias (véase la Figura 3.5) está basado en toda esa información para construir una secuencia a partir de la codificación binaria de un entero en el intervalo $[0, NS(PRODUCTO))$. Se usa un algoritmo recursivo, *SEQUENCE*, que emula al algoritmo de cambio de base de representación de enteros. En este caso, en lugar de realizar repetidamente la división por la base de representación, se realizan diferencias y divisiones por los factores que intervienen en la expresión recursiva (3.5).

Como puede observarse, en el algoritmo *SEQUENCE* se usan varias funciones auxiliares, aparte de la función *MERGE*, desarrollada en la propia Figura 3.5, y que será comentada más tarde. Por un lado, la función *buscarPrimeraTarea* determina la tarea que corresponde al valor asociado al valor numérico *x* en el submontaje. Relacionado con la tarea escogida, se tendrán los submontajes de partida y el número de piezas de los mismos. Por otro lado, se utiliza la función div (*a*, *b*) para obtener el cociente y el resto de la división entera, siendo *a* el dividendo y *b* el divisor. La función comb (*m*, *n*) es usada para hacer referencia al número combinatorio $\binom{m}{n}$. Por último, se ha usado el símbolo :: para aludir al operador correspondiente a la inserción de un elemento por delante en una secuencia.

La función *MERGE* toma dos secuencias y las mezcla para obtener otra, manteniendo el orden relativo entre las tareas de una misma secuencia. La construcción de la nueva secuencia se realiza seleccionando la primera tarea de una de las dos secuencias y procediendo recursivamente con las secuencias que contienen las tareas aún no seleccionadas. Se utilizan las funciones *first* y *rest* para la obtención del primer elemento de una secuencia y la secuencia resultante de eliminar el primer elemento, respectivamente.

```
Algoritmo SEQUENCE-BUILDER (x, PRODUCTO) devuelve secuencia
   devolver SEQUENCE (x, PRODUCTO)

Algoritmo SEQUENCE (x, SA) devuelve secuencia
   si SA contiene más de una pieza
      < task, SA1, SA2, NP1, NP2 > := buscarPrimeraTarea (x, SA)
      x := x – base (task)
      < q1, r1 > := div (x, comb (NP1 + NP2 – 2, NP1 – 1))
      < q2, r2 > := div (q1, NS (SA2))
      s1 := SEQUENCE (q2, SA1)
      s2 := SEQUENCE (r2, SA2)
      secuencia := task :: MERGE (s1, s2, NP1, NP2, r1)
   si no
      secuencia := [ ]
   fsi
   devolver secuencia

Algoritmo MERGE (s1, s2, NP1, NP2, x) devuelve secuencia
   si s1 = [ ]
      secuencia := s2
   si no, si s2 = [ ]
      secuencia := s1
   si no
      si x < comb (NP1 + NP2 – 3, NP1 – 2)
         s := MERGE (rest (s1), s2, NP1 – 1, NP2, x)
         secuencia := first (s1) :: s
      si no
         x := x – comb (NP1 + NP2 – 3, NP1 – 2)
         s := MERGE (s1, rest (s2), NP1, NP2 – 1, x)
         secuencia := first (s2) :: s
      fsi
   fsi
   devolver secuencia
```

Figura 3.5: El *constructor de secuencias* (*SEQUENCE-BUILDER*).

Como puede observarse, el algoritmo *MERGE* opera de forma recursiva hasta que una de las secuencias ha sido procesada completamente.

El criterio de selección del primer elemento de una u otra secuencia se basa en la siguiente propiedad. Como se indicó antes, el número de mezclas diferentes de dos secuencias de m y n elementos es igual a $\binom{m+n}{m}$. De ellas, el número de aquellas que comienzan con el primer elemento de la primera secuencia es $\binom{m+n-1}{m-1}$, y las que comienzan con el primer elemento de la segunda son $\binom{m+n-1}{m}$. De ahí que dicha elección se realice comparando x con la primera de esas cantidades (haciendo la traslación correspondiente para m y n), que divide en dos grupos el intervalo de valores posibles para x.

En el caso ideal de que todos los valores numéricos que aparezcan en las expresiones anteriores fueran potencias de 2, los bits más significativos indicarían la tarea inicial seleccionada, los siguientes denotarían la secuencia de tareas asociadas al primer submontaje, a continuación los bits relacionados con la secuencia de tareas del segundo submontaje, y los bits menos significativos estarían asociados a la mezcla de las secuencias (véase la Figura 3.6). Este esquema se repetiría de forma recursiva además para cada secuencia de bits asociada con una secuencia de tareas, es decir, el segundo y tercer bloque de bits.

En general, la distinción de bits referida antes no se realizará de forma nítida, por lo que la información genética que mantendrá una cadena de bits no será excesivamente elevada. Podría ocurrir que dos números naturales consecutivos con la misma raíz (bits más significativos) se refieran a diferentes planes de ensamblajes, es decir, a diferentes árboles de tareas, conteniendo por tanto tareas diferentes. También podrá ocurrir que dos números consecutivos refiriéndose a secuencias muy parecidas tengan una raíz diferente. Eso sí, todas las secuencias correspondientes a un mismo árbol vendrán codificados por números enteros consecutivos. Serán los bits menos significativos los que en general pierdan más información genética.

En línea con lo referido en el párrafo anterior, se ha considerado la representación de los números naturales correspondientes a los individuos mediante códigos Gray para evaluar su posible efecto en la mejora del algoritmo.

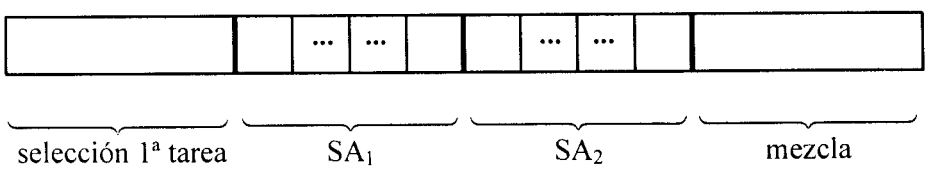


Figura 3.6: Codificación binaria de los individuos.

3.4 Operadores genéticos

De acuerdo con los modelos de representación propuestos, se realizan distintos planteamientos para cada uno de ellos, en cuanto a la utilización de operadores genéticos. Mientras que para el modelo de representación binaria pueden usarse los operadores genéticos estándar, para el modelo simbólico deben definirse operadores específicos, eso sí, con la esperanza de explotar mejor el conocimiento sobre el problema.

Para el modelo de representación binaria se han utilizado los operadores genéticos estándar, cruce y mutación, sobre estructuras binarias. Para el caso del operador de cruce, se ha optado únicamente por un *cruce simple*. La razón está en lo comentado en el apartado anterior, es decir, debido a la codificación utilizada, los bits menos significativos difícilmente pueden albergar ninguna información genética útil para otros individuos, por lo que ligarlos a los más significativos, como se haría en el caso del cruce *multipunto*, tendría poco sentido, aparte además del costo computacional añadido, aunque pequeño.

Para el caso del operador de mutación, no se ha realizado ninguna consideración especial respecto a los usados comúnmente en modelos de representación binaria.

Debe observarse que los operadores genéticos utilizados en el modelo binario obtienen individuos legales, ya que todos los códigos numéricos entre 0 y el número total de secuencias menos 1 están reservados a soluciones válidas. Sólo faltaría realizar una consideración al respecto. Dado que el número de secuencias de tareas no es en general una potencia de 2, los códigos binarios correspondientes a un número entero que supere al número total de secuencias no se corresponderían con individuos válidos. Ante esto, se han considerado dos opciones. La primera sería considerar el bit más significativo como no relevante en tales casos, con lo cual habría soluciones con más de una codificación posible. La otra opción es penalizar fuertemente a tales individuos en la función objetivo. Ninguna de las dos opciones resuelve con plena satisfacción el problema planteado, ya que ambos casos suponen una discontinuidad importante en la codificación de las soluciones del problema.

En el caso del modelo de representación simbólica, cada solución viene representada de forma indirecta a través de una secuencia de tareas, cuya ordenación satisface las restricciones de precedencia recogidas en el grafo *And/Or*. Además, todas las tareas de la secuencia deben pertenecer a un mismo árbol de ensamblaje para que se corresponda con una solución válida del problema. Esto puede suponer una fuerte dificultad en el proceso de búsqueda realizado por los operadores genéticos a utilizar. Una tarea de montaje viene definida por los submontajes usados para formar el submontaje mayor, y por éste último submontaje resultante. Así, la presencia de una tarea en una solución está fuertemente condicionada por la presencia de tareas relacionadas con estos submontajes. Normalmente, existirán pocas posibilidades de construir una nueva solución a partir de partes significativas de dos soluciones cualesquiera. Será necesario, pues, añadir otras tareas (probablemente no pocas) para completar la solución y eliminar

algunas otras. De esta forma se prevé la dificultad de encontrar operadores de cruce eficientes.

Dos familias de operadores genéticos se han usado para la búsqueda en el espacio de soluciones completo. La primera incluye operadores que buscan localmente secuencias en un plan de montaje predeterminado, el de los cromosomas padre. Estos operadores, denotados más abajo como operadores de *Re-Ordenamiento de Tareas*, son similares a los usados para la resolución de los problemas de TSP y JSSP en la literatura, pero obviamente resultan insuficientes para encontrar el óptimo. La otra familia de operadores tiene el propósito de buscar secuencias en otros planes de ensamblaje, y nos referiremos a ellos como operadores de *Re-Planning*. Esto se realiza básicamente introduciendo una nueva tarea en una solución, y sustituyendo ciertas tareas por otras para mantener la validez de los cromosomas. Se han usado varios algoritmos para ello, desde aquellos que minimizan el número de tareas sustituidas, lo que supone una búsqueda difícil en el grafo *And/Or*, hasta los que tienen como objetivo principal la minimización del coste computacional, sacrificando posiblemente la preservación de información genética.

En las siguientes subsecciones se describen con detalles los distintos operadores propuestos para ambas familias.

3.4.1 Operadores de *Re-Ordenamiento de Tareas* (ROT)

Este tipo de operador está diseñado con el propósito de buscar nuevas secuencias en un plan de montaje predeterminado. Debido a que es muy improbable que dos secuencias asociadas al mismo plan coincidan en una población, estos operadores están implementados únicamente mediante operadores de mutación. Actúan sobre un cromosoma seleccionando una tarea de la secuencia e intentando desplazarla a una posición diferente de la que ocupa. Sus tareas predecesoras o sucesoras también están involucradas en el desplazamiento, de manera que podrían mantener su posición en la secuencia o verse desplazadas con la tarea seleccionada. Estos desplazamientos se muestran a través de los algoritmos que se exponen a continuación. El operador ROT-FWD desplaza la tarea seleccionada hacia la parte final de la secuencia, y el operador ROT-BACK lo hace en sentido inverso. Los algoritmos marcados con un * también desplazan las tareas predecesoras o sucesoras. Nótese que la longitud de las secuencias es el número de piezas del producto a ensamblar (*NPIEZAS*) menos uno, y que la primera operación debe mantenerse fija debido a las restricciones de precedencia impuestas por el plan de montaje.

ROT-FWD:

1. Seleccionar una posición aleatoria entre 3 y *NPIEZAS*–1, *InitPos*. Sea *InitTask* la tarea situada en la posición *InitPos*.
2. Seleccionar una posición aleatoria entre la posición de la tarea predecesora de *InitTask* y la posición *InitPos*, *EndPos*.

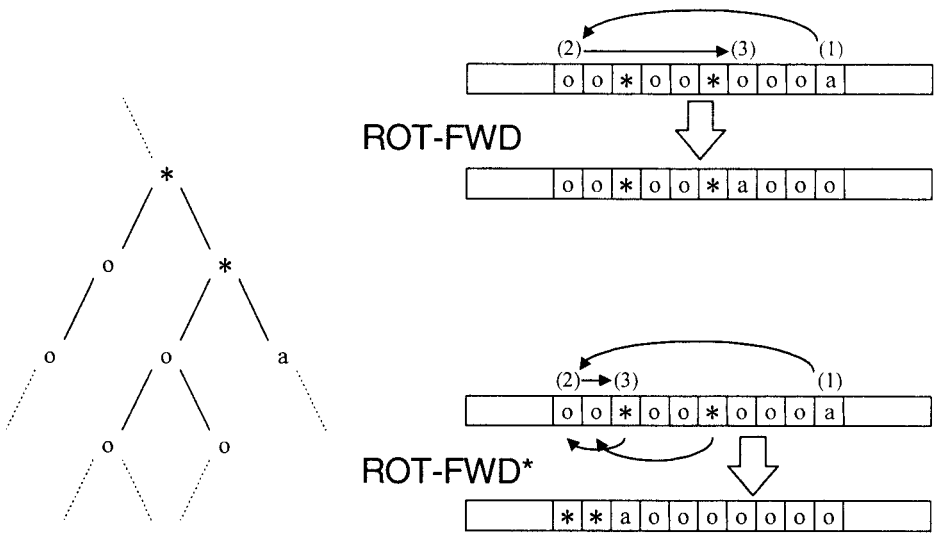


Figura 3.7: Operadores ROT-FWD y ROT-FWD*.

3. Situar la tarea *InitTask* en la posición *EndPos* y desplazar una posición todas las tareas que estaban situadas entre las posiciones *EndPos* e *InitPos*–1.

La Figura 3.7 ilustra de manera gráfica su funcionamiento. A la izquierda se muestra esquemáticamente el árbol de ensamblaje correspondiente al individuo, y a la derecha las secuencias correspondientes. La tarea seleccionada es la marcada con "a", sus tareas predecesoras son las señaladas con "*", mientras que las señaladas mediante "o" no tienen ninguna restricción de precedencia con respecto a la tarea seleccionada. Como puede observarse, el movimiento de la tarea seleccionada está restringido a su izquierda por la posición que ocupa su tarea predecesora más inmediata.

ROT-FWD*:

1. Seleccionar una posición aleatoria entre 3 y *NPIEZAS*–1, *InitPos*. Sea *InitTask* la tarea situada en la posición *InitPos*.
2. Seleccionar una posición aleatoria entre 2 e *InitPos*, *EndPos*.
3. Colocar la tarea *InitTask* y sus predecesoras situadas en el intervalo *EndPos..InitPos* en la posición *EndPos* y siguientes de forma consecutiva manteniendo el mismo orden relativo que tenían en la secuencia original. El resto de tareas en el mismo intervalo se desplazarán en sentido inverso ocupando las posiciones que siguen a la posición en que quedó la tarea *InitTask*, manteniendo también el orden relativo entre ellas.

En la Figura 3.7 puede observarse cómo ahora se permite un desplazamiento mayor para la tarea seleccionada. En el caso de superar las posiciones de tareas predecesoras, éstas también se verán desplazadas a su izquierda.

ROT-BACK:

1. Seleccionar una posición aleatoria entre 2 y *NPIEZAS*–2, *InitPos*. Sea *InitTask* la tarea situada en la posición *InitPos*.
2. Seleccionar una posición aleatoria entre *InitPos* y la posición anterior a la que ocupa la primera tarea sucesora de *InitTask* en la secuencia, *EndPos*.
3. Colocar *InitTask* en la posición *EndPos* y desplazar una posición todas las tareas que estaban situadas entre *InitPos*+1 y *EndPos*.

La Figura 3.8 muestra el funcionamiento del operador. Aparte de que ahora el desplazamiento es en sentido inverso al que se producía en el operador ROT-FWD, la tarea seleccionada tiene impedido su movimiento por las posiciones de sus dos posibles sucesoras inmediatas.

ROT-BACK*:

1. Seleccionar una posición aleatoria entre 2 y *NPIEZAS*–2, *InitPos*. Sea *InitTask* la tarea situada en la posición *InitPos*.
2. Seleccionar una posición aleatoria entre *InitPos* y *NPIEZAS*–1, *EndPos*.
3. Colocar *InitTask* y sus predecesoras situadas en el intervalo *EndPos*..*InitPos*

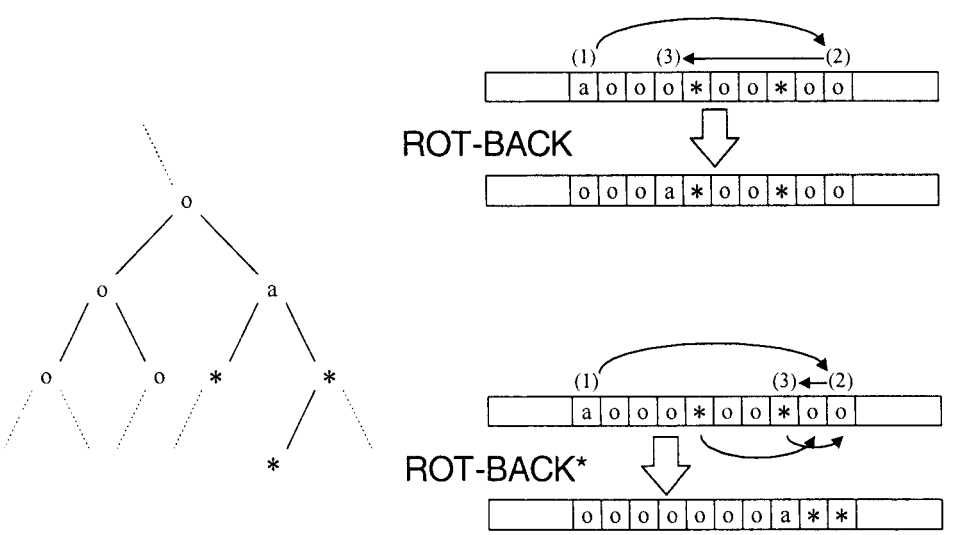


Figura 3.8: Operadores ROT-BACK y ROT-BACK*.

en la posición *EndPos* y las siguientes consecutivas en el mismo orden relativo que tenían, y desplazar las otras tareas. El resto de tareas en el mismo intervalo se desplazarán en sentido inverso ocupando las posiciones anteriores a la posición en que quedó la tarea *InitTask*, manteniendo también el orden relativo entre ellas.

El funcionamiento del operador puede verse de forma esquemática en la Figura 3.8, donde se indica el desplazamiento tanto de la tarea seleccionada como de sus tareas sucesoras comprendidas entre la posición inicial y la final de la tarea seleccionada.

3.4.2 Operadores *Re-Planning* (RP)

Los operadores de esta otra familia tienen como objetivo la búsqueda de secuencias correspondientes a otros planes de ensamblaje. Con el uso de estos operadores se asegura la búsqueda a través de todo el espacio de soluciones. Los individuos resultantes contendrán nuevas tareas de montaje, que provendrán de otros individuos presentes en la población (operadores de cruce) o generados de forma aleatoria (operadores de mutación). Se requerirán nuevas tareas para completar un cromosoma correcto, de manera que sustituyan a otras tantas.

Las secuencias generadas por estos operadores genéticos mantendrán las posiciones que las tareas ocupaban en los cromosomas padre, y las nuevas tareas llenarán los huecos, en algún orden compatible con las restricciones de precedencia.

Operadores de *cruce* RP (RPC(x)):

Estos operadores toman dos individuos (padres) y generan dos hijos, intentando mezclar información genética de ambos padres. El siguiente algoritmo muestra la generación de uno de los hijos. El otro individuo se genera de la misma forma, cambiando el orden de los padres.

1. Sea T_1 una tarea escogida al azar presente en uno de los padres, P_1. El hijo C_1 contendrá a T_1 y sus sucesores presentes en P_1. El resto de tareas para completar C_1 se seleccionarán del otro padre, P_2, cuando sea posible, según se verá a continuación.
2. Si uno de las posibles *tareas predecesoras inmediatas* de T_1 en el grafo *And/Or* está presente en P_2 (llamémosle F_1), las tareas que faltan por completar C_1 se tomarán de la parte del árbol de P_2 que queda tras eliminar las tareas sucesoras de F_1.
3. En el caso de que P_2 no contenga ninguna tarea que sea una predecesora inmediata de T_1:
   (a) **RPC(1)**: se escogerá al azar una de las posibles tareas predecesoras inmediatas de T_1 del grafo *And/Or* para formar parte de C_1, así como sus tareas sucesoras correspondientes al otro submontaje. Llamando T_1 a la tarea predecesora introducida, ir al paso 2 hasta que se encuentre un

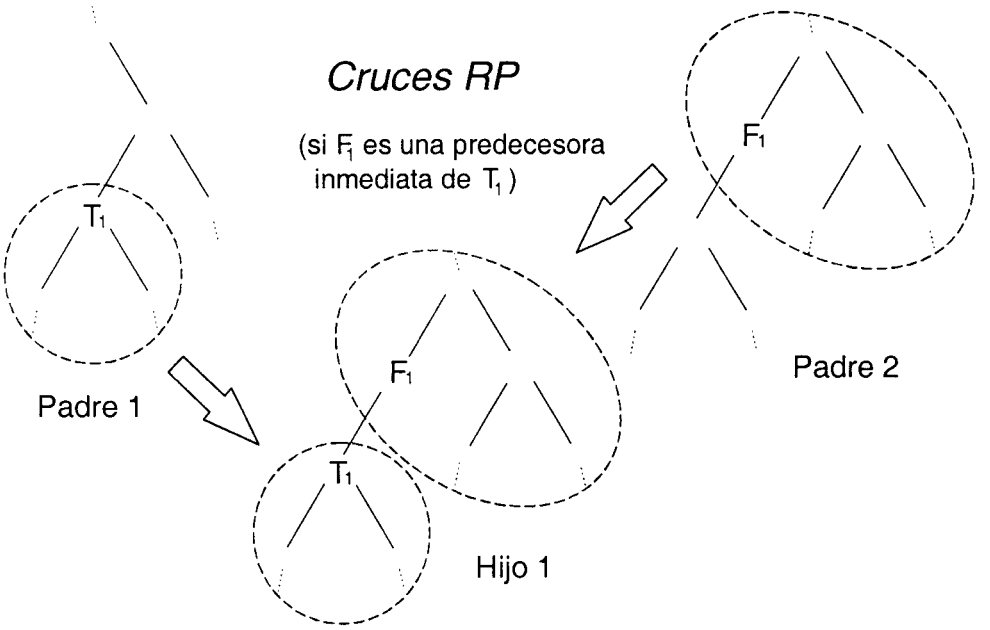


Figura 3.9: Operadores RPC(x).

predecesora introducida, ir al paso 2 hasta que se encuentre un predecesor inmediato en P_2, o se alcance el nodo raíz.

(b) **RPC(2)**: Si alguna de las tareas predecesoras inmediatas a T_1 en el grafo *And/Or* tiene una tarea predecesora inmediata en P_2, entonces será escogida para formar parte de C_1. Las tareas correspondientes al otro submontaje se seleccionan aleatoriamente. En el caso de que no se dé la condición anterior, se procede como en (a) para seleccionar la tarea predecesora. Llamando T_1 a la tarea predecesora introducida, ir al paso 2 hasta que se encuentre un predecesor inmediato en P_2, o se alcance el nodo raíz.

(c) **RPC(all)**: Se realiza una búsqueda exhaustiva para conectar el nodo raíz con la tarea seleccionada T_1, a través de las tareas de P_2. De esta forma, aquellas tareas de P_2 que puedan ser predecesoras de T_1 serán escogidas para formar parte de C_1. Además, también serán escogidas sus sucesoras correspondientes al otro submontaje. El resto de tareas serán seleccionadas de forma aleatoria.

La Figura 3.9 muestra la actuación de los operadores de cruce en el caso de encontrarse una tarea predecesora inmediata de la tarea seleccionada en el otro individuo (paso 2). La diferencia entre los tres operadores de cruce propuestos está en la búsqueda de tareas predecesoras en el otro individuo. El operador RPC(1) busca una predecesora inmediata, y en caso de no encontrarla escoge una de forma aleatoria, repitiendo el proceso hasta encontrarla o llegar a la raíz del grafo *And/Or*. El operador RPC(2) busca una predecesora dos niveles más arriba en el grafo *And/Or*, mientras que RPC(all) extiende la búsqueda hacia la raíz. En la Figura 3.10 se muestra el efecto de esta búsqueda a

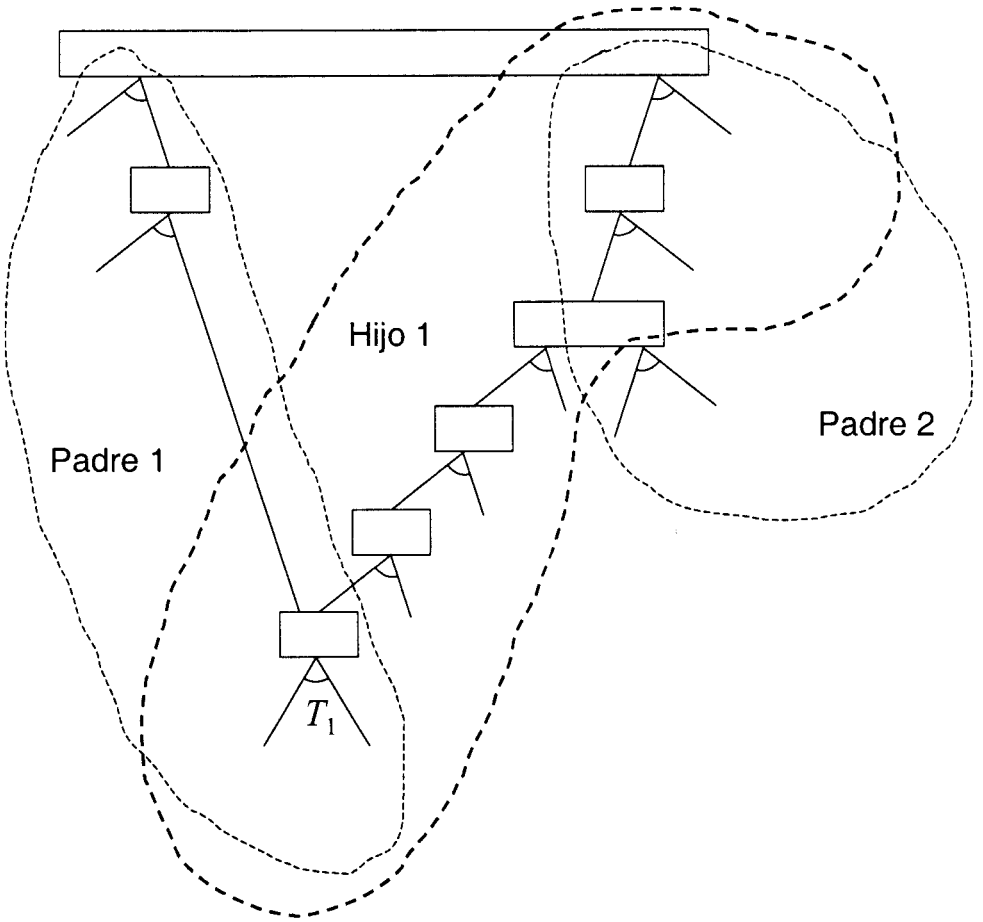


Figura 3.10: Búsqueda de tareas predecesoras a T_1 en RPC(all).

través del grafo *And/Or*. Para guiar de manera eficiente esa búsqueda, debe observarse la relación existente entre los nodos *Or*: el conjunto de piezas correspondiente al nodo *Or* padre de T_1 estará contenido en el conjunto de piezas correspondiente a uno de los dos nodos *Or* de destino de aquellas tareas de P_2 que aparezcan en el nuevo individuo obtenido en el cruce.

Nótese que ninguno de los tres operadores de cruce garantiza que los hijos tendrán información genética de los dos padres. RPC(all) tendrá más posibilidades de alcanzar ese objetivo, pero realizará un gasto computacional mayor. En el otro extremo, RPC(1) es más simple e incluye más aleatoriedad.

Mutación RP (RPM):

Este operador toma un individuo y lo modifica cambiando un subárbol del plan de montaje por otro. Las posiciones de las nuevas tareas en la secuencia serán las mismas que ocupaban las tareas sustituidas. Trabaja como sigue:

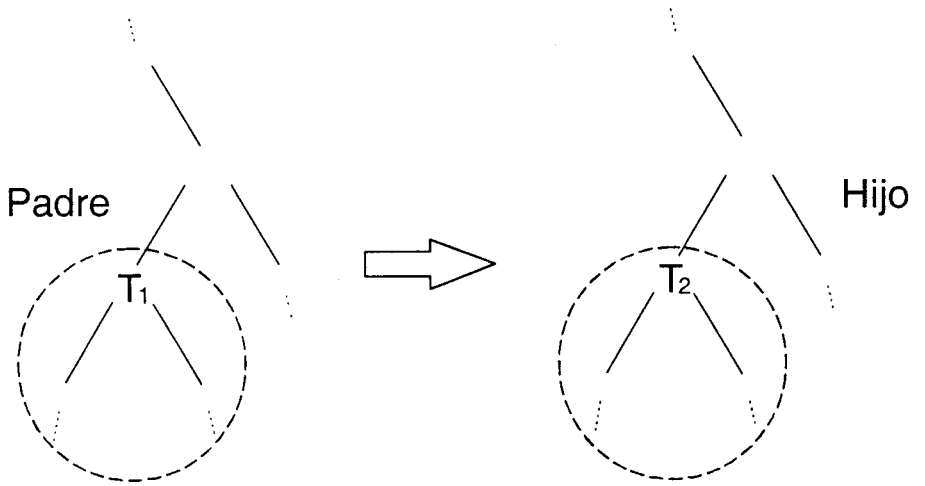


Figura 3.11: Operador RPM.

1. Seleccionar una tarea T al azar presente en el individuo.
2. Seleccionar al azar una tarea T' hermana de T.
3. Sustituir en el individuo T y sus sucesoras por T' y sus sucesoras, seleccionadas al azar, teniendo en cuenta las restricciones de precedencia.

En la Figura 3.11 se muestra el efecto de la sustitución del subárbol de tareas seleccionado por otro. La profundidad de la secuencia seleccionada determinará en general el nivel de actuación del operador. Por un lado, si la tarea está cerca de la raíz y tiene muchas tareas por debajo, será más probable un cambio importante en el individuo, por un lado, por el número de tareas que podrían cambiar, y por otro, por la mayor probabilidad de que se encuentren tareas alternativas. Si la tarea seleccionada está cerca de una hoja del árbol, las probabilidades de que no existan tareas alternativas para sustituir al subárbol, además de que el efecto del cambio será menor por el menor número de tareas involucradas

3.5 Resultados

Como se señaló previamente, existen muchos factores que influyen en la complejidad del problema propuesto. Algunos de los más importantes son el número de piezas del producto a ensamblar, el tamaño y estructura del grafo *And/Or*, y la distribución de las duraciones y recursos compartidos (máquinas y herramientas) entre todas las tareas candidatas a formar parte de la solución.

Para evaluar el comportamiento de los algoritmos genéticos correspondientes a los modelos propuestos se han usado dos productos hipotéticos. Ambos están formados por 30 piezas, y la estructura del grafo de conexiones es tal que el número de conexiones entre las piezas es mínimo. El grafo *And/Or* del primer producto (caso A) tiene

solamente un árbol de ensamblaje, es decir, sólo existe un nodo *And* por cada nodo *Or* que no sea hoja, por lo que contiene 59 nodos *Or* y 29 nodos *And*, aquellos correspondientes a las tareas necesarias para ensamblar el producto completo. Existen aproximadamente 10^{16} secuencias de tareas lineales legales (individuos) correspondientes a dicho grafo *And/Or*. El segundo ejemplo (caso B) incluye en su grafo *And/Or* varias tareas alternativas para cada nodo *Or*, y contiene en total 396 nodos *Or* y 764 nodos *And*. En este caso existen unos 10^{21} individuos posibles. Debe recordarse que el número de programas de montaje diferentes depende no sólo del número de secuencias, sino también de la distribución de recursos compartidos (y su número) y duraciones entre todas las tareas. De esta forma, a varios individuos podrían corresponderles el mismo programa de montaje, y por lo tanto, la misma solución.

Las medidas que reflejan las gráficas de resultados se refieren en la parte de arriba a valores medios de 50 experimentos. Cada experimento supone comenzar con una población inicial diferente. La parte de abajo representa el mejor resultado de los 50 experimentos. Además, todos los valores representan la media de 10 problemas diferentes, es decir, 10 distribuciones distintas de duraciones y recursos compartidos entre las tareas del grafo *And/Or*. Los valores muestran la mejor solución encontrada hasta el número de evaluaciones que se indica. Las gráficas incluyen también el valor de la solución óptima (OPT) y los resultados de un algoritmo totalmente aleatorio (RND).

3.5.1 Modelo de representación binaria

Para el primer producto, caso A, se necesitan 55 bits para representar a todos los posibles individuos, mientras que para el segundo, caso B, son necesarios 70 bits. Claramente se ve la necesidad de implementar una representación de los naturales, junto con sus operaciones básicas, que permita salvar los límites impuestos por los formatos básicos.

Para los dos casos planteados se han examinado por separado los operadores de cruce y mutación, comparándolos con la búsqueda aleatoria, quedándonos con la mejor solución encontrada hasta ese momento. En ambos casos se ha examinado la influencia de la selección proporcional a una medida de la función objetivo (su diferencia respecto al peor valor) y de la selección proporcional al rango que ocupa en la población (etiquetado R), así como de la codificación binaria tradicional o mediante código Gray (etiquetado G).

Cuando se utiliza exclusivamente el operador de *cruce* (Figura 3.12 y Figura 3.15), se observa cómo mejoran los resultados respecto a la búsqueda aleatoria, mucho más acusado en el caso B, en el que el conjunto de soluciones es mucho mayor y el problema incluye la selección de tareas. No se dan diferencias apreciables respecto al uso de un mecanismo de selección de individuos u otro, así como tampoco parece influir demasiado la utilización de código Gray.

El operador de *mutación* en solitario mejora más notoriamente los resultados (Figura 3.13 y Figura 3.16), también más claramente en el caso B. En este caso, parece que la selección de individuos en función del rango, lo cual disminuye en cierto modo la

influencia de los superindividuos, hace que el operador se comporte mejor. Nuevamente no parece que influya demasiado la utilización de código Gray (en un caso los resultados son ligeramente mejores y en el otro sucede lo contrario).

Cuando se utilizan conjuntamente ambos operadores, el comportamiento del algoritmo es algo mejor que cuando sólo actúa el operador de *cruce*, pero claramente peor que cuando solamente actúa el operador de *mutación* (ver Figura 3.14 y Figura 3.17). La razón de este comportamiento debe buscarse en que, como ya se ha apuntado anteriormente, la información genética correspondiente a los bits menos significativos de un individuo es muy poco útil para otros individuos con los bits más significativos diferentes, según la codificación que se ha empleado. En todo caso, la estructura del grafo *And/Or*, y en definitiva del problema planteado, hacen extremadamente difícil superar este punto.

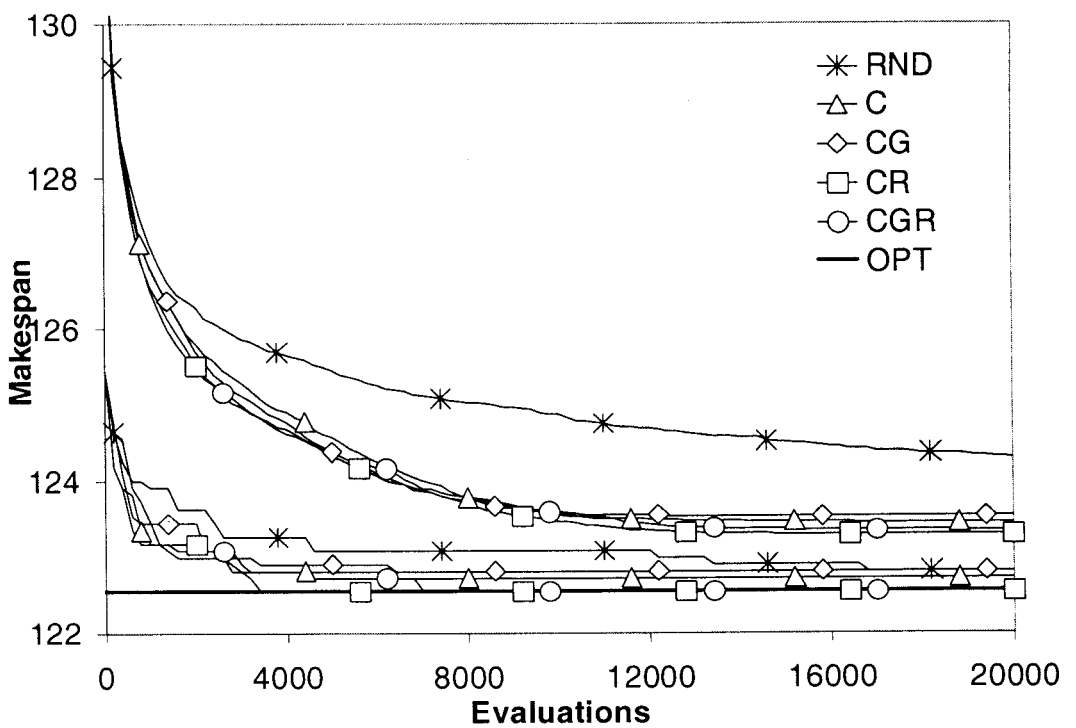


Figura 3.12: Representación binaria - Caso A: Operador Cruce.

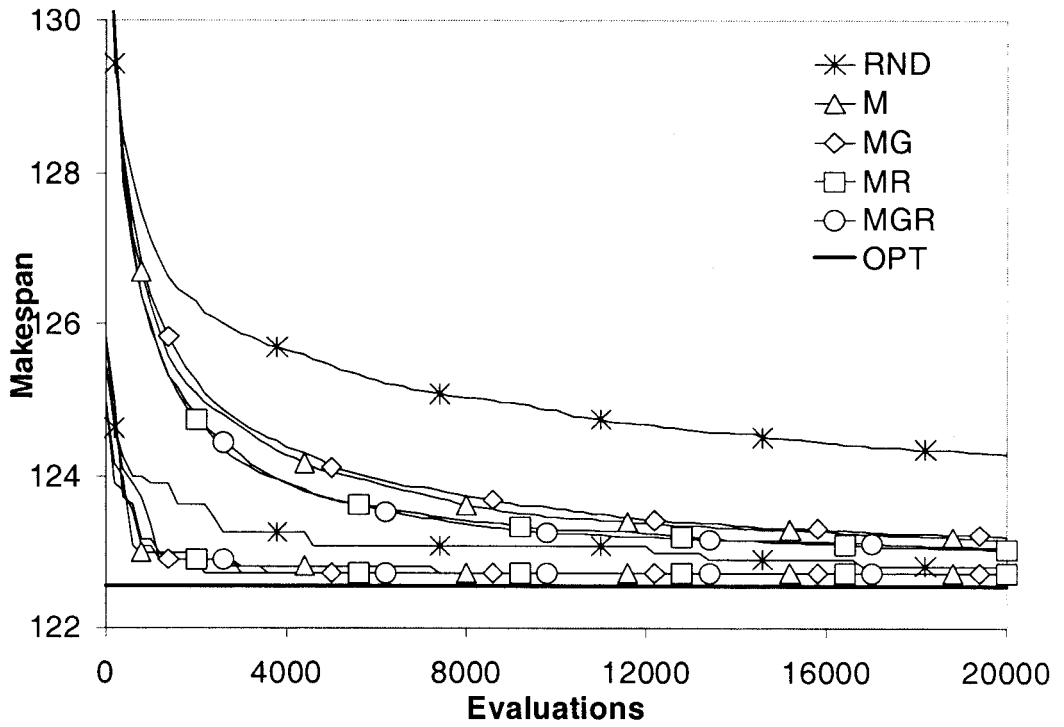


Figura 3.13: Representación binaria - Caso A: Operador Mutación.

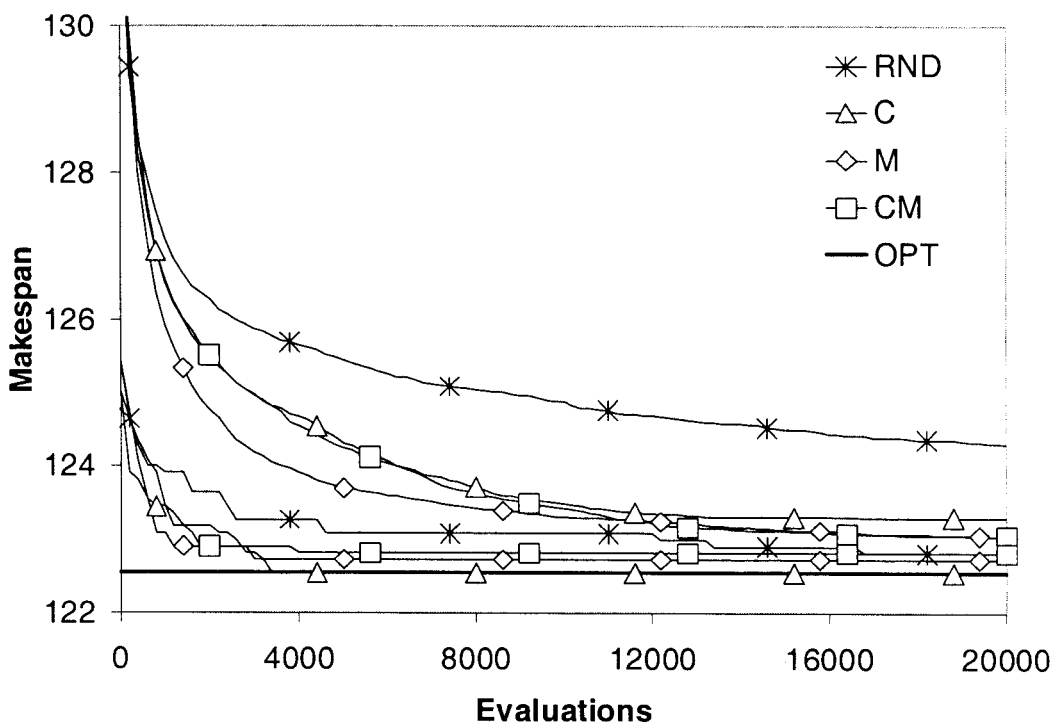


Figura 3.14: Representación binaria - Caso A: Comparación de operadores.

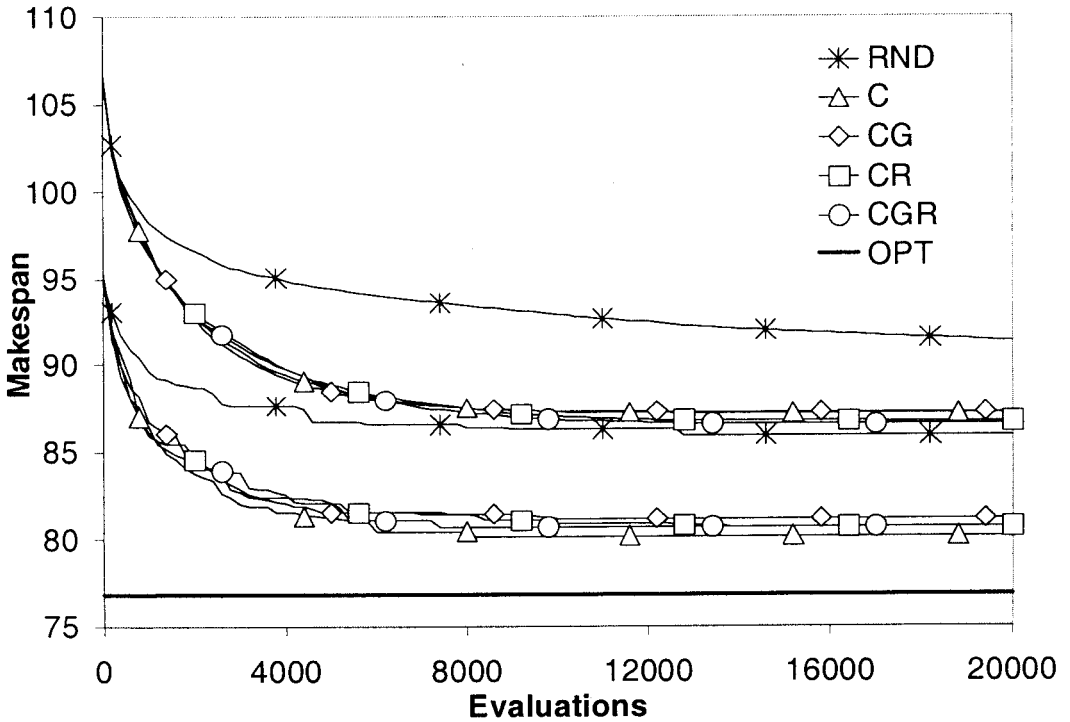


Figura 3.15: Representación binaria - Caso B: Operador Cruce.

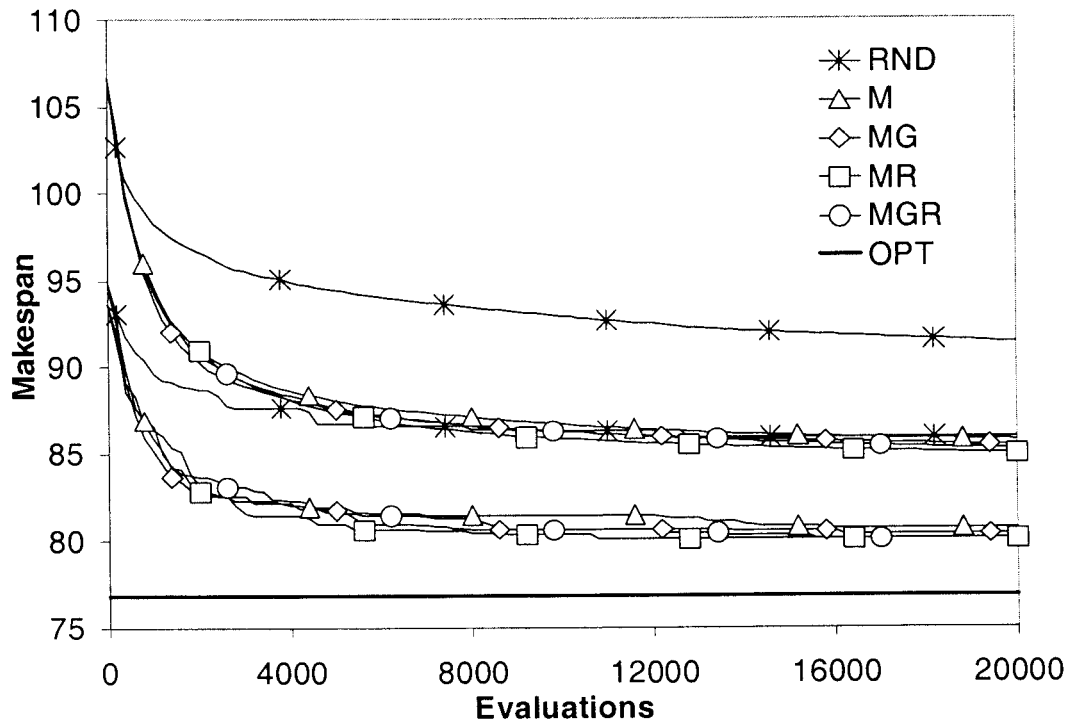


Figura 3.16: Representación binaria - Caso B: Operador Mutación.

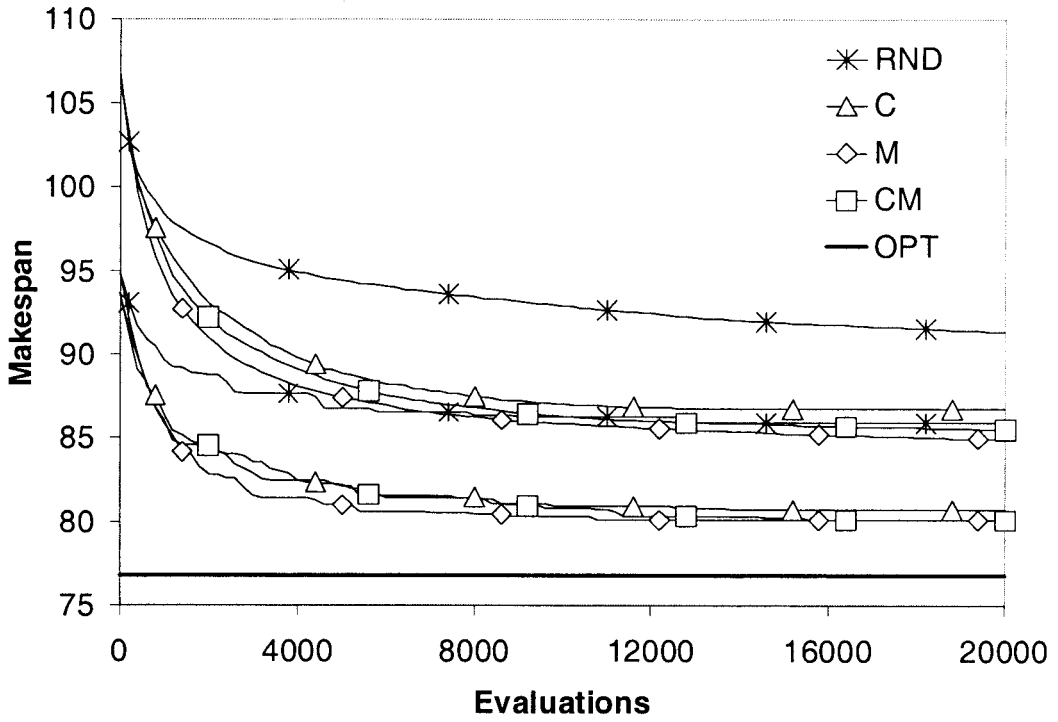


Figura 3.17: Representación binaria - Caso B: Comparación de operadores.

3.5.2 Modelo de representación simbólica

Para el modelo simbólico, el caso A ha sido usado para comparar los operadores ROT. Adviértase que los operadores RP no pueden trabajar en este caso, ya que no pueden generarse nuevos planes de montaje. Por otra parte, los operadores ROT trabajan para encontrar la secuencia óptima correspondiente a un plan de montaje predeterminado, por lo que el caso A representa un buen ejemplo para evaluar su funcionamiento.

La Figura 3.18 muestra las diferencias existentes en los resultados obtenidos por estos operadores trabajando por separado. Puede observarse cómo los operadores que desplazan hacia atrás en la secuencia a la tarea seleccionada (ROT-BACK y ROT-BACK*) se comportan ligeramente mejor que los que lo hacen en sentido contrario (ROT-FWD y ROT-FWD*). Esto puede deberse a que existen menos restricciones en el movimiento hacia atrás por existir un mayor número de tareas en ese sentido no involucradas en restricciones de precedencia, de acuerdo a la estructura de árbol binario del conjunto de esas restricciones. También pueden observarse unos mejores resultados de los operadores ROT-FWD* y ROT-BACK* con relación a ROT-FWD* y ROT-BACK* respectivamente, debido muy posiblemente a que producen mayores cambios en los individuos, aun cuando tienden a agrupar las tareas relacionadas mediante restricciones de precedencia de forma contigua en la secuencia. Se observa también una mejora cuando los operadores ROT trabajan conjuntamente

(ROT-COMB en la figura), mostrando el efecto positivo de la mayor variedad de individuos que pueden generarse.

El caso B incluye múltiples planes de montaje, de forma que representa una instancia más general del problema. Se ha usado para evaluar los operadores RP, y para comparar a todos los operadores genéticos entre sí y su rendimiento cuando trabajan de manera conjunta.

La Figura 3.19 muestra los resultados obtenidos por los operadores de cruce RP. La gran dificultad para mezclar información genética de dos individuos cualesquiera de la población podría explicar los relativamente pobres resultados obtenidos en comparación con los esperados para operadores de cruce típicos. No hay excesiva diferencia entre los distintos resultados, de modo que el uso de RPC(1) podría parecer más razonable frente al uso de RPC(2) y RPC(all) debido a su coste computacional. Pueden sorprender los peores resultados de RPC(all) frente a los otros operadores, teniendo en cuenta que aseguran una mezcla mayor de información de los dos individuos involucrados en el cruce. Sin embargo, puede entenderse que no es tanta la información que se mantiene cuando se diferencia de los otros operadores de cruce. Además, según puede observarse por el trazo horizontal, los resultados pueden reflejar una cierta convergencia prematura debido a que contiene un menor grado de aleatoriedad que los otros operadores de cruce, que generan nuevas tareas en lugar de realizar una búsqueda más profunda de información en los individuos, explorando de esta forma nuevas regiones en el espacio de soluciones.

En la Figura 3.20 se comparan los resultados de los distintos tipos de operadores trabajando por separado y de forma conjunta. Se aprecia por un lado que los resultados del operador de mutación RP-M son ligeramente mejores a los de los operadores de cruce, quizá porque preserva más información genética en los individuos en muchos casos (se sustituye un subárbol de tareas), manteniendo el efecto de la búsqueda en todo el espacio de soluciones al introducir nueva información genética, aunque esto último también puede suceder en los operadores de cruce.

En la misma Figura 3.20, puede observarse cómo los operadores ROT mejoran muy rápidamente la solución encontrada en las primeras generaciones. Al final, su rendimiento queda condicionado por los planes de montaje generados en la población inicial, ya que estos operadores no generan planes de ensamblaje nuevos, limitándose a poder optimizar únicamente los que estén ya presentes desde el principio y sobrevivan a lo largo de las sucesivas generaciones.

Una última curva aparece en la Figura 3.20. Muestra los resultados obtenidos por el algoritmo genético con todos los operadores considerados trabajando de manera conjunta (ALL). Los buenos resultados reflejan la combinación de dos efectos: los operadores ROT optimizan los planes de montaje que han sido generados, y los operadores RP obtienen nuevos planes de montaje.

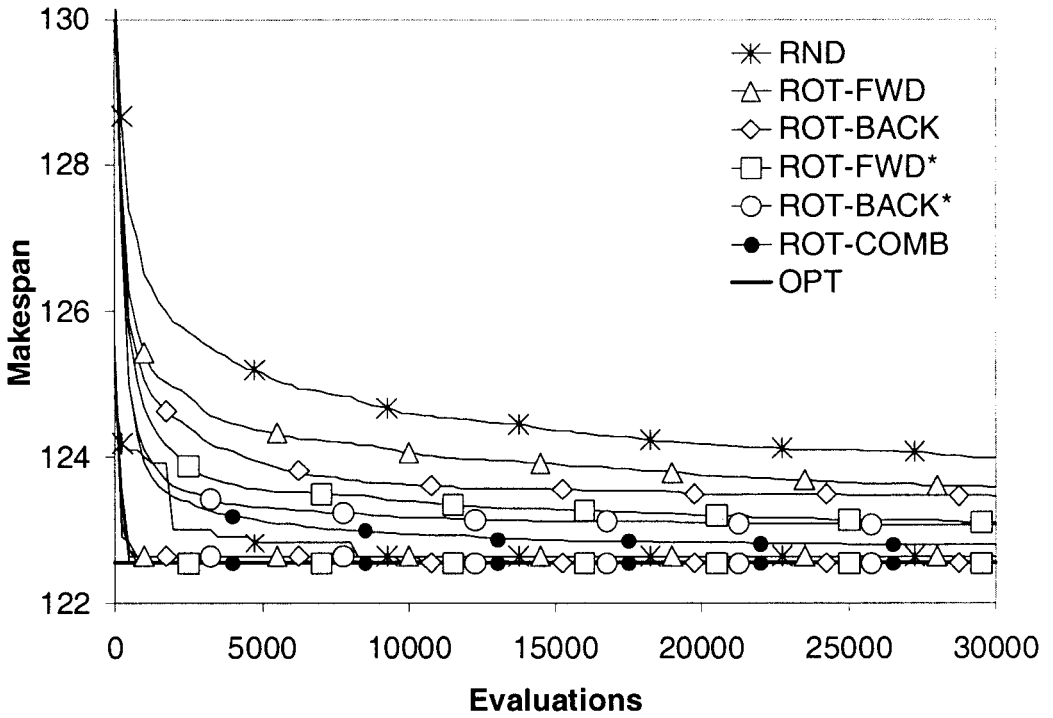


Figura 3.18: Representación simbólica - Caso A: comparación de operadores ROT.

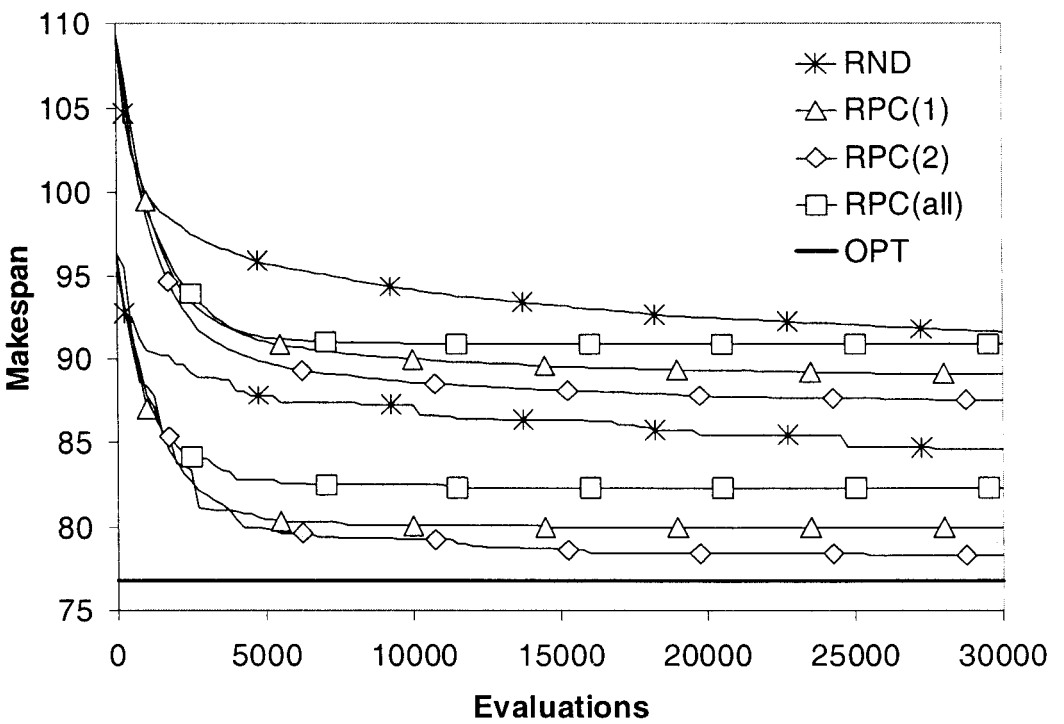


Figura 3.19: Representación simbólica - Caso B: comparación de operadores RPC.

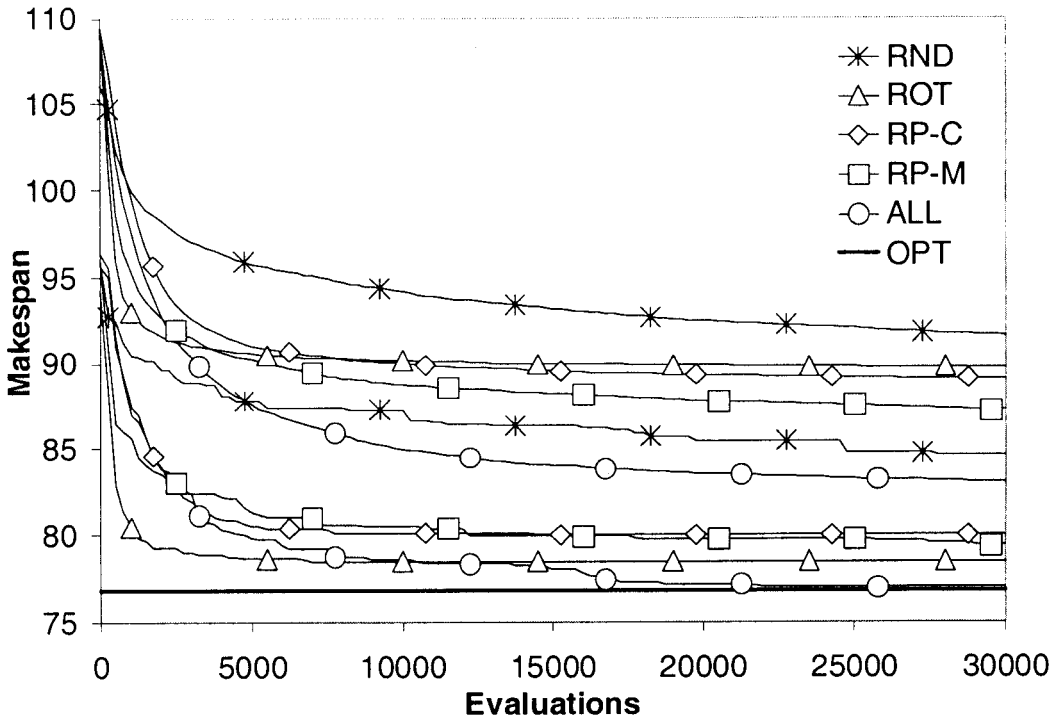


Figura 3.20: Representación simbólica - Caso B: comparación de operadores.

3.5.3 Comparación de resultados

Los mejores resultados sobre el problema correspondiente al caso más general (caso B) de los dos modelos estudiados, representación binaria frente a simbólica, se resumen en la Figura 3.21. Se observa que se obtienen mejores resultados con la representación simbólica, como era de esperar, por el tipo de información que se manipula a través de los operadores genéticos.

En cualquier caso, el funcionamiento de cualquier algoritmo genético depende de un número grande de parámetros: el tamaño de la población, el mecanismo de selección usado, y las probabilidades asignadas a los distintos operadores genéticos, entre otros. Así, es preciso ajustar esos parámetros para obtener una mejora en el funcionamiento del algoritmo genético.

Los diferentes parámetros de un algoritmo genético influyen en cómo evolucionan los miembros de la población. Concretamente, si los parámetros correspondientes a las probabilidades de cruce y mutación son altos, la fracción de población que se mantiene es pequeña, por lo que se estaría realizando una búsqueda más aleatoria si los operadores tienen baja fiabilidad, como parece ser el caso en que nos encontramos. Si por el contrario las probabilidades de aplicación de los operadores son bajas, los elementos mejores de la población tienen más probabilidad de quedarse, pero entonces el proceso puede sufrir una inmovilidad que la dificulta por mejorar los individuos de la

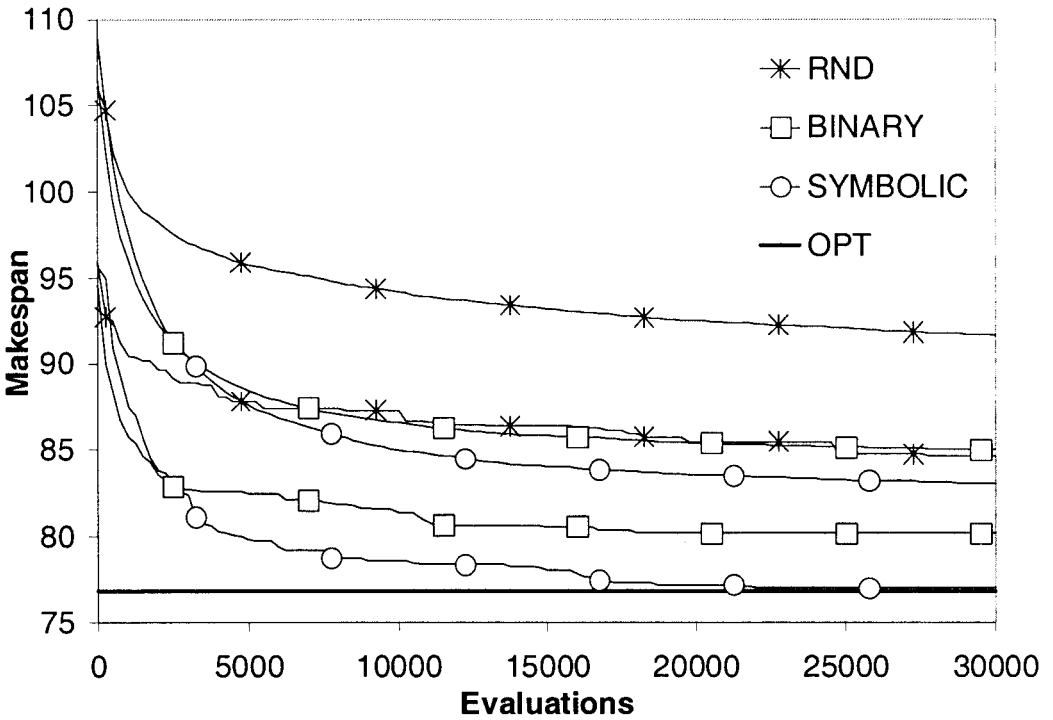


Figura 3.21: Caso B: comparación de los dos modelos.

población. Todos los valores de los parámetros en su conjunto influyen en mayor o menor medida en este proceso. No sólo influirá también la naturaleza del problema a optimizar, sino el propio problema concreto que se esté resolviendo.

Hay que señalar por otro lado que una posible estrategia para la utilización de los algoritmos genéticos sobre un problema de optimización como el que nos ocupa podría ser la ejecución del algoritmo un número determinado de veces, quedándonos con la mejor solución obtenida en todos los experimentos. Los distintos resultados que hemos obtenido anteriormente corroboran la bondad de dicha estrategia, ya que se observa una diferencia importante entre los resultados medios de los 50 experimentos y los correspondientes al mejor experimento, acercándose a los valores óptimos para los problemas ensayados.

3.6 Conclusiones y trabajo futuro

Se ha propuesto un algoritmo genético para resolver el problema de la selección de secuencias de ensamblaje, un problema mucho más difícil que otros problemas de secuenciamiento que ya han sido abordados con cierto éxito usando las mismas técnicas, tales como el TSP y el JSSP. El problema tratado involucra la selección de operaciones de ensamblaje que compondrán el plan de montaje, además de su ordena-

miento óptimo. Se han estudiado dos formas de representar las soluciones al aplicar los algoritmos genéticos: una codificación binaria basada en la enumeración implícita de todas las posibles soluciones, correspondiendo a cada solución un valor numérico distinto que es codificado en binario, y otra simbólica que representa una secuencia ordenada de tareas que cumple las restricciones de precedencia de tareas impuestas en la definición del problema a través del grafo *And/Or*.

El modelo de representación binaria ha permitido el uso de los operadores genéticos convencionales. A cambio, la información genética que parece albergar dicha codificación para ser aprovechada en la aplicación de tales operadores no parece ser demasiado elevada. Por ello, los resultados obtenidos pueden considerarse bastante satisfactorios.

El modelo de representación simbólica ha necesitado la definición de operadores genéticos específicos para la obtención de nuevos individuos legales. Se han propuesto dos familias de operadores. La primera de ellas, operadores ROT, tienen como objetivo la optimización de los planes de ensamblaje asociados a los individuos ya generados, mediante la reordenación de las tareas. Dichos operadores han sido definidos como operadores de mutación, debido a la muy baja probabilidad de que dos individuos de la población se correspondan con el mismo plan de montaje (árbol de tareas), y son similares a los encontrados en la literatura para resolver otros problemas de secuenciamiento.

Los operadores de la otra familia (RP) tienen como objetivo ampliar el ámbito de la búsqueda en todo el espacio de estados, permitiendo la generación de nuevos planes de ensamblaje, mediante la introducción de nuevas tareas en los individuos. Se han definido distintos operadores de cruce y mutación. Se ha constatado la gran dificultad de obtener operadores de cruce eficientes, en el sentido de que los nuevos individuos generados contengan suficiente información genética de ambos padres. Esto es debido a la propia estructura del problema a resolver, que impone como fuerte restricción que todas las tareas correspondientes a una solución correcta pertenezcan al mismo árbol de ensamblaje.

Los resultados obtenidos muestran que la representación simbólica con los operadores genéticos específicos que se han usado resulta más adecuada para encontrar una mejor solución que los operadores de cruce y mutación convencionales trabajando con la codificación binaria.

Como posibles mejoras al trabajo realizado pueden proponerse varias líneas. Por un lado, las mejoras en el modelo de representación binaria deben ir encaminadas a que los bits correspondientes a los cromosomas realicen una distinción más clara sobre los componentes que definen las secuencias asociadas. Para ello, los valores numéricos usados en el cálculo del número de secuencias deben ser representados preservándose en el código binario final una subsecuencia de bits determinada. En cualquier caso, esta cuestión no parece nada trivial debido a la gran heterogeneidad que puede tener la estructura del grafo *And/Or*.

Con respecto al modelo simbólico, las mejoras pueden dirigirse hacia la definición de nuevos operadores genéticos, que pudieran aprovechar mejor la información genética de los individuos.

Un mejor comportamiento del algoritmo genético también podría producirse si en la aplicación de los operadores de cruce son seleccionados individuos que pudieran compartir información genética compatible. Esto podría estar relacionado con la definición de un modelo de algoritmo genético paralelo.

Otra posible línea de trabajo sería la utilización de nuevos modelos de representación, incorporando más elementos de la solución en los cromosomas, como pudieran ser los submontajes involucrados en las operaciones de ensamblaje, las restricciones de precedencia entre las tareas, o los recursos utilizados por las mismas, cuestión que no ha sido considerada en las representaciones utilizadas en el presente trabajo. Los nuevos modelos de representación a definir conllevarían consigo la definición de operadores genéticos específicos, cuya complejidad algorítmica y computacional debería ser considerada.

Los algoritmos genéticos propuestos también permiten la definición de operadores genéticos que incorporen algoritmos de búsqueda local, como los que podrían derivarse de la aplicación de las heurísticas que se definen en el Capítulo 4. Asimismo, estos algoritmos pueden ser usados para generar los individuos que formen la población inicial.

Hay que decir por último que los algoritmos genéticos ofrecen un modelo de trabajo con un alto grado de flexibilidad. Las representaciones propuestas permiten la definición de nuevas funciones de evaluación, que pudieran considerar otros factores de bondad de las soluciones que el aquí utilizado, el tiempo total del ensamblaje. Tales factores podrían de tipo económico, incorporando por ejemplo el costo asociado al uso de determinados recursos.

Capítulo 4

Resolución mediante Algoritmos A*

4.1 Introducción

Los algoritmos A* fueron creación de Hart, Nilsson y Raphael [Hart, *et al.*, 1968]. En [Hart, *et al.*, 1972] corrigieron algunos errores de la presentación original. [Pearl, 1984] los trata con profundidad junto con distintas variantes, incluyendo una demostración rigurosa de sus propiedades formales.

Se trata de algoritmos de búsqueda heurística, en donde los pasos de expansión están guiados por la información del problema, a diferencia de los métodos de búsqueda a ciegas. Cada nodo generado en la búsqueda representa una solución parcial e incompleta del problema, y es evaluado en función de lo prometedor que representa para llegar a la solución óptima. La función de evaluación $f(n)$ se define como la suma del coste real correspondiente al estado del nodo desde el estado inicial, $g(n)$, y de una estimación del coste restante para completar una solución óptima a partir del nodo, $h(n)$. El nodo que se escoge para su expansión en cada paso del algoritmo será el que tenga el menor valor de la función de evaluación.

En la Figura 3.5 se muestra el esquema básico de tales algoritmos, en donde, por motivos de claridad, se obvian algunos detalles. Puede observarse el uso de dos estructuras, ABIERTOS y CERRADOS que contienen a los nodos pendientes de

```
Algoritmo A* (problema) devuelve solución
  iniciar la lista ABIERTOS con el estado inicial del problema
  solución := NULA
  mientras haya candidatos por expandir en ABIERTOS Y solución = NULA
    extraer el nodo n de ABIERTOS con el menor valor de f para su expansión
    incluir n en CERRADOS
    si n es un nodo meta
      solución := reconstruirSolución (n)
    si no
      para cada nodo n' generado en la expansión de n
        calcular f (n')
        si n' no estaba en ABIERTOS ni en CERRADOS
          incluir n' en ABIERTOS
        fsi
      fpara
    fsi
  fmientras
  devolver solución
```

Figura 4.1: Algoritmo *A.

expandir y a los ya expandidos, respectivamente. La estructura ABIERTOS, descrita generalmente como una lista, es en realidad una cola de prioridad en la que los nodos con mayor prioridad son los que tienen un valor más prometedor de la función objetivo. Se consigue con ello que mediante las operaciones asociadas a las colas de prioridad, la manipulación de esa estructura se realice con mayor eficiencia.

Aparte de ello, los nodos forman parte de un *grafo de expansión*, que inicialmente está formado por el nodo correspondiente al estado inicial de la búsqueda. Existirá, por tanto, una conexión entre cada nodo del grafo y los nodos originados en su expansión. Es frecuente que el grafo se considere como *árbol* de expansión para evitar la búsqueda en la estructura de los nodos generados.

Puede demostrarse que la búsqueda A* es *completa* y *óptima*, siempre que se exija a la función $h(n)$ que nunca sobreestime el costo para llegar a la meta. Cuando eso ocurre, se dice que la función heurística $h(n)$ es *admisible*. Nótese que en el caso de que $h(n)$ representara una estimación perfecta, el algoritmo convergería hacia la solución de forma inmediata. En el otro extremo, si $h(n) = 0$, la búsqueda estaría guiada por $g(n)$. En general, cuanto mayor sea el valor de $h(n)$, manteniendo su admisibilidad, más informada será la heurística. Se tendrá como consecuencia que, si una función heurística h_2 está

más informada que otra h_1, entonces h_2 dominará a h_1, es decir, cualquier nodo generado por un algoritmo A* usando h_2 también será generado si se utilizara h_1.

A pesar de que la búsqueda A* sea óptima y completa, la resolución de problemas NP-completos conllevará necesariamente un tiempo de ejecución exponencial, bien porque el número de nodos en el contorno de la meta sea exponencial, o bien porque la función heurística usada, siendo de gran exactitud, rebasa los límites de crecimiento polinómico en el tiempo necesario para su cálculo. Su mayor inconveniente, sin embargo, es la gran cantidad de memoria usada al almacenar todos los nodos generados en la búsqueda, por lo que, si no se toman medidas para reducir el problema, es muy posible que el algoritmo se quede sin memoria antes de poder ofrecer ninguna solución en un tiempo razonable.

Para mejorar la complejidad temporal, y aún más la espacial, será de gran importancia la calidad de la función heurística disponible. Para construir una buena heurística será necesario en general tener un buen conocimiento del problema a resolver, y saber llevar ese conocimiento a la definición de una función de evaluación admisible. A veces se suele recurrir a definir heurísticas no admisibles, de manera que en algunos casos, la heurística sobreestima el coste de la solución óptima. En general, se consigue que el comportamiento del algoritmo mejore, aunque a costa de perder la optimalidad.

Una manera de caracterizar la calidad de una heurística es mediante el *factor de ramificación efectiva* b^*, de forma que si el número total de nodos expandidos para encontrar una solución es N, y la profundidad de la solución es d, b^* sería el factor de ramificación de un árbol que creciera de forma homogénea hasta encontrar la solución en el nivel de profundidad d. De esta forma, b^* se define de manera que cumpla la relación

$$N = 1 + b^* + (b^*)^2 + \ldots + (b^*)^d \tag{4.1}$$

Nótese pues que, cuanto más informada sea la heurística usada por el algoritmo A*, su factor de ramificación efectiva será menor, aproximándose al valor 1. De esta forma se podrá usar el algoritmo para resolver ejemplares cada vez mayores conforme se disponga de heurísticas mejor informadas.

Este capítulo presenta un algoritmo, basado en la búsqueda A*, para la selección del "mejor" plan de ensamblaje de un producto en un sistema de múltiples máquinas de ensamblaje, desde el punto de vista del tiempo total de ensamblaje. El enfoque usado aquí es similar al de [Homem de Mello and Sanderson, 1990] y [Homem de Mello and Sanderson, 1991c], pero considerando una información de las tareas de ensamblaje más detallada, lo que nos conducirá a construir un programa de ensamblaje más preciso. Como se ha indicado en la definición del modelo del problema a resolver, el algoritmo tiene en cuenta, además de las duraciones de las tareas de ensamblaje, los tiempos necesarios para el cambio de herramientas en los manipuladores. Asimismo, también se ha considerado el tiempo necesario para el transporte de los submontajes intermedios desde la estación de trabajo donde se formaron hacia aquella otra en donde deben utilizarse para formar otros. El objetivo del algoritmo es la minimización del tiempo

total de ensamblaje (*makespan*). Para lograr este objetivo, el algoritmo parte de un grafo *And/Or* (representación comprimida de todos los planes de ensamblaje factibles) y de la información de cada una de las tareas de ensamblaje (estación de trabajo y herramientas necesarias, y tiempo estimado de ensamblaje).

La resolución del problema se plantea en dos fases: en la primera se aborda la ejecución estrictamente secuencial de tareas impuesta por las restricciones de precedencia de tareas, y en la segunda, la más compleja, se plantean distintas heurísticas para la obtención de planes de ensamblaje en los que puede darse un paralelismo en la ejecución de las tareas de ensamblaje, en función de las restricciones de precedencia y de utilización de recursos comunes.

Por último, se presentan mejoras al algoritmo que permiten un comportamiento más eficiente, desde la detección de simetrías que permitan ignorar una parte importante del espacio de búsqueda, hasta el ahorro de memoria ocupada por los nodos generados en la expansión.

4.2 Descripción del algoritmo

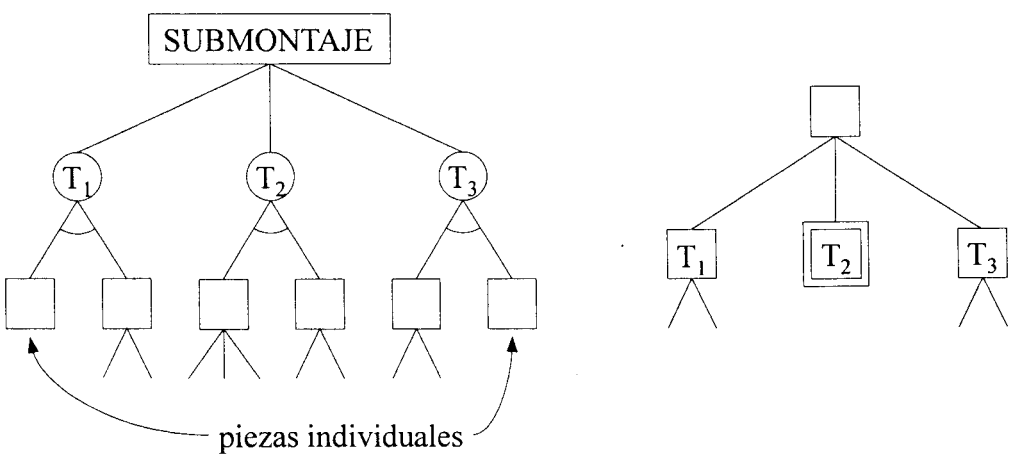
Como se estableció anteriormente, el algoritmo se centra en la elección de un plan de montaje para un producto completo en un sistema de múltiples máquinas manipuladoras, donde los recursos necesarios para llevar a cabo cada tarea representada en el grafo *And/Or* (manipuladores, herramientas...) aparecen como datos, así como los tiempos necesarios para su ejecución. Además de la elección de un plan de montaje, se especifican los órdenes de ejecución de las tareas en cada máquina, mediante un análisis de su ejecución en paralelo en el sistema de ensamblaje dado.

Debido a la estructura del grafo *And/Or*, el problema puede ser estudiado comenzando por el estado final (el producto ya ensamblado) y avanzando hacia el inicial (piezas individuales sin ensamblar).

El algoritmo tiene dos partes bien diferenciadas: una de ellas determina el orden de las tareas de ensamblaje que deben ser ejecutadas de forma secuencial, debido a la propia estructura del grafo *And/Or*, y con independencia de los recursos utilizados para su ejecución. La otra resuelve la posible ejecución en paralelo de tareas de ensamblaje (la representación mediante el grafo *And/Or* permite un estudio natural de este aspecto). Ésta última es de hecho la sección más compleja, debido a que la ejecución de tareas de una parte del ensamblaje total no es independiente del resto, y puede influir en la ejecución de tareas en la otra parte del ensamblaje. Esta dependencia, que se traduce en que el problema no es descomponible, se debe al uso de recursos compartidos por las tareas de distintas ramas del grafo *And/Or*. Es precisamente este aspecto el que impide usar algoritmos de tipo AO* para la resolución del problema planteado, empleados típicamente en problemas con una estructura similar.

Como puede observarse en la Figura 4.2, durante la exploración del grafo *And/Or* nos podemos encontrar con dos tipos de nodos *And*, los correspondientes a las tareas. Comenzando por el nodo raíz, las tareas que se encuentran en el primer nivel (alternativas entre sí), no pueden ejecutarse en paralelo con ninguna otra. Téngase en cuenta que se trata de las tareas que obtendrían el montaje completo del producto. En el caso de que una tarea tenga como uno de los submontajes iniciales a una pieza individual, a continuación la sucederá forzosamente otra tarea, la correspondiente a la formación del submontaje no trivial. Mientras en el descenso por el grafo *And/Or* nos sigamos encontrando con nodos *And* (tareas) que tengan como uno de los submontajes de partida a una pieza individual, las tareas se ejecutarán de forma secuencial. A los nodos correspondientes a la inclusión de estas tareas en el plan de ensamblaje los llamaremos nodos *secuenciales*. En la Figura 4.2, las tareas T_1, T_2 y T_3 tendrán nodos secuenciales en el árbol de expansión. Asimismo, las tareas sucesoras de T_1 y T_3 en el grafo *And/Or* también tendrán su correspondiente nodo secuencial en el árbol de expansión. Al proceso de ensamblaje en el que todas las tareas son secuenciales en el sentido descrito se conoce como ensamblaje *lineal*.

Por otro lado, en cuanto nos encontremos con una tarea para la cual los dos submontajes de partida sean no triviales, las tareas por debajo de aquella podrían ejecutarse en paralelo, en función de los recursos utilizados para ellas. Así, si se usan recursos compartidos, habrá que especificar el orden de uso de los mismos por las diferentes tareas, si éstas no están ligadas mediante relaciones de precedencia. Aparte de ello, todavía podrían permanecer tareas alternativas por debajo de la considerada en el grafo *And/Or*, por lo que el problema de la selección también ha de resolverse. Como se aprecia, el tratamiento de este tipo de nodos es más complejo, y en sí representa un problema de una mayor complejidad conceptual que el secuencial. A los nodos correspondientes a las tareas que cumplen con el requisito anterior los denominaremos nodos *paralelos*. En la Figura 4.2, la tarea T_2 tiene su correspondiente nodo secuencial



(a) Grafo *And/Or* (b) Árbol de expansión

Figura 4.2: Distintos tipos de nodos.

en el árbol de búsqueda, pero su expansión no generaría nodos secuenciales. Para completar la solución por debajo de esa tarea se usará un algoritmo secundario, también de tipo A*. De hecho, los mayores esfuerzos se han centrado en resolver tal subproblema.

Con el objeto de expandir el menor número de nodos y evitar nodos redundantes se han usado funciones heurísticas basadas en la ejecución únicamente de las tareas que están por debajo del nodo en cuestión en el grafo *And/Or*, y el tiempo restante para el uso de herramientas y máquinas (suponiendo el mínimo número de cambios de herramientas, para mantener el algoritmo como A*). Asimismo, se han propuesto nuevas heurísticas que mezclan información de ambos tipos.

Debido a que, una vez resuelto un nodo paralelo, existe un límite superior para el tiempo total de ensamblaje (*makespan*), el algoritmo "paralelo" no necesita terminar cuando el mejor coste esperado supera ese límite.

4.2.1 Representación del espacio de estados

En el proceso de expansión del algoritmo, un nodo representará el estado correspondiente a la ejecución de las tareas que han sido incluidas de forma sucesiva a lo largo de los pasos anteriores, que dieron lugar a los nodos predecesores en el árbol de expansión. El orden de inclusión de tareas indicará, además, el orden de ejecución de las mismas. Así, una tarea no deberá ser incluida hasta que su predecesora en el grafo *And/Or* lo haya sido. Las restricciones de precedencia se cumplen, pues, al seguir esta estrategia de exploración del grafo *And/Or*. Éste será el único condicionante para las tareas secuenciales.

Para las tareas que no están relacionadas mediante precedencia, cuando usen un recurso compartido, el orden de inclusión se corresponderá con el orden de ejecución. Cuando los recursos usados sean independientes, las tareas podrán ejecutarse simultáneamente.

El estado asociado a un nodo de expansión puede describirse indicando cuál ha sido la última tarea incluida, y guardando memoria de los nodos ancestros en el árbol de expansión. Son de interés, además, el tiempo correspondiente a la última utilización de cada máquina M, *lastTime* (n, M), así como la última herramienta que se usó en ella, *lastTool* (n, M).

En cuanto a los tiempos asociados a la última tarea introducida, serán tales que la tarea se ejecute lo antes posible, teniendo en cuenta el estado anterior. Se considerará, por un lado, que la tarea se deberá ejecutar con posterioridad a la finalización de su tarea predecesora, teniendo en cuenta el posible tiempo asociado al transporte del submontaje intermedio desde una máquina a otra. Por otro lado, la ejecución de la tarea se realizará con posterioridad al último tiempo de uso de la máquina asociada a la tarea, teniendo en cuenta el posible cambio de herramienta que pueda ser necesario. Así, el tiempo de comienzo de T, última tarea incluida en la solución parcial asociada al nodo en cuestión, vendrá dado por la ecuación:

$$begin(T) = \max\Big(end\big(pred(T)\big) + \Delta_{mov}\big(sa(T), M(T), M\big(pred(T)\big)\big), \\ lastTime\big(n, M(T)\big) + \Delta_{cht}\big(M(T), H(T), lastTool\big(n, M(T)\big)\big)\Big) \quad (4.2)$$

En la ecuación anterior hay que tener en cuenta los posibles casos particulares. Uno de ellos ocurrirá para la primera tarea, que por ser tal no tendrá predecesora, ni se habrá usado ninguna máquina hasta ese momento. Es evidente que su tiempo de comienzo debe tomarse como cero, o el origen de tiempos[2]. Por otro lado, cuando la máquina correspondiente a T no haya sido usada por ninguna predecesora, sólo estará dispuesta a cumplir la restricción de precedencia con respecto a su predecesora, así como el posible movimiento del submontaje asociado.

Lógicamente, el tiempo de finalización de T vendrá dado por

$$end(T) = begin(T) + dur(T) \quad (4.3)$$

ecuación que se cumplirá para cualquier tarea, ya que se está suponiendo que no son interrumpibles.

4.2.2 Ejecución Secuencial de Tareas

Para obtener el programa de ensamblaje completo del producto se usa el algoritmo *global-A**, que se muestra en la Figura 4.3. En él se distinguen dos listas[3] de abiertos, para incluir por separado a los nodos *secuenciales* y a los *paralelos*, a las que se les ha llamado ABIERTOS_SEC y ABIERTOS_PAR respectivamente. El algoritmo comienza incluyendo en ABIERTOS_SEC un nodo secuencial correspondiente a un estado en el que no se ha considerado aún ninguna tarea y que prepara la búsqueda a partir del nodo raíz del grafo *And/Or*. A diferencia de los nodos paralelos, que se tratarán más adelante, en los nodos secuenciales se guarda memoria del correspondiente nodo *Or* del grafo *And/Or*, que servirá para determinar las tareas sucesoras en el proceso de expansión del algoritmo.

Como se ha indicado, pueden generarse *dos tipos de nodos*, dependiendo de los nodos *Or* de destino de cada nodo *And* escogido. Si al menos uno de esos nodos *Or* corresponde a una pieza individual, como en el caso de la Figura 4.4, el proceso de ensamblaje continuará siendo secuencial. El nodo resultante de la expansión podrá ser tratado como el inicial, siendo el nodo correspondiente al submontaje no trivial el que haga las veces del nodo raíz. En este caso, pues, los nodos generados son también secuenciales y son llevados a la lista ABIERTOS_SEC.

---

[2] Téngase en cuenta que, al realizar la inversión necesaria para obtener la secuencia de ensamblaje, esto supondrá que al final de! proceso la última máquina usada quedará con la última herramienta usada instalada en ella.

[3] La formulación original del algoritmo las denotaba como listas. Se ha demostrado que una implementación más óptima se corresponde con *montículos* de mínimos, debido a las operaciones que se usan más a menudo con las mismas.

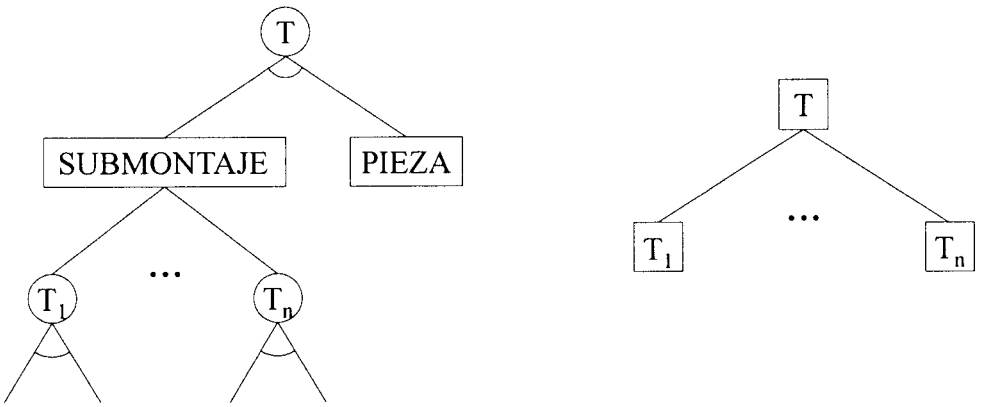
```
Algoritmo global-A* (grafoAndOr) devuelve solución
  situar el nodo inicial en ABIERTOS_SEC
  iniciar ABIERTOS_PAR con la lista vacía
  mejorTiempo := ∞
  solución := NULA
  mientras haya elementos en ABIERTOS_SEC y ABIERTOS_PAR
    escoger el mejor nodo de entre ABIERTOS_SEC y ABIERTOS_PAR
    si el nodo escogido es de ABIERTOS_SEC
      si los dos nodos And se corresponden con piezas individuales
        construir la solución a partir del nodo secuencial
        vaciar las listas ABIERTOS_SEC y ABIERTOS_PAR
      si no, si uno de los nodos And se corresponde con un submontaje no trivial
        expandir el nodo, incluyendo los nodos sucesores en ABIERTOS_SEC
      si no, si los dos nodos And se corresponden con un submontaje no trivial
        generar los nodos paralelos a partir del nodo e incluirlos en
        ABIERTOS_PAR
      fsi
    si no (el nodo escogido es de ABIERTOS_PAR)
      usar el algoritmo paralelo-A* para obtener la mejor solución asociada al
      nodo, solucPar, y su tiempo total, tiempoSolucPar
      si tiempoSolucPar < mejorTiempo
        mejorTiempo := tiempoSolucPar
        solución := solucPar
        eliminar de ABIERTOS_SEC y ABIERTOS_PAR todos los nodos peores
        que solucPar
      fsi
    fsi
  fmientras
  devolver solución
```

Figura 4.3: Algoritmo *global-A.**

La función de evaluación usada para los nodos secuenciales es

$$f(n) = g(n) + h(n) \tag{4.4}$$

siendo $g(n)$ el tiempo acumulado en la ejecución de las tareas correspondientes al estado del nodo n, incluyendo los retardos asociados a los movimientos de submontajes y a los cambios de herramientas necesarios, y $h(n)$ una estimación optimista del tiempo restante



(a) Grafo *And/Or* (b) Árbol de expansión

Figura 4.4: Tratamiento de nodos secuenciales.

para completar el proceso global. Para que la heurística sea admisible, y, por tanto, tengamos garantías de encontrar la solución óptima, $h(n)$ debería ser un límite inferior del tiempo restante para completar el plan de montaje. Caben dos alternativas para obtener una estimación optimista. Si se quiere un cálculo fácil y casi inmediato, se debe buscar una expresión dependiente de los tiempos de ejecución de las tareas restantes y del número de ellas. Debe tenerse en cuenta, además, que la topología del grafo *And/Or* puede ser muy variada. Así, para acertar en la estimación podría ser necesario un estudio de la misma, por lo que se puede suponer el caso correspondiente a una topología en la que las tareas vienen ordenadas minimizando el nivel de profundidad. La disposición de las tareas sería la equivalente a un árbol binario equilibrado. Con estos condicionantes, se ha escogido

$$h(n) = durMin \cdot \lfloor \log_2 (a(n)+1) \rfloor \tag{4.5}$$

siendo $a(n)$ el número de tareas necesarias para completar el plan de montaje, y *durMin* la duración mínima de las tareas. Nótese que $a(n)$ se corresponde con el número de piezas del submontaje menos el número de tareas incluidas en el estado asociado al nodo n menos uno.

Como puede observarse, se han despreciado los retardos asociados a los posibles cambios de herramientas, ya que, sin un estudio más detallado, es imposible determinar el mínimo número de cambios de herramientas que se necesitaría, por lo que se asume como nulo al estimar $h(n)$.

Tampoco es fácil determinar a priori los retardos asociados a los movimientos de los submontajes intermedios de unas máquinas a otras. En principio podría concluirse que si las tareas se pueden ejecutar en paralelo, como se extrae de la expresión (4.5), entonces serán necesarios movimientos entre máquinas. Sin embargo, el número de ellos que se producirán de manera secuencial, y, por tanto, dando lugar a la acumulación de retardos para la estimación buscada, no es trivial. Así, por ejemplo, para 4 tareas

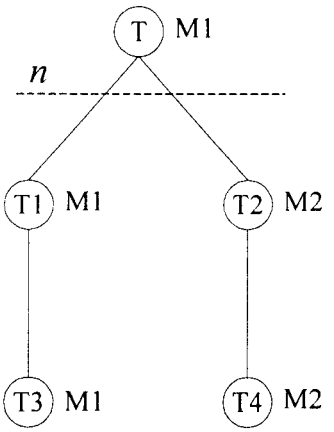


Figura 4.5: Configuración óptima para mínimos movimientos de submontajes.

restantes, a lo más que podría llegarse a decir es a que como mínimo se necesitaría un movimiento entre dos máquinas, si las tareas pudieran realizarse en paralelo, correspondiente a la configuración de la Figura 4.5. En tal caso, la estimación debería tomarse a partir de la expresión $h(n) = \min(4 \cdot durMin, 2 \cdot durMin + \Delta_{mov})$. En general, habría que distinguir si el árbol balanceado tiene más o menos completo el último nivel para precisar en la estimación. Toda esta complicación no merece la pena en general, como se observará a continuación, por lo que la expresión (4.5) puede tomarse como aceptable.

Como se ha visto, la estimación anterior puede obtenerse muy rápidamente. Esto puede ser suficientemente útil si existen pocos nodos secuenciales, es decir, la estructura del grafo *And/Or* es tal que se generan nodos paralelos muy cercanos a la raíz. Para el caso de ensamblajes muy secuenciales, en último extremo lineales, tal estimación podría resultar muy pobre. En tal situación puede merecer la pena realizar una estimación con un mayor gasto computacional. La siguiente expresión representa una estimación optimista para $h(n)$, dando lugar a una heurística admisible:

$$h(n) = \max(\min_{T_i \in Or_1(n)} (hs(T_i)), \min_{T_j \in Or_2(n)} (hs(T_j))) \tag{4.6}$$

En la ecuación (4.6), $Or_1(n)$ y $Or_2(n)$ representan los nodos *Or* correspondientes al nodo n, y la expresión $T \in Or(n)$ indica que la tarea T produce como ensamblaje resultante el correspondiente al nodo $Or(n)$. En la citada expresión, $hs(T)$ representa el tiempo mínimo necesario para ejecutar T y un subconjunto de sus sucesoras escogidas para completar una solución válida, y que viene dada por

$$hs(T) = dur(T) + \max(\min_{T_i \in Or_1(T)} (hs(T_i)), \min_{T_j \in Or_2(T)} (hs(T_j))) \tag{4.7}$$

en donde $Or_1(T)$ y $Or_2(T)$ son los nodos *Or* correspondientes a los submontajes de partida de la tarea T.

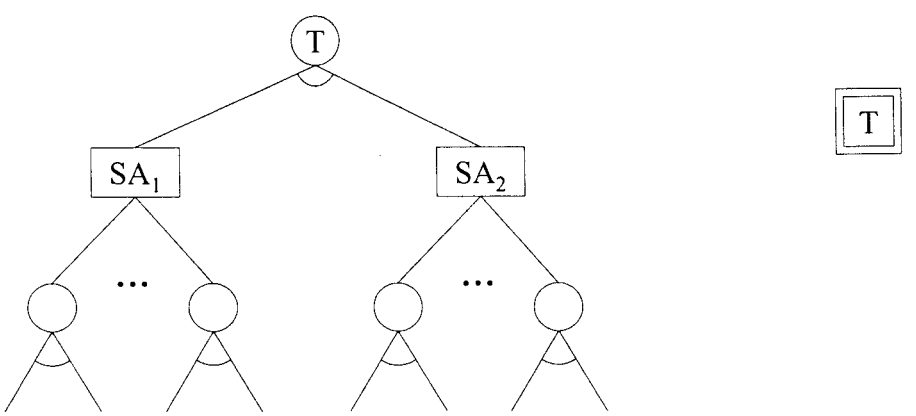
Nótese que en las ecuaciones (4.6) y (4.7) ha de tenerse en cuenta el caso particular que se produce cuando el nodo *Or* es una hoja, por lo que la expresión del mínimo correspondiente se debe tomar como cero.

Es preciso hacer observar que los cálculos necesarios para $hs(T)$ sólo requieren un recorrido en profundidad del grafo *And/Or*, que puede realizarse antes de comenzar el algoritmo *global-A**. Una vez calculadas estas cantidades, el cálculo en línea correspondiente a (4.6) es de un nivel de complejidad similar al que teníamos en (4.5).

En la última estimación no se han considerado los retardos correspondientes a los movimientos de submontajes intermedios y a los cambios de herramientas que podrían ser necesarios. En el Capítulo 6 se considerará tal circunstancia, por lo que los resultados allí obtenidos se podrán utilizar también aquí.

4.2.3 Ejecución Paralela de Tareas

En el algoritmo *global-A**, cuando el nodo secuencial escogido para su procesamiento tiene sus dos nodos *Or* correspondientes a submontajes no triviales, como en el caso de la Figura 4.6, se debe realizar el estudio del posible paralelismo de las tareas restantes. Se plantean dos estrategias posibles. El algoritmo podría en cada paso añadir una nueva tarea y no decidir nada todavía respecto a las tareas que vendrán por debajo de ellas en el grafo *And/Or*, en el sentido de seleccionarlas para formar parte del plan. Las heurísticas posibles según esta estrategia estarían basadas en las ecuaciones (4.6) y (4.7). Como se verá en el siguiente apartado, es posible estimar con mayor precisión el tiempo necesario para las tareas restantes si se separan los distintos planes de montaje posible. De esta forma, para un plan de montaje en el que no aparecen tareas alternativas, no se deberá realizar la operación correspondiente al mínimo de las posibilidades, por lo que al final se tendrá una estimación más exacta. A cambio, se deben contemplar todos los árboles de tareas. La mejora obtenida por las heurísticas deberá contrarrestar este aspecto.



(a) Grafo *And/Or* (b) Árbol de expansión

Figura 4.6: Tratamiento de nodos paralelos.

Así, para los nodos secuenciales referidos en el párrafo anterior, se generan todos los árboles de precedencia de tareas, considerando a cada uno de ellos como un nodo paralelo, que se incluirán en la lista ABIERTOS_PAR. La ordenación de ellos se realizará en función de las heurísticas utilizadas, aspecto que se tratará en el siguiente apartado.

Según se indicó en el algoritmo *global-A**, el procesamiento de los nodos paralelos viene dado por el algoritmo *paralelo-A**. Este algoritmo responde al esquema clásico, en donde tendremos una lista ABIERTOS conteniendo a los nodos sin expandir. Estos nodos, al igual que se tenía en el subapartado 4.2.1, tienen asociado un estado correspondiente a la ejecución de las tareas que han sido incluidas en pasos anteriores.

En cada paso de expansión deben cubrirse las distintas alternativas a considerar. Ahora las opciones se corresponden con el orden en que aparecen las tareas en la solución, ya que todas las pertenecientes al árbol de precedencias deben estar presentes en la misma. La estrategia es la misma que se estableció en el algoritmo *global-A**. Así, se incluye una sola tarea en cada paso de expansión, de manera que se ejecutará lo antes posible. Para determinar esto último, se tiene en cuenta el estado de partida, es decir, las tareas que ya fueron incluidas, y cómo afectaron éstas al uso de los recursos.

Para facilitar el proceso de expansión, una indicación de cuál o cuáles tareas pueden ser incluidas en el próximo paso de expansión será útil incluir en el nodo. A estas tareas las denominaremos "*candidatas*". Asociado con cada tarea candidata se indicará su tiempo de comienzo más temprano si fuera introducida en el siguiente paso de expansión, *est* (*n*, *T*). Para cada tarea *T* candidata a ser incluida en el plan de ensamblaje, su tiempo de comienzo más temprano viene dado por la expresión:

$$est(n,T) = \max\Big(end\big(prec(T)\big) + \Delta_{mov}\big(sa(T), M(T), M\big(pred(T)\big)\big), \\ lastTime\big(n, M(T)\big) + \Delta_{chl}\big(M(T), H(T), lastTool\big(n, M(T)\big)\big)\Big) \tag{4.8}$$

La inclusión de una tarea en el plan de montaje determinará un nuevo nodo de expansión cuyo estado será el mismo del nodo del cual se ha generado, en el que se modificará el estado asociado a la estación de trabajo usado por la tarea. Asimismo, se añadirán como tareas candidatas las sucesoras de la introducida en el árbol de precedencia. Los tiempos de comienzo más tempranos pueden verse modificados en la medida que afecte el cambio en la máquina usada. Sólo se añade una nueva tarea de ensamblaje, y su tiempo de procesamiento afectará solamente a una de las estaciones de trabajo, manteniéndose para las demás el mismo estado.

Como se verá en el apartado 4.4, cuando dos tareas candidatas sean independientes en el uso de los recursos (diremos que son *compatibles*, y refinaremos el concepto), el orden de inclusión de las tareas en la solución será indiferente. Para mejorar el comportamiento del algoritmo será necesario detectar tales simetrías y generar el menor número de nodos en la búsqueda. Para ilustrar el problema de fondo, supóngase que todas las tareas pudieran ejecutarse en paralelo. La solución al problema es trivial, por

lo que no sería necesario generar distintos nodos alternativos en cada etapa de expansión.

La función de evaluación para los nodos obtenidos por este algoritmo es de la forma $f(n) = g(n) + h(n)$, siendo ahora $g(n)$ el mayor de los tiempos de comienzo más tempranos de las tareas candidatas presentes en n, expresadas a través del conjunto $cand(n)$, y los tiempos finales de las tareas ya terminadas en n sin sucesoras. Esto último se traduce en mirar los tiempos finales de uso de las máquinas de ensamblaje:

$$g(n) = \max\left(\max_{T_i \in cand(n)} \left(est(n, T_i) \right), \max_{máquinas} \left(lastTime(n, M_i) \right) \right) \tag{4.9}$$

La función $h(n)$ deberá indicar una estimación optimista del tiempo necesaria para las tareas que aún restan por incluir en la solución, tal como se verá en el apartado siguiente.

La Figura 4.7 muestra un árbol de precedencia de tareas, diferentes nodos de expansión e información acerca de sus estados correspondientes. Vienen también acompañados de los diagramas de Gantt. Puede observarse en la misma los retardos asociados a las distintas causas que pueden motivarlo: precedencia entre tareas y uso exclusivo de recursos compartidos (*prec* en la figura), movimientos de submontajes

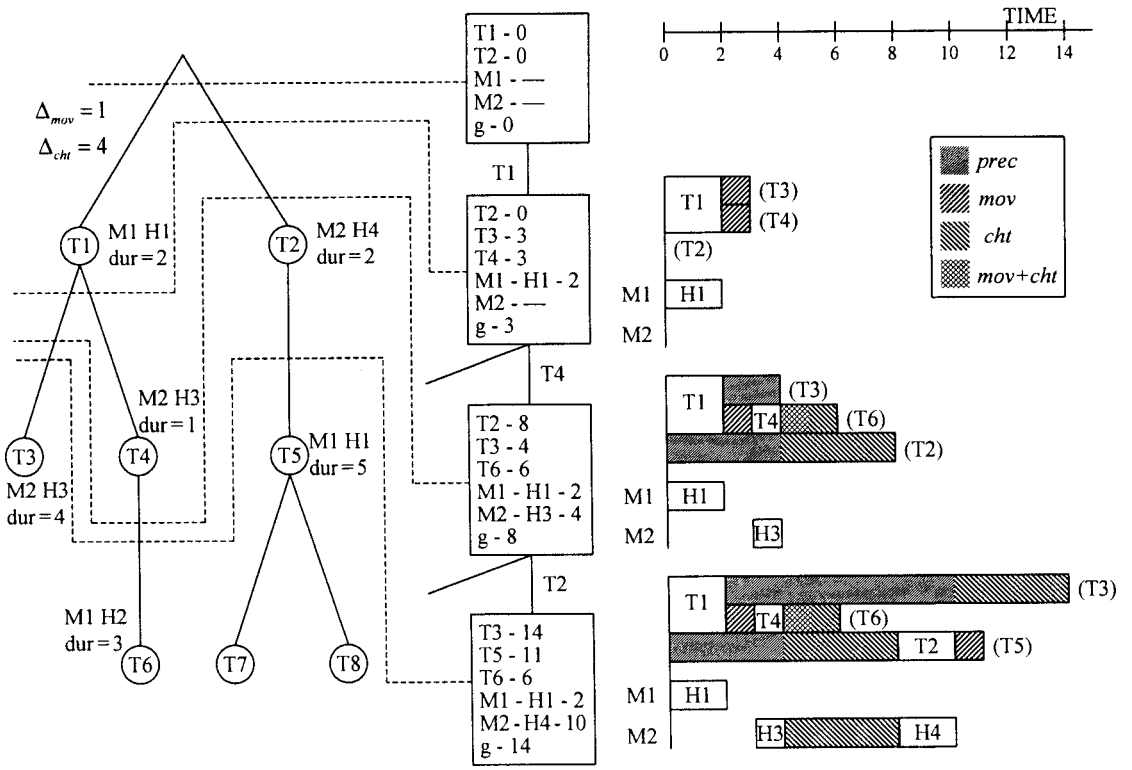


Figura 4.7: Un árbol de precedencia de tareas, algunos nodos de expansión, y sus diagramas de Gantt correspondientes.

recursos compartidos (*prec* en la figura), movimientos de submontajes intermedios (*mov*), y cambios de herramientas en las máquinas de ensamblaje (*cht*).

4.3 Heurísticas

El objetivo en este apartado es establecer una estimación optimista para el tiempo total necesario para ejecutar las tareas incluidas en un árbol de precedencias, que servirá para ser usado en el algoritmo *paralelo-A**. Como es sabido, se trata de que tal estimación no suponga un costo de computación excesivo, lo cual se consigue relajando ciertas restricciones del conjunto que definen el problema completo. Resuelto el problema de la selección de tareas, las restricciones restantes pueden englobarse en dos grupos. En uno de ellos tendríamos aquellas relacionadas con la precedencia entre tareas, incluyendo los posibles retardos añadidos entre la ejecución de unas tareas y otras. Por otro, las restricciones asociadas al uso de recursos compartidos. En los dos subapartados siguientes se definen sendas heurísticas según esta separación. En los subsiguientes, se proponen nuevas heurísticas considerando restricciones de ambos tipos.

4.3.1 Heurística h_1: precedencia de tareas

Como se ha indicado, se trata de calcular una estimación del tiempo que, como mínimo, se necesitará para ejecutar cada tarea candidata incluida en el nodo de expansión n y sus sucesoras en el árbol de precedencia. En esta estimación no se tendrá en cuenta las posibles interdependencias entre tareas pertenecientes a distintas ramas del árbol de precedencias. En el cálculo de tal estimación, la parte más costosa computacionalmente puede realizarse una sola vez para todas las tareas del árbol, mediante un algoritmo recursivo que recorrería en profundidad el árbol de precedencias y que se ejecutaría antes del propio algoritmo *paralelo-A**.

Según la definición dada para $g(n)$, es conveniente tener en cuenta las holguras relacionadas con cada tarea candidata, que servirán para enlazar los tiempos calculados en el estado asociado al nodo con las estimaciones de tiempos correspondientes a las tareas aún no incluidas. Así, para cada tarea $T_i \in cand(n)$, se tendrá una holgura

$$e(n,T_i) = g(n) - est(n,T_i) \tag{4.10}$$

La función heurística $h_1(n)$ se define como

$$h_1(n) = \max\left(0, \max_{T_i \in cand(n)} \left(h_1(T_i) - e(n,T_i)\right)\right) \tag{4.11}$$

donde $h_1(T_i)$ representa el tiempo mínimo necesario para ejecutar T_i y sus sucesoras en el árbol de precedencias, y no depende del nodo n. En la expresión anterior, el cero

aparece para contemplar el que las tareas restantes no usen alguna de las máquinas y el tiempo final de uso de la misma pueda superar al del final de ejecución de esas tareas.

Teniendo en cuenta únicamente las duraciones de las tareas del árbol de precedencias, junto con las propias restricciones de precedencias definidas en el mismo, una primera definición de $h_1(T)$ viene dada por

$$h_1(T) = dur(T) + \max_{T_i \in suc(T)} \left(h_1(T_i) \right) \tag{4.12}$$

teniendo en cuenta que cuando no haya tareas sucesoras la expresión del máximo sería cero.

Para contemplar los posibles retardos asociados a los cambios de herramientas, vamos a usar la función $\tau(T, M, H)$, que representa el tiempo *adicional* al necesario para la ejecución de T y sus sucesoras, si antes de la ejecución de T la herramienta H está instalada en la máquina M. De acuerdo con esto, se puede refinar la definición para $h_1(T)$, siendo ahora:

$$h_1(T) = dur(T) + \max_{T_i \in suc(T)} \left(h_1(T_i) + \tau\left(T_i, M(T), H(T)\right) \right) \tag{4.13}$$

En la Figura 4.8 se muestra el efecto considerado en la definición anterior.

La función $\tau(T, M, H)$ viene dada por la ecuación:

$$\tau(T, M, H) = \begin{cases} \Delta_{cht}\left(M, H(T), H\right) & \text{si } M = M(T) \\ \max\left(0, \max_{T_i \in suc(T)} \left(h(T_i) + \tau\left(T_i, M(T), H(T)\right) - h_1(T) \right)\right) & \text{si } M \neq M(T) \end{cases} \tag{4.14}$$

Nótese que, por un lado, $\Delta_{cht}\left(M, H(T), H\right)$ es cero si $H = H(T)$, por lo que no hay

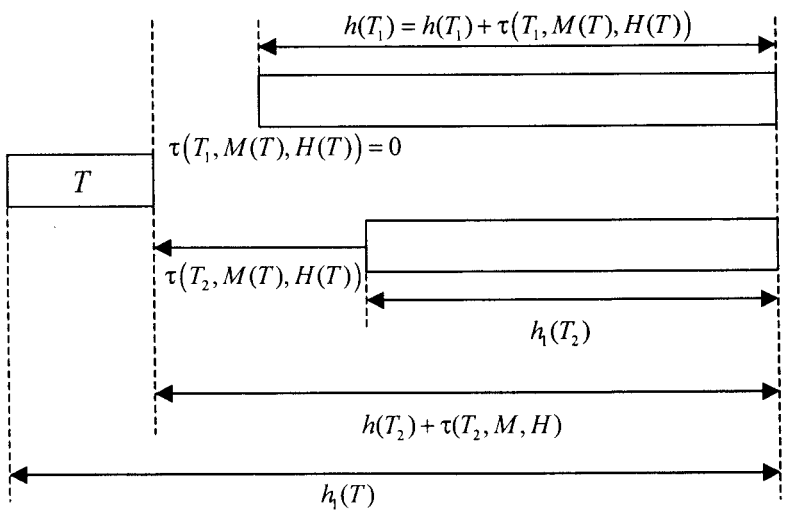


Figura 4.8: Cálculo de $h_1(T)$.

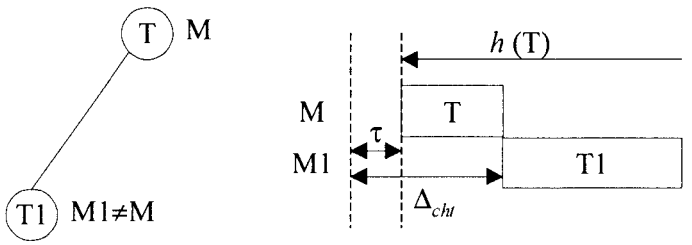


Figura 4.9: Ilustración de $\tau > 0$, con $M \neq M(T)$.

retardo adicional si no hay cambio de herramienta. Por otro lado, cuando $M \neq M(T)$, para que el retardo sea positivo, la duración de la tarea en cuestión deberá ser menor que la duración asociada al cambio de herramienta (ver Figura 4.9). De esta forma, en el ejemplo mostrado en la Figura 4.8, el valor cero de $\tau(T_1, M(T), H(T))$ puede deberse bien a que T y T_1 utilicen la misma máquina y herramienta, o bien porque, usando T_1 una máquina distinta que T, la duración de T_1 sea lo suficientemente grande como para compensar un posible cambio de herramienta necesario por el uso de la misma máquina y otra herramienta distinta a $H(T)$ en alguna tarea sucesora de T_1. Por otro lado, el valor positivo de $\tau(T_2, M(T), H(T))$ puede deberse bien a que T y T_2 utilicen la misma máquina y distinta herramienta, o bien porque, usando T_2 una máquina distinta que T, la duración de T_2 no sea lo suficientemente grande como para compensar un posible cambio de herramienta necesario por el uso de la misma máquina y otra herramienta distinta a $H(T)$ en alguna tarea sucesora de T_2.

Si, además, se quieren considerar los retardos asociados al movimiento de submontajes intermedios, el término que deberá sumarse a $h_1(T_i)$ será el valor máximo entre $\tau(T_i, M(T), H(T))$ y $\Delta_{mov}(sa(T_i), M(T_i), M(T))$, valor éste último correspondiente al retardo asociado al posible movimiento del submontaje que se obtiene en la ejecución de T_i, $sa(T_i)$, entre las máquinas donde se ejecutan T_i y T. Llamando

$$\tau_{mov}(T_i, T) = \max\left(\tau(T_i, M(T), H(T)), \Delta_{mov}(sa(T_i), M(T_i), M(T))\right) \qquad (4.15)$$

la definición para $h_1(T)$ sería ahora:

$$h_1(T) = dur(T) + \max_{T_i \in suc(T)} \left(h_1(T_i) + \tau_{mov}(T_i, T)\right) \qquad (4.16)$$

En las expresiones anteriores, cuando $M(T_i) = M(T)$, no habrá movimiento del submontaje intermedio, por lo que el retardo que podría darse sería el debido a un eventual cambio de herramientas. Si por el contrario $M(T_i) \neq M(T)$, entonces sí se deberá trasladar el submontaje intermedio desde la máquina $M(T_i)$ hacia $M(T)$. Para que afecte más un cambio de herramienta deberá ocurrir que $\Delta_{cht} > 2 \cdot \Delta_{mov} + dur(T_i)$, suponiendo que Δ_{cht} representa la duración del cambio de herramienta necesario en $M(T)$ y Δ_{mov} la duración del traslado de los submontajes intermedios involucrados en T_i (inicial y final), supuestos iguales. Esta misma circunstancia se observa en el ejemplo de la Figura 4.7, en donde entre la ejecución de T_1 y T_6 debe transcurrir el tiempo asociado

al cambio de herramienta, que es superior al tiempo acumulado de ejecución de la tarea intermedia T_4 más los traslados de los submontajes resultantes de T_6 y T_4.

Es fácil comprobar que todas las definiciones dadas para $h_1(T)$ representan un límite inferior para el tiempo necesario para ejecutar T y sus sucesoras en el árbol de precedencias, por lo que la heurística resultante es admisible.

4.3.2 Heurística h_2: utilización de recursos

El hecho de que las restricciones de precedencia de tareas vengan recogidas en forma de árbol de precedencias nos conduce a tener que realizar un estudio más complejo que si las restricciones fueran estrictamente secuenciales, en cuyo caso la heurística h_1 serviría para establecer la información buscada. En el caso general, puede existir una fuerte dependencia entre tareas pertenecientes a distintas ramas del árbol de precedencias, es decir, no vienen ligadas por ninguna restricción de precedencia. Tal dependencia se debe al posible uso exclusivo de un recurso compartido: el de la máquina de ensamblaje.

El enfoque para esta nueva heurística es diametralmente opuesto al que se tuvo para h_1. Ahora se tendrán en cuenta los tiempos restantes de utilización de cada herramienta en cada máquina, suponiendo además un número de cambios de herramientas mínimo. No se tendrá en cuenta la precedencia de tareas bajo ningún aspecto.

Nuevamente es útil considerar la holgura entre $g(n)$ y el último instante de uso de cada máquina en el estado asociado al nodo n:

$$e(n,M) = g(n) - lastTime(n,M) \tag{4.17}$$

Según esto, la definición de la heurística h_2 viene dada por la expresión

$$h_2(n) = \max_{máquinas} \left(h_2(n,M_i) - e(n,M_i) \right) \tag{4.18}$$

siendo $h_2(n, M)$ una estimación optimista del tiempo restante total de uso de la máquina M para las tareas aún no incluidas en el estado asociado a n. Tal estimación puede ofrecerse mediante la expresión

$$h_2(n,M) = \left(\sum_{H_j \in M} \sum_{T_i \in cand(n)} h_2(T_i,H_j) \right) + \Sigma\left(n, \Delta_{chl}(M)\right) \tag{4.19}$$

donde $h_2(T, H)$ representa el tiempo total de uso de la herramienta H por T y sus sucesoras en el árbol de precedencias, y viene dado por la ecuación

$$h_2(T,H)=\begin{cases} dur(T)+\sum_{T_i\in suc(T)} h_2(T_i,H) & \text{si } H=H(T) \\ \sum_{T_i\in suc(T)} h_2(T_i,H) & \text{si } H\neq H(T) \end{cases} \tag{4.20}$$

Pueden definirse también otras cantidades, según las distintas sumas parciales de la ecuación (4.19). Así,

$$h_2(n,H)=\sum_{T_i\in cand(n)} h_2(T_i,H) \tag{4.21}$$

es el tiempo total de uso de la herramienta H para todas las tareas aún no incluidas en el estado asociado a n, y

$$h_2(T,M)=\left(\sum_{H_j\in M} h_2(T,H_j)\right)+\Sigma\left(T,\Delta_{cht}(M)\right) \tag{4.22}$$

es el tiempo total de uso de la máquina M para la tarea T y sus sucesoras en el árbol de precedencias.

En (4.19) el término $\Sigma\left(n,\Delta_{cht}(M)\right)$ representa una estimación optimista del tiempo total empleado en los cambios de herramientas producidos en la máquina M. Sin querer entrar en un estudio detallado sobre las posibles alternativas de uso de las distintas herramientas, se puede suponer que en el caso más favorable, una vez instalada una herramienta en una máquina, todas las tareas que la usan van a ejecutarse de forma consecutiva. De esta forma, el cálculo de $\Sigma\left(n,\Delta_{cht}(M)\right)$es equivalente al problema del viajante, considerando las herramientas con algún uso pendiente como los nodos a visitar y el nodo de partida el correspondiente a la última herramienta usada en la máquina. Las distancias entre nodos equivaldrían a la duración del cambio de herramientas.

En (4.22) el término $\Sigma\left(T,\Delta_{cht}\left(M_i\right)\right)$ representa el tiempo mínimo necesario para los cambios de herramienta que se deben producir en la máquina M_i para ejecutar la tarea T y sus sucesoras en el árbol de precedencias. Para su cálculo, ahora no hay que partir de una herramienta previamente instalada en la máquina, ya que queremos que la estimación sea independiente del nodo de expansión. Excepto en lo anterior, se procede como se indicó en el párrafo anterior.

Si en la estimación de $\Sigma\left(n,\Delta_{cht}(M)\right)$ se supone como *simplificación adicional* que los tiempos de cambio de herramientas no dependen del tipo de herramienta, bastará con determinar el número de herramientas que restan por usar en cada máquina, $NH(n, M)$. Así, si llamamos $\Delta_{cht}(M)$ a la duración del cambio de herramienta en M,

$$\Sigma(n,\Delta_{cht}(M))=\begin{cases} \Delta_{cht}(M)\cdot NH(n,M) & \text{si } h_2\left(n,lastTool(n,M)\right)=0 \\ \Delta_{cht}(M)\cdot\left(NH(n,M)-1\right) & \text{si } h_2\left(n,lastTool(n,M)\right)>0 \end{cases} \tag{4.23}$$

donde debe tenerse en cuenta que en el caso particular de que aún no haya sido usada la máquina M, no debe computarse un cambio de herramienta adicional, con lo que estaríamos en el segundo caso de (4.23).

La función $h_2(n)$ podría ser mejorada usando el tiempo de uso más temprano de M en lugar de $e(n, M)$, calculado a partir de los tiempos de comienzo más temprano de las tareas candidatas. Nótese que en el caso de que ninguna de las tareas candidatas use la máquina en cuestión, o bien no se toma en cuenta o bien podrían ser consideradas las tareas no incluidas en n y que no son candidatas. Las heurísticas que se definirán más adelante tienen en cuenta este aspecto.

4.3.3 Heurística h_3: estimación del número de cambios de herramientas a partir del árbol de precedencias

Como se vio antes, la heurística h_2 consideraba, para cada máquina, el tiempo total de uso de todas las herramientas. Además, despreciaba cualquier consideración asociada a las restricciones de precedencia de tareas en la estimación de los posibles cambios de herramientas que pudieran sucederse. De esta forma, para mantener la admisibilidad de la heurística, se suponía el caso más favorable, en el que todos los usos de una misma herramienta iban a ser consecutivos, minimizando así el número de cambios de herramientas. Por otro lado, en la heurística h_1 únicamente se tiene en cuenta la precedencia entre tareas, despreciando el efecto asociado al uso de recursos compartidos. En este apartado se propone una mejora de la heurística h_2, de manera que en la estimación del número de cambios de herramientas se tenga en cuenta la precedencia entre las tareas que usan distintas herramientas asociadas a una misma máquina. Esto puede entenderse como una mezcla de la información que recogen las heurísticas h_1 y h_2, es decir, información de las restricciones de precedencia de tareas y del uso de recursos compartidos.

Buscamos un límite inferior del número de cambios de herramientas que se producirán en la ejecución de un árbol de precedencias. La solución adoptada en la heurística h_2 es de las más simples (obviamente todavía existe otra solución más simple: no considerar cambios de herramientas), ya que, al no considerar las precedencias de las tareas, debía contemplar una hipotética situación en la que la utilización de las distintas herramientas se pueda realizar instalándolas una sola vez en la máquina correspondiente, lo que nos daría que el número de cambios de herramientas $NCH(n, M)$ sería igual al número de herramientas $NH(n, M)$ que deben usarse para las tareas que faltan por completar el plan ($NH(n, M) - 1$ si la última herramienta usada en n para M debe usarse de nuevo).

Lo que se pretende ahora es encontrar una estimación más realista del límite inferior del número de cambios de herramientas que se necesitarán. Para ello se considerarán las restricciones de precedencia de tareas para valorar con mayor exactitud el número de cambios de herramientas que se precisarán, siempre mediante estimaciones optimistas. En esa valoración, no se tendrá en cuenta la duración de las tareas, sino solamente el uso por parte de las tareas de unas herramientas u otras.

Teniendo en cuenta que el número de secuencias de herramientas crece exponencialmente con el tamaño del árbol de precedencias, se precisa un modelo que recoja la información más relevante del conjunto de las secuencias que representen a aquellas en las que el número de cambios de herramientas sea menor. Para que el modelo resulte manejable, en algunas situaciones nos deberemos conformar con simplificaciones en las que perdamos exactitud. Ello comportará por un lado que el modelo no sólo recogerá las posibles secuencias óptimas, sino también a otras espúreas. Por otra parte, las secuencias óptimas reales podrían resultar más largas que las que el modelo nos ofrece. De esta forma se garantizará que, al realizar una simplificación, sea de forma que la heurística siga siendo *admisible*.

Se van a construir varios modelos simplificados en los que aparecerán las restricciones fundamentales que sirven para definir las posibles secuencias de herramientas para cada máquina según el árbol de precedencia de tareas, obviándose aquellas que requerirían un tratamiento más complejo. En el primer modelo se consideran restricciones sencillas, y los siguientes se irán complicando con nuevas restricciones que permiten una estimación más exacta, aunque a la vez más costosa de evaluar computacionalmente. Los sucesivos modelos servirán para definir distintas funciones heurísticas. Los resultados nos indicarán la importancia de las restricciones que se están despreciando, y si los cálculos de las diferentes heurísticas son meritorios respecto al compromiso entre una mejor aproximación (y por tanto a un menor número de nodos expandidos) y un tiempo de cómputo adecuado.

Como se indicó antes, se trata de calcular una estimación optimista del número de cambios de herramientas en un árbol de precedencias. Este cálculo se realiza para cada máquina, por lo que las tareas relevantes del árbol de precedencias serán aquellas que usen la máquina en cuestión. Las demás tareas no proporcionarán información directa, aunque el hueco que dejan en el árbol será de gran importancia para el análisis de las secuencias de herramientas.

En un análisis recursivo de las secuencias de herramientas posibles, el aspecto más inmediato a estudiar es el cambio de herramienta que puede darse debido a la precedencia de la tarea situada en la raíz respecto a sus descendientes. El otro aspecto que habrá que analizar será el efecto debido a la consideración de las herramientas necesarias provenientes de distintas ramas del árbol.

A continuación se van a ir formalizando los distintos componentes del modelo, comenzando por la relación entre un árbol de precedencia y las secuencias de herramientas que van a ser representadas de una forma indirecta por el mismo.

Dado un árbol de precedencia de tareas, las tareas que usen una misma máquina deben ejecutarse de forma secuencial. Para cada secuencia de tareas que usen una misma máquina, se tendrá una secuencia de herramientas con las herramientas de las tareas como elementos de la misma y en el mismo orden que las tareas que las utilizan. Como estamos interesados únicamente en los cambios de herramientas, podemos simplificar la secuencia de herramientas anterior sustituyendo las herramientas consecutivas iguales por una sola. A las secuencias así formadas son a las que nos referiremos al hablar de *secuencias de herramientas*.

Como se ha indicado, no existe una sola secuencia de tareas asociada a un árbol de precedencias, y lo que se está analizando es una estimación del número de cambios de herramientas en el caso más favorable, es decir, para alguna o algunas de las secuencias del árbol. De esta forma, necesitamos representar a esas secuencias según un modelo, que en nuestro caso, será una simplificación del conjunto de restricciones involucradas en el árbol de precedencias.

Los distintos modelos recogerán el número de apariciones de cada herramienta, así como cuál de las herramientas es la primera que se usará. Para esto último, tomará la usada por la tarea situada más arriba en el árbol de precedencia, aquella cuyas predecesoras en el árbol no usen la misma máquina. Para el resto de las herramientas, no se tendrá en cuenta ninguna restricción de precedencia, simplificando de esta manera el modelo. El valor que nos interesará será el número de cambios de herramientas, que coincidirá con el número total de apariciones de todas las herramientas menos uno.

Para la definición de los modelos nos basaremos en una serie de observaciones importantes, que iremos enumerando.

Observación 4.1:

Sea un árbol de precedencias cuya raíz es la tarea T, y sean M(T) y H(T) la máquina y la herramienta usadas por T respectivamente. Para determinar la primera herramienta que se necesitará para una máquina genérica M se tendrán dos casos:

- Si M=M(T), entonces la primera herramienta será la usada por T, H(T).
- Si M≠M(T), entonces puede que no esté determinada, ya que podría ser la primera herramienta asociada a cualquiera de los dos posibles subárboles.

Ejemplo 4.1:

La tarea situada en la raíz del árbol de la Figura 4.10 usa la máquina M_1. La primera herramienta para la máquina M_2 puede ser H_2 o H_3, las primeras herramientas de los subárboles debajo de la raíz.

De esta observación extraemos que debemos definir un conjunto de primeras herramientas posibles. En ese conjunto podrá haber más de dos herramientas, como puede verse en el siguiente ejemplo. Las letras en éste y en sucesivos ejemplos representan a las distintas herramientas asociadas a una misma máquina.

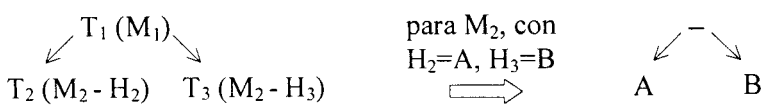


Figura 4.10: Primera herramienta para una máquina no usada en la raíz.

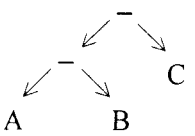


Figura 4.11: Conjunto de primeras herramientas.

Ejemplo 4.2:

En el árbol de la Figura 4.11, cualquiera de las tres herramientas puede ser usada en primer lugar en la secuencia.

Observación 4.2:

En la combinación o *mezcla* de secuencias de distintos subárboles, se intentará que las tareas que usen una misma herramienta aparezcan consecutivas. Con ello se evitarán cambios de herramientas y se consigue que las secuencias de herramientas sean más cortas.

Será necesario analizar las secuencias de herramientas en cada rama para determinar el caso más favorable en el número de cambios de herramientas. Hay que observar, sin embargo, que no hay una sola secuencia posible para cada rama, por lo que el análisis se hará en función de los casos más favorables.

Ejemplo 4.3:

En la Figura 4.12, se muestran dos árboles de herramientas que deben combinarse (a). Para evitar cambios de herramientas, se aprovecha que en el primero de los árboles aparecen las herramientas del segundo. La restricción de precedencia del segundo árbol aparece como flecha discontinua en (b). Como consecuencia de todas las restricciones de precedencia, se obtiene la (única) secuencia óptima A-B-C, representada en el árbol (c). En general, resultaría muy complejo considerar todas las restricciones de precedencia para obtener el óptimo, y el modelo no sería operativo. Como simplificación, el modelo sólo contemplará que las herramientas A, B y C se usan una vez, siendo A la primera. De esta forma, es fácil ver que el modelo incluirá secuencias que en realidad no pueden darse. En el caso del ejemplo de la Figura 4.12 (d), la secuencia A-C-B es tenida en cuenta, aunque en realidad no puede producirse.

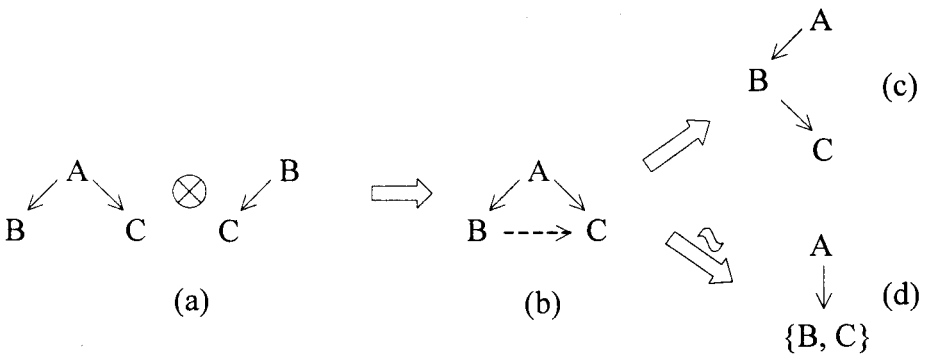


Figura 4.12: Ejemplo de combinación de secuencias de dos árboles.

En el ejemplo anterior, y en los que siguen, se usa el símbolo ⊗ para denotar la combinación de secuencias de herramientas, de manera que, según lo indicado antes, el resultado que se persigue debe incluir a todas las secuencias posibles con el menor número de elementos. Por otro lado, para referirnos a varias secuencias o grupos de secuencias alternativas posibles, se usará el símbolo ⊕. El modelo a definir deberá englobar a todas esas secuencias.

MODELO 1

Estamos ya en disposición de dar una primera especificación para el modelo inicial, que servirá de base para los posteriores. En él definiremos, para cada máquina, un *multiconjunto*[4] $\mathcal{B}$ que contiene los mismos elementos (y con la misma multiplicidad) que la(s) secuencia(s) de herramientas óptima(s) del árbol de precedencia. Además, se definirá un *conjunto* $\mathcal{F}$ conteniendo aquellas herramientas que pueden ser utilizadas en primer lugar.

La estructura resultante es un multiconjunto parcialmente ordenado o *pomset*[5], término acuñado por Pratt y usado en la especificación de procesos concurrentes [Pratt, 1986]. Se llama *alineamiento* de un pomset a una secuencia formada por los elementos del multiconjunto que cumpla con las restricciones de ordenación definidas entre sus elementos. Esas restricciones son, por un lado, las asociadas al conjunto de primeros $\mathcal{F}$, y por otro, el que los elementos contiguos en los alineamientos del pomset deben ser distintos. Un alineamiento se corresponde, por tanto, con el pomset resultante de añadir las restricciones necesarias para definir un *orden total* para formar la secuencia. El conjunto de alineamientos posibles de un pomset de nuestro modelo nos daría lugar al conjunto de posibles secuencias que podrían construirse a partir de él. Para el caso que nos ocupa, tales secuencias serían, en un caso ideal, las secuencias de herramientas óptimas.

[4] Un multiconjunto es una colección no ordenada de valores que puede contener duplicados.
[5] El término *pomset* proviene de *partially ordered multiset*. Por comodidad usaremos *pomset* al no existir el equivalente en español a la abreviatura de "multiconjunto parcialmente ordenado"

Al referirnos al pomset de nuestro modelo usaremos según convenga el término $\mathcal{P}$ o la tupla equivalente $<\mathcal{B},\mathcal{F}>$. Será necesario definir dos operaciones básicas, *push* y *merge*, para la construcción del pomset asociado a un árbol de precedencias. La operación *push* se usará para añadir la herramienta que debe usarse en primer lugar por estar en la raíz del árbol. La operación *merge* será utilizada al mezclar los pomsets correspondientes a distintos subárboles. De esta forma, el pomset asociado a la tarea T y la máquina M viene dado por

$$\mathcal{P}(T,M)\ \ (M=M(T)) \equiv \begin{cases} <\{H(T)\},\{H(T)\}> & \text{si } suc(T)=\varnothing \\ push\big(H(T),\mathcal{P}(T_1,M)\big) & \text{si } suc(T)=\{T_1\} \\ push\big(H(T),merge\big(\mathcal{P}(T_1,M),\mathcal{P}(T_2,M)\big)\big) & \text{si } suc(T)=\{T_1,T_2\} \end{cases} \tag{4.24}$$

si la máquina M es la usada por T. En la ecuación (4.24), $suc(T)$ representa al conjunto de tareas que suceden a T como hijas en el árbol de precedencias. Si la máquina M no es la usada por T, entonces el pomset viene dado por las expresiones:

$$\mathcal{P}(T,M)\ \ (M\neq M(T)) \equiv \begin{cases} \varnothing & \text{si } suc(T)=\varnothing \\ \mathcal{P}(T_1,M) & \text{si } suc(T)=\{T_1\} \\ merge\big(\mathcal{P}(T_1,M),\mathcal{P}(T_2,M)\big) & \text{si } suc(T)=\{T_1,T_2\} \end{cases} \tag{4.25}$$

Para completar la descripción del modelo, sólo necesitamos definir la operación que buscábamos desde el principio, la que nos dará el número mínimo de cambios de herramientas asociado al árbol de precedencias. Llamamos *ntrans*($\mathcal{P}$) al número de transiciones en cualquiera de los alineamientos posibles del pomset $\mathcal{P}$. Refiriéndonos a las secuencias de herramientas, que en nuestro modelo se corresponden con los posibles alineamientos del pomset obtenido, *ntrans*($\mathcal{P}$) coincidirá con el número mínimo de cambios de herramientas asociado al árbol de precedencias.

En cuanto a la utilización de los resultados ofrecidos por los modelos sobre el algoritmo A*, se procederá como sigue:

> Sean T_1, T_2, ..., T_m las tareas candidatas para un nodo n del árbol de expansión, y $\mathcal{P}_1$, $\mathcal{P}_2$, ..., $\mathcal{P}_m$ sus respectivos pomsets, correspondientes a la máquina M, es decir, $\mathcal{P}_i = \mathcal{P}(T_i, M)$ (i = 1, 2, ..., m). Sea H la última herramienta usada en el estado correspondiente al nodo n. El número de cambios de herramientas que como mínimo se necesitarán en la máquina M es
>
> $$ntrans\big(push\big(H,merge(\mathcal{P}_1,\mathcal{P}_2,...,\mathcal{P}_m)\big)\big) \tag{4.26}$$

Antes de proceder a la definición de las operaciones *push* y *merge* es necesario aclarar las operaciones que se usarán sobre los multiconjuntos, dado que, a diferencia de los conjuntos, no existe una notación estándar. En la Tabla 4.1 se indican las operaciones

Por otro lado, para representar un multiconjunto a partir de sus elementos suele usarse una notación similar a la de los conjuntos, de manera que cuando hay varios ejemplares de un mismo elemento, éste aparece repetido. Así, por ejemplo, un multiconjunto formado por tres ejemplares del elemento x, un ejemplar del elemento y y dos ejemplares del elemento z se representaría como $\{x, x, x, y, z, z\}$. Cuando en el multiconjunto hay varios ejemplares de los elementos, es muy útil otra notación alternativa, muy usada también, en la que se indica, para cada elemento, el número de ejemplares presentes en el multiconjunto. El multiconjunto del ejemplo se representaría como $\{3{\cdot}x, 1{\cdot}y, 2{\cdot}z\}$, de manera que el número de repeticiones de un elemento aparece como un "multiplicador".

Para los pomsets que nos ocupa, llamaremos *α-pomset* a un pomset en donde el primer elemento de todos los posibles alineamientos es único. Esto equivale a decir que un pomset $\mathcal{P} \equiv < \mathcal{B}, \mathcal{F} >$ es un α-pomset si $\mathcal{F}$ sólo tiene un elemento. Un α-pomset vendrá representado de forma alternativa a la notación para los pomsets, por su primer elemento diferenciado del multiconjunto de elementos restantes, separados por una flecha. Así, por ejemplo, el α-pomset $<\{3{\cdot}x, 1{\cdot}y, 2{\cdot}z\}, \{x\}>$ también se representará como $x \rightarrow \{2{\cdot}x, 1{\cdot}y, 2{\cdot}z\}$.

Usaremos el operador $\oplus$ para descomponer un pomset en todos sus α-pomsets. Así, si $\mathcal{P} \equiv < \mathcal{B}, \mathcal{F} >$, con $\mathcal{F} = \{x_1, x_2, ..., x_n\}$, se tendrá:

$$\begin{aligned} \mathcal{P} &\equiv < \mathcal{B}, \{x_1, x_2, ..., x_n\} > \\ &\equiv < \mathcal{B}, \{x_1\} > \oplus < \mathcal{B}, \{x_2\} > \oplus \ \ldots \ \oplus < \mathcal{B}, \{x_n\} > \\ &\equiv x_1 \rightarrow (\mathcal{B} - \{x_1\}) \oplus x_2 \rightarrow (\mathcal{B} - \{x_2\}) \oplus \ \ldots \ \oplus x_n \rightarrow (\mathcal{B} - \{x_n\}) \end{aligned} \tag{4.27}$$

La operación *push* viene definida por la fórmula

$$push(x, < \mathcal{B}, \mathcal{F} >) \equiv \begin{cases} < \mathcal{B}, \{x\} > & \text{si } x \in \mathcal{F} \\ < \mathcal{B} \uplus \{x\}, \{x\} > & \text{si } x \notin \mathcal{F} \end{cases} \tag{4.28}$$

o, teniendo en cuenta el pomset resultante es un α-pomset,

$$push(x, < \mathcal{B}, \mathcal{F} >) \equiv \begin{cases} x \rightarrow (\mathcal{B} - \{x\}) & \text{si } x \in \mathcal{F} \\ x \rightarrow \mathcal{B} & \text{si } x \notin \mathcal{F} \end{cases} \tag{4.29}$$

Tabla 4.1: Operaciones con multiconjuntos

Multiset ( ) – $\varnothing$ multiconjunto vacío
singleton (x) – $\{x\}$ multiconjunto con x como único elemento
add (x, A) – $A \uplus \{x\}$ añade un ejemplar del elemento x al multiconjunto A
contains (A, x) – $x \in A$ el elemento x está contenido en el multiconjunto A
containsAll (A, B) – $B \subseteq A$ todos los elementos del multiconjunto B están incluidos en el multiconjunto A
remove (x, A) – $A - \{x\}$ elimina un ejemplar del elemento x del multiconjunto A, si está presente
removeAll (A, C) elimina del multiconjunto A todos los ejemplares de los elementos incluidos en el conjunto C
retainAll (A, C) mantiene en el multiconjunto A únicamente los elementos contenidos en el conjunto C
card (A) – $\# A$ número total de elementos, distintos o no, incluidos en el multiconjunto A
cardDiff (A) – $\#_d A$ número de elementos distintos en el multiconjunto A
card (x, A) – $\# (x, A)$ número de ejemplares del elemento x en el multiconjunto A
union (A, B) – $A \cup B$ unión de los multiconjuntos A y B de acuerdo con el valor máximo de la cardinalidad de cada elemento en A y B. El mayor valor determinará el número de ejemplares de cada elemento en el multiconjunto resultado. Esto es, para cada elemento x tal que $x \in A$ ó $x \in B$, se cumplirá: $$\#(x, A \cup B) = \max\left(\#(x, A), \#(x, B)\right)$$

Tabla 4.1: Operaciones con multiconjuntos (continuación)

additiveUnion (A, B) – $A \uplus B$ unión aditiva de los multiconjuntos A y B de acuerdo con la suma de las cardinalidades de cada elemento en A y B. Esta suma determinará el número de ejemplares de cada elemento en el multiconjunto resultado. Esto es, para cada elemento x tal que $x \in A$ ó $x \in B$, se cumplirá: $\#(x, A \uplus B) = \#(x, A) + \#(x, B)$
intersection (A, B) – $A \cap B$ intersección de los multiconjuntos A y B de acuerdo con el valor mínimo de la cardinalidad de cada elemento en A y B. El menor valor determinará el número de ejemplares de cada elemento en el multiconjunto resultado. Es decir, para cada elemento x tal que $x \in A$ ó $x \in B$, se cumplirá: $\#(x, A \cap B) = \min(\#(x, A), \#(x, B))$
diff (A, B) – $A - B$ diferencia de los multiconjuntos A y B de acuerdo con el valor máximo entre cero y la diferencia de las cardinalidades de cada elemento en A y B. El mayor valor determinará el número de ejemplares de cada elemento en el multiconjunto resultado. Es decir, para cada elemento x tal que $x \in A$ ó $x \in B$, se cumplirá: $\#(x, A - B) = \max(0, \#(x, A) - \#(x, B))$
flatten (A) – $\widehat{A}$ aplanamiento del multiconjunto A, cuyos elementos son multiconjuntos, de acuerdo a la unión aditiva de los multiconjuntos de A. De esta forma, si $A = \{A_1, A_2, ..., A_n\}$, entonces $\widehat{A} = \biguplus_{i=1..n} A_i$.

El análisis de la operación *push* es muy simple. Como se recoge en las ecuaciones (4.28) y (4.29), en cualquier caso, el conjunto $\mathcal{F}$ del pomset resultante deberá estar compuesto únicamente por el nuevo elemento a considerar, la herramienta correspondiente a la raíz. En el caso de que el elemento a añadir no fuera uno de los que podían aparecer como primero en algunas de las posibles secuencias, será necesario añadir un ejemplar suyo al multiconjunto, de forma que refleje al cambio de herramienta adicional que deberá producirse necesariamente. En el caso de que el elemento a añadir fuera uno de los que podían aparecer como primero en las posibles secuencias, ese cambio de herramienta adicional podría ahorrarse, por lo que el multiconjunto no debe cambiar.

Nótese que, cuando la operación *push* modifica el pomset, desaparecen las restricciones de precedencia asociadas a los elementos que antes estaban en el conjunto de

primeros con respecto a los demás elementos, mientras que aparecen otras nuevas correspondientes al nuevo elemento primero.

La operación *push* así definida obviamente garantiza que el pomset obtenido incluye a las secuencias óptimas.

La definición de la operación *merge* viene dada por:

$$merge(\mathcal{P}_1,\mathcal{P}_2) \equiv \begin{cases} <\mathcal{B}_1 \cup \mathcal{B}_2, \mathcal{D}_{\mathcal{F}}(\mathcal{P}_1,\mathcal{P}_2)> & \text{si } \mathcal{D}_{\mathcal{F}}(\mathcal{P}_1,\mathcal{P}_2) \neq \varnothing \\ <\mathcal{B}_1 \cup \mathcal{B}_2, \mathcal{F}_1 \cup \mathcal{F}_2> & \text{si } \mathcal{D}_{\mathcal{F}}(\mathcal{P}_1,\mathcal{P}_2) = \varnothing \end{cases} \tag{4.30}$$

donde $\mathcal{P}_1 \equiv <\mathcal{B}_1, \mathcal{F}_1>$ y $\mathcal{P}_2 \equiv <\mathcal{B}_2, \mathcal{F}_2>$, y $\mathcal{D}_{\mathcal{F}}(\mathcal{P}_1, \mathcal{P}_2)$ es el conjunto de elementos de $\mathcal{P}_1$ y $\mathcal{P}_2$ definido por

$$\mathcal{D}_{\mathcal{F}}(\mathcal{P}_1,\mathcal{P}_2) \equiv (\mathcal{F}_1 \cap \mathcal{F}_2) \cup \mathcal{D}'_{\mathcal{F}}(\mathcal{P}_1,\mathcal{P}_2) \cup \mathcal{D}'_{\mathcal{F}}(\mathcal{P}_2,\mathcal{P}_1) \tag{4.31}$$

siendo

$$\mathcal{D}'_{\mathcal{F}}(\mathcal{P}_i,\mathcal{P}_j) \equiv \{x \in \mathcal{F}_i \mid \#(x,\mathcal{B}_i) > \#(x,\mathcal{B}_j)\} \tag{4.32}$$

Como se observa, el análisis de la operación *merge* es más complejo. Atendiendo al número de ejemplares de cada elemento, está claro que como mínimo resultará ser el valor máximo que tenía en los dos pomsets originales, ya que cada aparición en el pomset donde hay menos elementos (o igual número) podría asociarse a otra aparición en el otro pomset, evitando así nuevos cambios de herramienta. Esto se corresponde precisamente con la unión de los dos multiconjuntos. Nótese que en éste análisis se está considerando que los elementos pueden su incluye el número de apariciones que debe tener cada elemento en una mezcla óptima, a falta de considerar las restricciones asociadas a los conjuntos de primeros $\mathcal{F}$.

Por otro lado, queda por resolver cuáles son los elementos que deben aparecer en el conjunto de primeros del pomset resultante de la mezcla. Resulta evidente que si un elemento aparece en el conjunto de primeros en ambos pomsets de entrada, también debe hacerlo en el resultante. Para los elementos que sólo aparecen en el conjunto $\mathcal{F}$ de uno sólo de los pomsets, hay que considerar el número de apariciones en los dos multiconjuntos.

Para ello vamos a definir el concepto de *dominancia*. Un elemento que aparece en un conjunto de primeros en uno de los pomsets $\mathcal{P}_i$ es un elemento *dominante* en $\mathcal{P}_i$ respecto al otro pomset $\mathcal{P}_j$ si el número de apariciones en $\mathcal{P}_i$ es mayor que el que tiene en $\mathcal{P}_j$. Así, los elementos dominantes de cada uno de los pomsets deben aparecer en el conjunto de primeros del pomset resultante, ya que cada aparición del elemento en el pomset en donde no es dominante puede emparejarse con cada una de las apariciones, excepto la asociada a la primera, en el pomset en donde es dominante, sin alterar el número total de apariciones que debe haber en la mezcla, según se indicó antes. Por contra, un elemento no dominante no debe aparecer en el conjunto de primeros de la

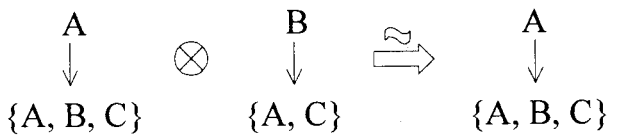


Figura 4.14: Ejemplo de elementos dominantes.

mezcla, ya que entonces el número de apariciones sería superior en uno al que le corresponde según la unión de multiconjuntos. En este caso, la aparición correspondiente a la primera posición en la secuencia no puede emparejarse con ninguna aparición del otro pomset, por lo que se añadiría un ejemplar al número que hubiera en ese multiconjunto.

Ejemplo 4.4:

En la Figura 4.14, el elemento A es dominante en el primer pomset, mientras que B no lo es en el segundo. Se obtiene un pomset con un número mínimo de elementos, cuatro, en el que sólo aparece en el conjunto de primeros el elemento A.

De esta forma, el conjunto $\mathcal{D}'_{\mathcal{F}}(\mathcal{P}_i,\mathcal{P}_j)$ contiene a los elementos de $\mathcal{F}_i$ que son *dominantes* respecto a $\mathcal{P}_j$. Se justifica así que el conjunto de primeros del pomset resultante de la mezcla venga dado por el conjunto $\mathcal{D}_{\mathcal{F}}(\mathcal{P}_1,\mathcal{P}_2)$ definido en (4.31).

Observación 4.3:

El proceso a seguir para realizar el análisis de *merge* es el siguiente:

1. Descomponer en sus respectivos α-pomsets los dos pomsets de entrada $\mathcal{P}_1$ y $\mathcal{P}_2$, $\mathcal{P}_1 = \mathcal{P}_{11} \oplus \mathcal{P}_{12} \oplus \ldots \oplus \mathcal{P}_{1n}$, y $\mathcal{P}_2 = \mathcal{P}_{21} \oplus \mathcal{P}_{22} \oplus \ldots \oplus \mathcal{P}_{2m}$
2. Obtener la mezcla de cada par de α-pomsets $\mathcal{P}_{1i}$, $\mathcal{P}_{2j}$.
3. De los α-pomsets obtenidos, descartar aquellos cuyo número de elementos supere el mínimo de entre ellos.
4. Los α-pomsets que quedan del paso anterior son los que forman el pomset resultante.

La mezcla de dos α-pomsets se realiza como sigue:

1. Sean $\mathcal{P}_1 = x_1 \rightarrow (\mathcal{B}_1 - \{x_1\})$ y $\mathcal{P}_2 = x_2 \rightarrow (\mathcal{B}_2 - \{x_2\})$
2. Si $x_1 = x_2 \Rightarrow \mathcal{P}_1 \otimes \mathcal{P}_2 = x_1 \rightarrow \big((\mathcal{B}_1 - \{x_1\}) \cup (\mathcal{B}_2 - \{x_2\})\big) = < \mathcal{B}_1 \cup \mathcal{B}_2, \{x_1\} >$
3. Si $x_1 \neq x_2 \Rightarrow \mathcal{P}_1 \otimes \mathcal{P}_2 = x_1 \rightarrow \big((\mathcal{B}_1 - \{x_1\}) \cup \mathcal{B}_2\big) \oplus x_2 \rightarrow \big((\mathcal{B}_2 - \{x_2\}) \cup \mathcal{B}_1\big)$
4. Si el número de elementos de los α-pomsets obtenidos en el paso anterior es diferente, eliminar el correspondiente al mayor número.

Veamos en qué condiciones se tiene la circunstancia expresada en el paso 4:

$$\#\left(x_1 \to \left(\left(\mathcal{B}_1 - \{x_1\}\right) \cup \mathcal{B}_2\right)\right) = \sum_{x \neq x_1} \#\left(x, \mathcal{B}_1 \cup \mathcal{B}_2\right) + 1 + \max\left(\#\left(x_1, \mathcal{B}_1 - \{x_1\}\right), \#\left(x_1, \mathcal{B}_2\right)\right)$$

Ahora bien,

$$\max\left(\#\left(x_1, \mathcal{B}_1 - \{x_1\}\right), \#\left(x_1, \mathcal{B}_2\right)\right) = \begin{cases} \#\left(x_1, \mathcal{B}_1 - \{x_1\}\right) & \text{si } \#\left(x_1, \mathcal{B}_1 - \{x_1\}\right) \geq \#\left(x_1, \mathcal{B}_2\right) \\ \#\left(x_1, \mathcal{B}_2\right) & \text{si } \#\left(x_1, \mathcal{B}_1 - \{x_1\}\right) < \#\left(x_1, \mathcal{B}_2\right) \end{cases}$$

y $\#\left(x_1, \mathcal{B}_1 - \{x_1\}\right) = \#\left(x_1, \mathcal{B}_1\right) - 1$, luego

$$\#\left(x_1 \to \left(\left(\mathcal{B}_1 - \{x_1\}\right) \cup \mathcal{B}_2\right)\right) = \begin{cases} \sum_{x \in \mathcal{B}_1 \cup \mathcal{B}_2} \#\left(x, \mathcal{B}_1 \cup \mathcal{B}_2\right) & \text{si } \#\left(x_1, \mathcal{B}_1\right) > \#\left(x_1, \mathcal{B}_2\right) \\ \sum_{x \in \mathcal{B}_1 \cup \mathcal{B}_2} \#\left(x, \mathcal{B}_1 \cup \mathcal{B}_2\right) + 1 & \text{si } \#\left(x_1, \mathcal{B}_1\right) \leq \#\left(x_1, \mathcal{B}_2\right) \end{cases}$$

teniendo en cuenta que $\#\left(x_1, \mathcal{B}_1\right) = 1 + \#\left(x_1, \mathcal{B}_1 - \{x_1\}\right)$. Un resultado simétrico se tendría para el otro α-pomset.

Luego en el mejor de los casos se obtendría una secuencia mínima con un número de elementos igual al correspondiente a la unión de los multiconjuntos si alguno de los elementos que aparecía como primero es dominante.

En el caso de que $\mathcal{D}_{\mathcal{F}}(\mathcal{P}_1, \mathcal{P}_2)$ esté vacío sucede que, como $\mathcal{F}$ no puede quedar vacío, es necesario escoger sus elementos de entre los elementos de $\mathcal{F}_1$ y $\mathcal{F}_2$, pero como ningún elemento es dominante, todos están en las mismas condiciones para aparecer como primeros en una eventual secuencia óptima.

Ejemplo 4.5:

En la Figura 4.15, ni A ni B son dominantes. Las secuencias óptimas pueden contener como primero a cualquiera de ellos. La aplicación de las ecuaciones (4.30) a (4.32) al ejemplo nos da como resultado:

$$\mathcal{D}'_{\mathcal{F}}\left(\mathcal{P}_1, \mathcal{P}_2\right) = \left\{x \in \{\text{A}\} \mid \#\left(x, \{\text{A,B,C}\}\right) > \#\left(x, \{\text{A,B,C}\}\right)\right\} = \varnothing$$

$$\mathcal{D}'_{\mathcal{F}}\left(\mathcal{P}_2, \mathcal{P}_1\right) = \left\{x \in \{\text{B}\} \mid \#\left(x, \{\text{A,B,C}\}\right) > \#\left(x, \{\text{A,B,C}\}\right)\right\} = \varnothing$$

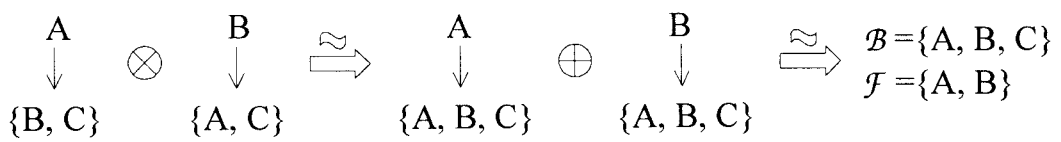


Figura 4.15: Ejemplo de elementos no dominantes.

$$\mathcal{D}_{\mathcal{F}}(\mathcal{P}_1,\mathcal{P}_2)=(\{A\}\cap\{B\})\cup\varnothing\cup\varnothing=\varnothing$$

$$merge(<\{A,B,C\},\{A\}>,<\{A,B,C\},\{B\}>)\equiv<\{A,B,C\},\{A,B\}>$$

Como se observará más adelante, el modelo propuesto en la ecuación (4.30) realiza una simplificación adicional para el caso $\mathcal{D}_{\mathcal{F}}(\mathcal{P}_1, \mathcal{P}_2) = \varnothing$, en cuanto a que el número de elementos en el multiconjunto es inferior al número de elementos secuencias que en realidad tendrá cualquier secuencia óptima.

El número de transiciones del pomset viene dado por el número de apariciones de todas los elementos en el multiconjunto menos 1:

$$ntrans(\mathcal{P})=\sum_{x\in\mathcal{B}}\#(x,\mathcal{B})-1 \tag{4.33}$$

De la definición del modelo, se concluye que la heurística obtenida es *admisible*, ya que, según se ha visto, cualquier secuencia posible de un árbol no puede ser más pequeña que ningún alineamiento del pomset generado. Además, en el caso de que la secuencia sea de igual longitud, y por tanto será óptima, coincidirá con uno de los alineamientos del pomset.

Las siguientes observaciones permitirán refinar aún más la heurística, motivando en su caso la modificación del modelo.

Observación 4.4:

> En los casos en que $\mathcal{D}_{\mathcal{F}}(\mathcal{P}_1, \mathcal{P}_2) = \varnothing$, la operación *merge* definida por la ecuación (4.30) ofrece una valoración por debajo del óptimo. Así pues, aparte de la aparición de soluciones espúreas, el modelo proporciona una imprecisión en el sentido indicado. Un modelo más exacto permitiría estimar con más exactitud el número de cambios de herramientas.

El Ejemplo 4.5 es un exponente de tal situación. En la Figura 4.16 se ofrece un análisis más detallado de las restricciones que se añaden realmente al mezclar. En el ejemplo, dependiendo de cuál de los árboles se elija para tomar la primera tarea, la primera

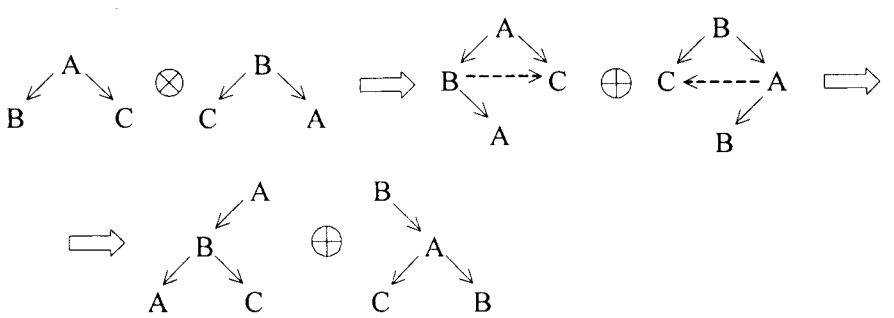


Figura 4.16: Ejemplo de elementos no dominantes.

herramienta sería distinta. Además, el número de apariciones de la primera herramienta está por determinar. Si la primera herramienta fuera A, entonces aparecería dos veces, mientras que si fuera B, entonces sería esta la que aparecería dos veces, por una la herramienta A. En cualquiera de las dos opciones, el número de cambios de herramientas sería 3, mientras que el modelo propuesto antes nos daría un total de dos cambios de herramientas, según puede verse en la Figura 4.15.

La siguiente proposición formaliza la observación anterior.

Proposición 4.1:

Sea $\mathcal{P} = merge(<\mathcal{B}_1,\mathcal{F}_1>,<\mathcal{B}_2,\mathcal{F}_2>)$. Si $\mathcal{D}_{\mathcal{F}}(\mathcal{P}_1,\mathcal{P}_2)=\varnothing$, uno sólo de los elementos x de $\mathcal{F}_1 \cup \mathcal{F}_2$ aparecerá $\#(x,\mathcal{B}_1 \cup \mathcal{B}_2)+1$ veces. Tal elemento puede ser cualquiera de los elementos de $\mathcal{F}_1 \cup \mathcal{F}_2$.

Demostración: Sea un elemento cualquiera $x \in \mathcal{F}_1$. Como $\mathcal{D}_{\mathcal{F}}(\mathcal{P}_1,\mathcal{P}_2)=\varnothing$, entonces $x \notin \mathcal{F}_2$ y $\#(x,\mathcal{B}_1) \leq \#(x,\mathcal{B}_2)$. Esto quiere decir que si queremos mantener en $\mathcal{P} \equiv <\mathcal{B},\mathcal{F}>$ el número de apariciones de x como $\#(x,\mathcal{B}_2)$ será a costa de que x no esté en $\mathcal{F}$. Como a todos los elementos de $\mathcal{F}_1$ y $\mathcal{F}_2$ les ocurre lo mismo, $\mathcal{F}$ quedaría vacío, cosa que no puede suceder. Esto implica que cualquier elemento de $\mathcal{F}_1$ y $\mathcal{F}_2$ puede ser candidato a ser primero en una secuencia mínima, por lo que $\mathcal{F} = \mathcal{F}_1 \cup \mathcal{F}_2$. Si un elemento $x \in \mathcal{F}$ no fuera el primero de la secuencia, el número de apariciones en $\mathcal{B}$ sería $\max(\#(x,\mathcal{B}_1),\#(x,\mathcal{B}_2))$, pero si fuera el escogido como primero de la secuencia, su número de apariciones sería $\max(\#(x,\mathcal{B}_1),\#(x,\mathcal{B}_2))+1$.

Corolario:

Sean $\mathcal{P}_1 \equiv <\mathcal{B}_1,\mathcal{F}_1>$ y $\mathcal{P}_1 \equiv <\mathcal{B}_2,\mathcal{F}_2>$. Si $\mathcal{D}_{\mathcal{F}}(\mathcal{P}_1,\mathcal{P}_2)=\varnothing$, el número real de transiciones en los elementos de $\mathcal{P} = merge(<\mathcal{B}_1,\mathcal{F}_1>,<\mathcal{B}_2,\mathcal{F}_2>)$ es $\sum_{x \in \mathcal{B}_1 \cup \mathcal{B}_2} \#(x,\mathcal{B}_1 \cup \mathcal{B}_2)$.

Demostración: Inmediata a partir de la proposición anterior, de la definición de unión de multiconjuntos y de la ecuación (4.33).

MODELO 2

Para usar los resultados anteriores es necesario revisar el modelo anterior. Llamaremos *u-pomset* a la estructura resultante, en el sentido de que habrá elementos para los que el número de ejemplares en el multiconjunto no está completamente determinado. Las observaciones siguientes ayudarán a perfilar la nueva estructura, así como las nuevas definiciones de las operaciones *push*, *merge* y *ntrans*.

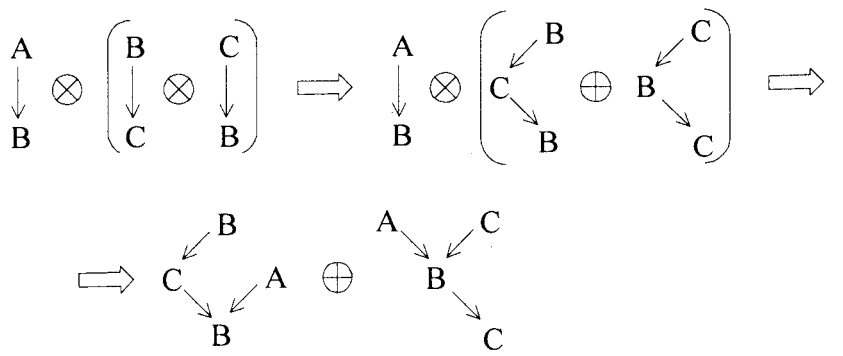


Figura 4.17: El número de ejemplares de un elemento puede o no estar determinado.

Observación 4.5:

En el conjunto de primeros puede haber elementos cuyo número de apariciones esté determinado y otros que no.

Ejemplo 4.6:

En *merge*(<{A,B},{A}>, *merge*(<{B,C},{B}>,<{B,C},{C}>)), el conjunto $\mathcal{F}$ está formado por los elementos A, B y C. El número de ejemplares de A está perfectamente definido, 1, mientras que los de B y C no, pueden ser 1 ó 2. En la Figura 4.17 se muestran las restricciones reales que aparecen al mezclar los subárboles.

Observación 4.6:

Podemos tener más de dos elementos en un conjunto de elementos con un número indeterminado de apariciones, lo cual significa que uno sólo de ellos tendrá un ejemplar más.

Ejemplo 4.7:

En *merge*(<{A,B,A,B},{A}>, *merge*(<{B,C},{B}>,<{C,A,B,A},{C}>)), el conjunto $\mathcal{F}$ está formado por los elementos A, B y C. El número mínimo de ejemplares de ellos es de 2, 2 y 1 respectivamente. Pero uno de ellos tendrá un elemento más de lo indicado, según se muestra en la Figura 4.18. En ella se muestran las únicas secuencias posibles. Los pomsets generados por los modelos incluirán más secuencias, dado que sólo la restricción de precedencia correspondiente al primer elemento con respecto a los demás se mantiene, junto con la restricción de que dos elementos contiguos deben ser diferentes.

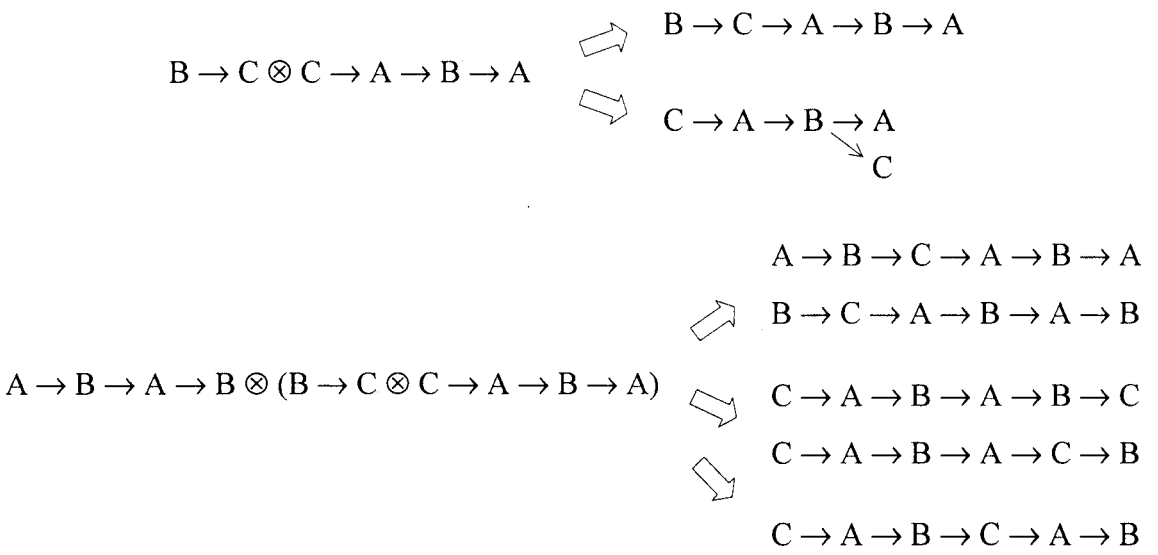


Figura 4.18: Ejemplo de indeterminación con tres elementos.

Observación 4.7:

Puede haber varias indeterminaciones independientes, es decir, varios conjuntos de elementos, disjuntos entre sí, que involucren la aparición de uno de sus elementos una vez más.

Ejemplo 4.8:

$(A \to B \otimes B \to A) \otimes (C \to D \otimes D \to C) \Rightarrow$

$(A \to B \to A \oplus B \to A \to B) \otimes (C \to D \to C \oplus D \to C \to D) \Rightarrow$

$$\begin{array}{llllllll} A\to B\to A & \oplus & A\to B\to A & \oplus & B\to A\to B & \oplus & B\to A\to B & \oplus \\ \hookrightarrow C\to D\to C & & \hookrightarrow D\to C\to D & & \hookrightarrow C\to D\to C & & \hookrightarrow D\to C\to D & \\ C\to D\to C & \oplus & C\to D\to C & \oplus & D\to C\to D & \oplus & D\to C\to D & \\ \hookrightarrow A\to B\to A & & \hookrightarrow B\to A\to B & & \hookrightarrow A\to B\to A & & \hookrightarrow B\to A\to B & \end{array}$$

Observación 4.8:

Las indeterminaciones pueden no estar ligadas a que los elementos estén en el conjunto de primeros, ya que al aplicarle la operación *push*, la indeterminación puede no desaparecer, aunque el número de ejemplares de los elementos del conjunto de primeros puede estar claramente determinado.

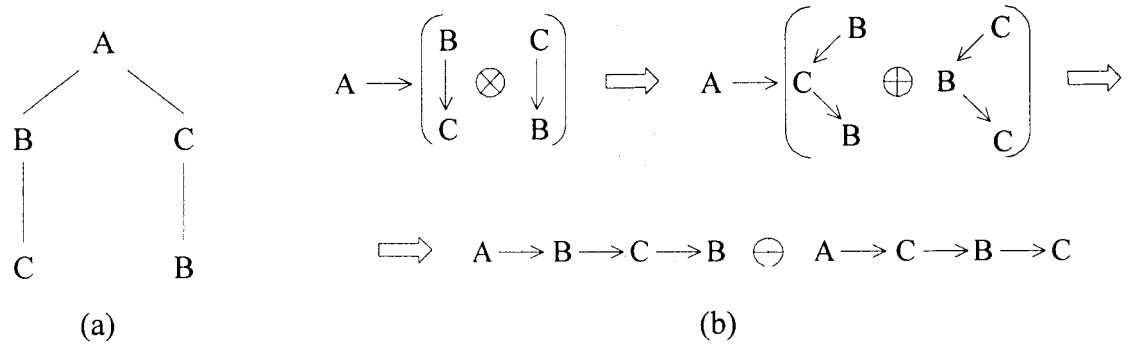


Figura 4.19: Ejemplo de indeterminación no asociada a $\mathcal{F}$.

Ejemplo 4.9:

En la Figura 4.19, para el árbol (a) se obtiene como secuencias óptimas las que aparecen en (b), donde se comprueba que existe una indeterminación en el número de ejemplares de B y C, que no son primeros de ninguna de las secuencias.

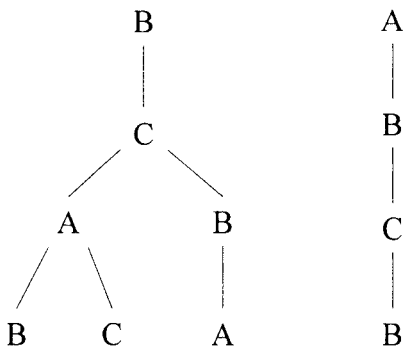
Observación 4.9:

Un mismo elemento puede aparecer en varias indeterminaciones. Incluso, una misma indeterminación puede aparecer repetida.

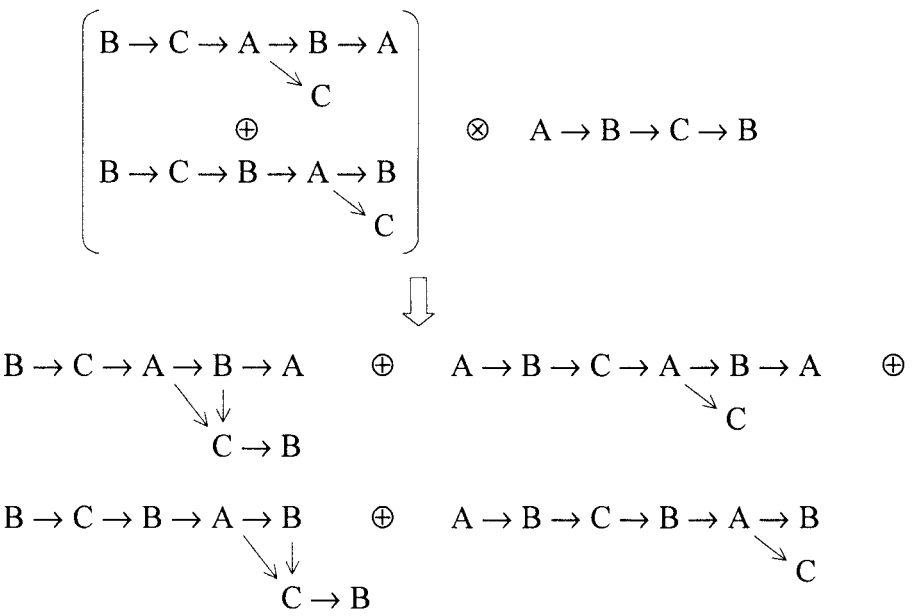
Ejemplo 4.10:

Las secuencias mínimas producidas al mezclar los árboles de la Figura 4.20 (a) tienen dos ejemplares de C, un ejemplar como mínimo de A y un mínimo de dos ejemplares de B, como se muestra en (b). Además, los dos componentes restantes pueden ser ambos A, ambos B o uno de cada uno de los elementos. Esto equivale a dos indeterminaciones en las que aparecen los elementos A y B. Nótese que, aunque una de las indeterminaciones se refiere a la elección de uno de los elementos como primero en la secuencia, en cuanto a la estructura resultante se le aplique la operación *push* enraizándole un elemento distinto, por ejemplo C, las dos indeterminaciones dejarán de estar asociadas al conjunto de primeros y tendrán el mismo tratamiento.

El nuevo modelo deberá pues incluir un multiconjunto $\mathcal{U}$ de indeterminaciones, dado que puede haber indeterminaciones repetidas. Una *indeterminación* se representará mediante un conjunto de elementos. De cada indeterminación, uno de sus elementos tendrá un ejemplar adicional a los que indica el multiconjunto. Como se ha observado, para un mismo elemento puede tenerse más de un ejemplar adicional, si aparece en más de una indeterminación. Además, como se ha mostrado, una indeterminación puede estar ligada al conjunto de primeros. En el Modelo 2 no se hará tal distinción, quedando



(a)



(b)

Figura 4.20: Ejemplo de indeterminaciones repetidas.

un modelo más simple. En el Modelo 3 sí consideraremos tal situación, complicando el tratamiento de las indeterminaciones.

Todavía es necesario realizar una consideración adicional, dado que pretendemos estimar con mayor exactitud el número de cambios de herramientas, y esto se hace analizando el caso de que ningún elemento perteneciente a los conjuntos de primeros fuera dominante. Todo ello sin quebrantar el carácter admisible de la heurística resultante.

Observación 4.10:

El análisis recursivo del árbol de precedencia de tareas nos llevó a utilizar el operador *merge* como un operador binario. Pero en último extremo, podría ocurrir que el pomset sobre el que se aplica la operación *push* fuera el resultado de aplicar el operador binario *merge* de manera sucesiva sobre varios pomsets. Sin embargo, como se observará en el siguiente ejemplo, el operador binario *merge* no es *asociativo*, pudiéndose obtener distintos resultados dependiendo del orden en que se consideran los distintos pomsets.

Ejemplo 4.11:

Sean $\mathcal{P}_1$=B→{B,2·C}, $\mathcal{P}_2$=C→{A} y $\mathcal{P}_3$=A→{2·B,C}. Queremos obtener el pomset resultante de la mezcla de los tres pomsets dados. Para ello consideraremos el uso del operador binario ⊗ según las expresiones $(\mathcal{P}_1 \otimes \mathcal{P}_2) \otimes \mathcal{P}_3$, $(\mathcal{P}_1 \otimes \mathcal{P}_3) \otimes \mathcal{P}_2$ y $\mathcal{P}_1 \otimes (\mathcal{P}_2 \otimes \mathcal{P}_3)$:

$\mathcal{P}_1 \otimes \mathcal{P}_2$ = B→{A,B,2·C}

$\mathcal{P}_1 \otimes \mathcal{P}_3$ = A→{2·B,2·C}

$\mathcal{P}_2 \otimes \mathcal{P}_3$ = C→{A,2·B,C} ⊕ A→{A,2·B,C}

$(\mathcal{P}_1 \otimes \mathcal{P}_2) \otimes \mathcal{P}_3$ = B→{A,2·B,2·C} ⊕ A→{A,2·B,2·C}

$(\mathcal{P}_1 \otimes \mathcal{P}_3) \otimes \mathcal{P}_2$ = A→{A,2·B,2·C} ⊕ C→{A,2·B,2·C}

$\mathcal{P}_1 \otimes (\mathcal{P}_2 \otimes \mathcal{P}_3)$ = B→{A,2·B,2·C} ⊕ A→{A,2·B,2·C} ⊕ C→{A,2·B,2·C}

Como puede observarse, el resultado varía para cada expresión. Para este ejemplo, el resultado correcto coincide con la última expresión.

¿Qué es lo que no se está considerando adecuadamente? Analizando el ejemplo anterior, se observa que los elementos que aparecían como primeros en sus respectivos pomsets no son dominantes, es decir, existe otro pomset en el que el número de ejemplares del elemento en cuestión no es menor que en el que aparece como primero. Tanto en $(\mathcal{P}_1 \otimes \mathcal{P}_2) \otimes \mathcal{P}_3$ como en $(\mathcal{P}_1 \otimes \mathcal{P}_3) \otimes \mathcal{P}_2$ no se detecta esa situación porque en el paso anterior de aplicación del operador ⊗ se desestimó el pomset que tenía un número mayor de elementos, que precisamente incluía como primer elemento al que luego no aparece como uno de los primeros posibles en el resultado final. Así, por ejemplo, en el caso de $(\mathcal{P}_1 \otimes \mathcal{P}_2) \otimes \mathcal{P}_3$, el elemento C, que aparecía como primero en $\mathcal{P}_1$, fue "absorbido" en la mezcla $\mathcal{P}_1 \otimes \mathcal{P}_2$, por lo que la posterior mezcla con $\mathcal{P}_3$ no lo considera como posible primero, únicamente en el caso de que los primeros de $\mathcal{P}_3$ y de $\mathcal{P}_1 \otimes \mathcal{P}_2$ no fueran dominantes.

La forma de corregir esta situación se intuye: cuando un pomset se ha obtenido como un resultado intermedio de una mezcla múltiple, debe mantener el recuerdo de los elementos que aparecían como primeros en los pomsets originales y que fueron

desestimados como tales. Sólo se tendrán en consideración cuando el pomset se mezcle con otro de forma que los primeros elementos no sean dominantes. En tal circunstancia, como se ha visto en el Ejemplo 4.11, esos elementos pueden aparecer como primeros en algunos de los alineamientos óptimos del pomset resultante de la mezcla completa.

Nótese que el modelo anterior no es necesario realizar esta consideración, debido a que se subestima la solución, como se ha observado en el corolario de la Proposición 4.1, y como puede comprobarse en la continuación del Ejemplo 4.11.

Ejemplo 4.11 (*continuación*):

La aplicación de la operación *merge* definida en el modelo inicial daría lugar a:

$merge\ (\mathcal{P}_1,\mathcal{P}_2) = <\{A,2\cdot B,2\cdot C\},\{B\}>$

$merge\ (\mathcal{P}_1,\mathcal{P}_3) = <\{A,2\cdot B,2\cdot C\},\{A\}>$

$merge\ (\mathcal{P}_2,\mathcal{P}_3) = <\{A,2\cdot B,C\},\{A,C\}>$

$merge\ (merge\ (\mathcal{P}_1,\mathcal{P}_2),\mathcal{P}_3) = <\{A,2\cdot B,2\cdot C\},\{A,B\}>$

$merge\ (merge\ (\mathcal{P}_1,\mathcal{P}_3),\mathcal{P}_2) = <\{A,2\cdot B,2\cdot C\},\{A,C\}>$

$merge\ (\mathcal{P}_1,merge\ (\mathcal{P}_2,\mathcal{P}_3)) = <\{A,2\cdot B,2\cdot C\},\{A\}>$

Como se observa, siguen saliendo resultados distintos, dependiendo del orden considerado. En todos los casos, el número de elementos de los alineamientos óptimos (5) es menor que el óptimo real que obtendríamos al mezclar los pomsets originales (6), según se vio en la primera parte del ejemplo. Luego la heurística correspondiente al *Modelo 1* ofrece una solución por debajo del óptimo, por lo que es indiferente el que el conjunto de primeros esté compuesto por unos u otros elementos. Si hubiera que elegir el mejor resultado, ése sería el que realiza una simplificación menor, y esto ocurre, al tener todos los resultados el mismo número de elementos, para la última expresión, que sólo ofrece una posibilidad para el primer elemento de las secuencias óptimas.

Nótese que la aplicación de la operación *push* sobre un pomset no necesita considerar a los elementos despreciados como primeros en la aplicación de *merge*. La razón es obvia: la operación *push* define claramente cuál va a ser el primer elemento de cualquier alineamiento óptimo del pomset.

Así pues, para mantener la admisibilidad de la heurística, vamos a incluir en el nuevo modelo un nuevo conjunto, al que denotaremos como $\mathcal{F}_m$, en el que estarán aquellos elementos que han sido descartados como primeros tras la aplicación de la operación *merge*, por no ser dominantes. Estos elementos aparecerían en la indeterminación que se crearía si en una posterior aplicación de *merge* los elementos de los conjuntos de primeros no fueran dominantes. En tal situación, además, deben considerarse como posibles primeros elementos de los alineamientos óptimos del pomset resultante. Sin embargo, dejarían de tenerse en cuenta al aplicar la operación *push* sobre el pomset.

A la estructura resultante la llamaremos *u-pomset*, denotándose mediante $\mathcal{P}^u = < \mathcal{B}, \mathcal{F}, \mathcal{F}_m, \mathcal{U} >$.

Las ecuaciones (4.24) y (4.25) siguen siendo válidas para la formación de los u-pomsets asociados a cada tarea y máquina. Obviamente, las que cambiarán serán las definiciones de las operaciones *push* y *merge* que los manipulen. Teniendo en cuenta que seguimos persiguiendo un modelo que realice una estimación optimista del número de cambios de herramienta, con objeto de que la heurística sea admisible, las simplificaciones que se realicen deberán respetar tal objetivo.

La definición de *push* se presenta en la ecuación (4.34):

$$push(x, < \mathcal{B}, \mathcal{F}, \mathcal{F}_m, \mathcal{U} >) \equiv \begin{cases} < \mathcal{B}, \{x\}, \varnothing, \mathcal{U} > & \text{si } x \in \mathcal{F} \\ < \mathcal{B} \uplus \{x\}, \{x\}, \varnothing, \mathcal{U} > & \text{si } x \notin \mathcal{F} \end{cases} \tag{4.34}$$

Obsérvese la simplificación que se produce cuando se mantiene las indeterminaciones en el caso $x \in \mathcal{F}$. Hay situaciones, como la que sucede en el Ejemplo 4.12, en que el modelo no tiene conocimiento suficiente para resolver las indeterminaciones en favor de uno de los elementos. Nótese que, en cualquier caso, el modelo permite una estimación optimista, al incluir entre las posibles secuencias óptimas a otras que no pueden darse, pero manteniendo las que sí son óptimas.

Ejemplo 4.12:

Para el árbol de la Figura 4.21, las posibles secuencias óptimas tienen a A como primer elemento, y a continuación se tendrá un ejemplar de A, B y C, en un orden compatible. El u-pomset que se obtiene al aplicar el *Modelo 2* es

push(A, *merge*(<{A,B,C},{A},∅,∅>,<{A,B},{B},∅,∅>)) =
push(A, <{A,B,C},{A,B},∅,{{A,B}}>) = <{A,B,C},{A},∅,{{A,B}}>.

Aunque los detalles de la definición de *merge* se verán después, es fácil ver que, según la descripción que se ha hecho del modelo, se genera una indeterminación intermedia al no ser dominantes ninguno de los elementos que aparecían como

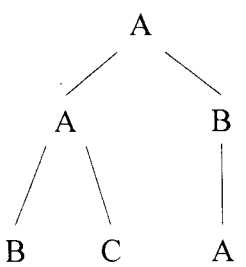


Figura 4.21: Simplificación en *push* para el Modelo 2.

primeros. Tal indeterminación no es distinguida por el modelo como asociada al conjunto de primeros.

El ejemplo anterior constituye un caso atípico, ya que sólo hay un elemento en $\mathcal{B}$ de A y B, con lo que la indeterminación, que estaría ligada forzosamente al conjunto de primeros, podría haberse eliminado. La operación *push* podría contemplar como casos distintos que el número de elementos en el multiconjunto de los elementos de la indeterminación que se está cuestionando mantener es 1 o mayor que 1, complicándose así su definición.

Antes de dar la definición de la operación *merge* es necesario todavía realizar una serie de observaciones que nos guiarán a la misma, justificando la admisibilidad de la heurística resultante.

Observación 4.11:

Sean $\mathcal{P}_1^u = <\mathcal{B}_1, \mathcal{F}_1, \mathcal{F}_{m1}, \{\mathcal{S}\}>$ y $\mathcal{P}_2^u = <\mathcal{B}_2, \mathcal{F}_2, \mathcal{F}_{m2}, \varnothing>$. Si alguno de los elementos $x \in \mathcal{S}$ es tal que $\#(x, \mathcal{B}_1) < \#(x, \mathcal{B}_1)$, entonces la indeterminación $\mathcal{S}$ no deberá aparecer en el u-pomset resultante de la mezcla de $\mathcal{P}_1^u$ y $\mathcal{P}_2^u$. Se dice entonces que la indeterminación *no* es *dominante*.

En efecto, si algún elemento x de $\mathcal{S}$ apareciera más veces en $\mathcal{B}_2$ que en $\mathcal{B}_1$, la indeterminación se podría resolver tomando a x como el elemento con un ejemplar adicional, y siendo absorbidos todos los x de $\mathcal{B}_1$ en los de $\mathcal{B}_2$. Esta solución sería mejor que cualquiera que mantuviera la indeterminación o la resolviera en favor de otro elemento en el que su número de apariciones en $\mathcal{B}_1$ fuera mayor o igual que en $\mathcal{B}_2$.

La observación anterior se puede extender a los casos en que haya varias indeterminaciones en cada u-pomset, siempre que ninguna de ellas contenga ningún elemento existente en otra. El razonamiento es idéntico. En tales casos, la condición anterior puede tomarse como necesaria y suficiente, es decir, si todos los elementos de la indeterminación son tales que su número de ejemplares en el mismo multiconjunto no es inferior al del otro, entonces la indeterminación es dominante y además puede aparecer en el u-pomset resultante de la mezcla respetando la admisibilidad de la heurística.

Cuando un elemento aparece en más de una indeterminación, sea del mismo u-pomset o del contrario, la condición anterior sólo sería suficiente. El razonamiento se complica porque el número de ejemplares del elemento en cuestión depende de si es escogido o no en las otras indeterminaciones.

Dado que tenemos que garantizar que el tamaño de las secuencias mínimas dadas por el pomset resultante de aplicar la operación *merge* según el modelo no supere el tamaño de la secuencia óptima real, debemos seguir ciertos criterios a la hora de realizar simplificaciones:

1. No debe favorecerse el mantenimiento de indeterminaciones, salvo en la total seguridad de ser dominante, y que, en tal caso, puedan ser escogidas cuando entren en conflicto con otras del u-pomset contrario. Nótese que en el caso de tomar como dominante a una indetermianción que en realidad no lo es estaríamos suponiendo un elemento adicional al que podríamos tener en el óptimo. Para llevar a cabo lo anterior:

   (a) No se tendrá en cuenta ninguna otra indeterminación en la que aparezca el elemento para el que se está considerando su número de ejemplares.

   (b) El número de ejemplares de los elementos en el otro u-pomset se toma como el que aparece en el multiconjunto. De esta forma, las indeterminaciones dominantes con elementos comunes deberán ser escogidas adecuadamente, como se verá más adelante.

2. Debe favorecerse la consideración de dominantes para los elementos de $\mathcal{F}$. Téngase en cuenta que no considerar como dominantes a los elementos de $\mathcal{F}$ puede generar una indeterminación adicional, y por tanto un elemento más en las secuencias óptimas asociadas al u-pomset. Así pues, sólo cuando haya suficientes garantías de que efectivamente el elemento de $\mathcal{F}$ no es dominante será considerado como tal. Para llevar a cabo lo anterior, se atenderá a lo siguiente en cuanto al número de ejemplares de cada elemento en los pomsets involucrados:

   (a) Una vez determinadas las indeterminaciones dominantes del u-pomset, el número de ejemplares que se tiene en cuenta para cada elemento de $\mathcal{F}$ es el número de ejemplares en el multiconjunto más el número de indeterminaciones dominantes en las que aparece.

   (b) En el u-pomset contrario, el número de ejemplares que se considera es exclusivamente el correspondiente al multiconjunto, suponiendo así que el elemento en cuestión no va a ser escogido en ninguna indeterminación.

Según lo anterior, una indeterminación se considerará como *dominante* cuando el número de apariciones de todos sus elementos en el u-pomset, en el caso más desfavorable, no sea inferior al número de apariciones en el otro u-pomset. Así,

$$\mathcal{D}'_{\mathcal{U}}\left(\mathcal{U}_i,\mathcal{B}_i,\mathcal{B}_j\right)\equiv\left\{\mathcal{S}\in\mathcal{U}_i \mid \forall x\in\mathcal{S}\bullet\#\left(x,\mathcal{B}_i\right)\geq\#\left(x,\mathcal{B}_j\right)\right\} \tag{4.35}$$

representa el conjunto de indeterminaciones dominantes del u-pomset $\mathcal{P}_i^u = < \mathcal{B}_i, \mathcal{F}_i, \mathcal{F}_{mi}, \mathcal{U}_i >$ en su mezcla con el u-pomset $\mathcal{P}_j^u = < \mathcal{B}_j, \mathcal{F}_j, \mathcal{F}_{mj}, \mathcal{U}_j >$.

En cuanto a los elementos de $\mathcal{F}$, la ecuación (4.32) puede ser usada para determinar los que son dominantes, debiendo aparecer en el conjunto de primeros del u-pomset resultante de la mezcla, teniendo en cuenta que el número de ejemplares de cada

elemento viene incrementado con el número de indeterminaciones dominantes en que aparece en el u-pomset. Así,

$$\mathcal{D}'_{\mathcal{F}}\left(<\mathcal{B}_i \uplus retainAll\left(\widetilde{\mathcal{D}'_{\mathcal{U}}(\mathcal{U}_i,\mathcal{B}_i,\mathcal{B}_j)},\mathcal{F}_i\right),\mathcal{F}_i>,\mathcal{P}_j\right) \tag{4.36}$$

representa el conjunto de elementos dominantes del u-pomset $\mathcal{P}_i^u = < \mathcal{B}_i, \mathcal{F}_i, \mathcal{F}_{mi}, \mathcal{U}_i >$ en su mezcla con el u-pomset $\mathcal{P}_j^u = < \mathcal{B}_j, \mathcal{F}_j, \mathcal{F}_{mj}, \mathcal{U}_j >$. En (4.36), $\mathcal{P}_j$ representa el pomset equivalente según la definición que se hizo en el modelo 1, es decir, compuesto por la tupla $< \mathcal{B}_j, \mathcal{F}_j >$.

El siguiente ejemplo es una muestra de las simplificaciones que se llevan a cabo con los criterios tomados.

Ejemplo 4.13:

Sean $\mathcal{P}_1^u$ = <{A,2·B,3·C},{C},∅,{{A,B},{A,B}}> y $\mathcal{P}_2^u$ = <{2·A,2·B,2·C},{A},∅, ∅>. Comparemos los resultados de la mezcla de ambos u-pomset según usemos los pasos indicados en la Observación 4.3 y la definición del Modelo 2. La descomposición en α-pomsets de $\mathcal{P}_1^u$ es

$$\mathcal{P}_1^u = C\rightarrow\{3\cdot A,2\cdot B,2\cdot C\} \oplus C\rightarrow\{2\cdot A,3\cdot B,2\cdot C\} \oplus C\rightarrow\{A,4\cdot B,2\cdot C\}$$

Teniendo en cuenta la Observación 4.3, el resultado sería

$$\begin{aligned}\mathcal{P}_1^u \otimes \mathcal{P}_2^u = \; & C\rightarrow\{3\cdot A,2\cdot B,2\cdot C\} \oplus C\rightarrow\{2\cdot A,3\cdot B,2\cdot C\} \oplus \cancel{C\rightarrow\{2\cdot A,4\cdot B,2\cdot C\}} \oplus \\ & \cancel{A\rightarrow\{3\cdot A,2\cdot B,3\cdot C\}} \oplus \cancel{A\rightarrow\{2\cdot A,3\cdot B,3\cdot C\}} \oplus \cancel{A\rightarrow\{A,4\cdot B,3\cdot C\}} \\ \Rightarrow \; & <\{2\,A,2\cdot B,3\cdot C\},\{C\},\{A\},\{\{A,B\}\}>\end{aligned}$$

Según los criterios indicados para el Modelo 2, las indeterminaciones de $\mathcal{P}_1^u$ no serían dominantes, y teniendo en cuenta eso, A sería dominante como primero en $\mathcal{P}_1^u$. Aparte de todo lo anterior, es claro que C también es dominante como primero en $\mathcal{P}_2^u$. El resultado que daría la aplicación de la operación *merge* según el modelo 2 sería *merge* ($\mathcal{P}_1^u$, $\mathcal{P}_2^u$) = <{2·A,2·B,3·C},{A,C},∅,∅>. Como se observa, el modelo no contempla una de las indeterminaciones que permanecerían de aplicar estrictamente la mezcla de u-pomsets. En cualquier caso, la estimación que realiza es optimista.

Aunque el ejemplo anterior puede parecer fácil de analizar y por lo tanto tener en cuenta en el modelo tal análisis, es fácil imaginar que en cuanto las indeterminaciones no sean idénticas y participen varios elementos que a su vez puedan estar involucrados en otras indeterminaciones, la complejidad aumenta bastante, lo cual justifica llevar a cabo las simplificaciones indicadas.

Observación 4.12:

Según los criterios tomados a partir de la observación anterior, podría pensarse que por no tener en cuenta como dominantes algunas indeterminaciones donde aparezca un elemento $x \in \mathcal{F}$ podría no determinarse a x como dominante, con lo que podría generarse una ingerminación si ningún otro elemento primero es dominante, haciendo peligrar la admisibilidad. El caso a considerar es cuando un elemento aparece en varias indeterminaciones, ya que en el caso en que sólo aparezca en una no hay duda en el número de ejemplares. Como se verá en el siguiente ejemplo, lo anterior no puede ocurrir, es decir, la indeterminación que se genera porque los elementos de $\mathcal{F}$ no son dominantes contrarresta a la no consideración de las indeterminaciones como dominantes.

Ejemplo 4.14:

Sean $\mathcal{P}_1^u = <\{2\cdot A,2\cdot B,3\cdot C\},\{A\},\varnothing,\{\{A,B\},\{A,B\}\}>$ y $\mathcal{P}_2^u = <\{3\cdot A,2\cdot B,2\cdot C\},\{C\},\varnothing,\varnothing>$. Comparemos los resultados de la mezcla de ambos u-pomset según usemos los pasos indicados en la Observación 4.3 y la definición del Modelo 2. La descomposición en α-pomsets de $\mathcal{P}_1^u$ es

$$\mathcal{P}_1^u = A \rightarrow \{3\cdot A,2\cdot B,3\cdot C\} \oplus A \rightarrow \{2\cdot A,3\cdot B,3\cdot C\} \oplus A \rightarrow \{A,4\cdot B,3\cdot C\}$$

Teniendo en cuenta la Observación 4.3, el resultado sería

$$\begin{aligned} \mathcal{P}_1^u \otimes \mathcal{P}_2^u = {} & A \rightarrow \{3\cdot A,2\cdot B,3\cdot C\} \oplus \cancel{A \rightarrow \{3\cdot A,3\cdot B,3\cdot C\}} \oplus \cancel{A \rightarrow \{3\cdot A,4\cdot B,3\cdot C\}} \oplus \\ & \cancel{C \rightarrow \{4\cdot A,2\cdot B,3\cdot C\}} \oplus \cancel{C \rightarrow \{3\cdot A,3\cdot B,3\cdot C\}} \oplus \cancel{C \rightarrow \{3\cdot A,4\cdot B,3\cdot C\}} \\ \Rightarrow {} & <\{4\cdot A,2\cdot B,3\cdot C\},\{A\},\{C\},\varnothing> \end{aligned}$$

Según los criterios indicados para el Modelo 2, las indeterminaciones de $\mathcal{P}_1^u$ no serían dominantes, y teniendo en cuenta eso, A tampoco sería dominante como primero en $\mathcal{P}_1^u$. Aparte de todo lo anterior, es claro que C tampoco es dominante como primero en $\mathcal{P}_2^u$, con lo que se generaría una indeterminación conteniendo a los elementos A y C. El resultado que daría la aplicación de la operación *merge* según el modelo 2 sería pues *merge* $(\mathcal{P}_1^u,\mathcal{P}_2^u) = <\{3\cdot A,2\cdot B,3\cdot C\}, \{A,C\}, \varnothing, \{\{A,C\}\}>$. Como se observa, el número total de elementos en las secuencias óptimas es el mismo en ambos casos. El modelo incluye al resultado que se obtiene en realidad, incluyendo también el caso en que C pueda aparecer como primero. La estimación es pues optimista.

La siguiente consideración es respecto a cuando se tienen indeterminaciones dominantes en ambos u-pomsets. Como se indicó antes, las indeterminaciones que no tengan elementos comunes con las del otro pomset pueden ser escogidas para formar parte del u-pomset resultante.

Observación 4.13:

> Sean $\mathcal{P}_1^u = <\mathcal{B}_1, \mathcal{F}_1, \mathcal{F}_{m1}, \mathcal{U}_1>$ y $\mathcal{P}_2^u = <\mathcal{B}_2, \mathcal{F}_2, \mathcal{F}_{m2}, \mathcal{U}_2>$. Sean n' y m' el número de indeterminaciones dominantes en ambos u-pomsets al considerar su mezcla. Entonces, el número de elementos a añadir a los del multiconjunto resultante de la unión de los multiconjuntos de ambos pomsets en una secuencia óptima es como mínimo el valor máximo entre n' y m'.

Nótese que cuando una indeterminación dominante contiene a un elemento que también se encuentra en otra indeterminación dominante en el otro pomset, no se pueden contemplar ambas indeterminaciones, ya que se añadirían elementos adicionales y las secuencias no serían óptimas.

Ejemplo 4.15:

> Sean $\mathcal{P}_1^u = <\{A,B,C\},\{A\},\varnothing,\{\{B,C\}\}>$ y $\mathcal{P}_2^u = <\{A,C,D\},\{A\},\varnothing,\{\{C,D\}\}>$. La mezcla de ambos u-pomsets nos daría:
>
> $$(A \rightarrow \{2\cdot B,C\} \oplus A \rightarrow \{B,2\cdot C\}) \otimes (A \rightarrow \{2\cdot C,D\} \oplus A \rightarrow \{C,2\cdot D\}) =$$
> $$A \rightarrow \{2\cdot B,2\cdot C,D\} \oplus A \rightarrow \{2\cdot B,C,2\cdot D\} \oplus A \rightarrow \{B,2\cdot C,D\} \oplus A \rightarrow \{B,2\cdot C,2\cdot D\}$$
>
> Como puede observarse, ambas indeterminaciones son dominantes, y en el resultado de la mezcla hay un α-pomset, el tercero, que contiene 5 elementos, correspondiente a que el elemento común en las indeterminaciones, C, ha sido escogido, con lo que sólo una indeterminación ha sido efectiva.

Observación 4.14:

> Para mantener admisible la heurística resultante, no se deben resolver las indeterminaciones dominantes de distintos u-pomsets en favor de los elementos comunes.

Del Ejemplo 4.15 podría concluirse que la mejor solución se corresponde precisamente con el supuesto anterior. Sin embargo, con ello se estaría obligando que el número de ejemplares del elemento C debe ser 2 no sólo en cualquier alineamiento óptimo del u-pomset resultante, sino también en los resultantes de futuras mezclas con otros. Como se verá en el siguiente ejemplo, eso no será válido.

Ejemplo 4.16:

> Dados los u-pomsets del Ejemplo 4.15, queremos obtener la mezcla óptima de ambos junto con $\mathcal{P}_3^u = <\{A,2\cdot B,2\cdot D\},\{A\},\varnothing,\varnothing> = A \rightarrow \{2\cdot B,2\cdot D\}$. El resultado

real que obtendríamos vendría dado por la mezcla de este último con cada uno de los α-pomsets que se obtuvieron en el Ejemplo 4.15, resultando:

$$A \rightarrow \{2\cdot B,2\cdot C,2\cdot D\} \oplus A \rightarrow \{2\cdot B,C,2\cdot D\} \oplus A \rightarrow \{2\cdot B,2\cdot C,2\cdot D\} \oplus A \rightarrow \{2\cdot B,2\cdot C,2\cdot D\}$$

Como se observa, el mejor α-pomset resultante, el segundo, sólo tiene un ejemplar de C, y sólo se puede obtener si no se resuelven las indeterminaciones de $\mathcal{P}_1^u$ y $\mathcal{P}_2^u$ en favor del elemento común.

Puede advertirse también que, de los cuatro α-pomsets resultantes en el Ejemplo 4.15, sólo el primero y el cuarto son peores que el tercero, ya que ninguno de los elementos contiene menos ejemplares que en éste último. El segundo no se puede despreciar, y se ha comprobado en el Ejemplo 4.16, porque C tiene menos ejemplares que en el tercer α-pomset, aunque en su conjunto tenga mayor número de elementos.

De las observaciones anteriores, se concluye que pueden entonces tomarse como indeterminaciones a tener en cuenta en el u-pomset resultante las del u-pomset que mantenga el mayor número de indeterminaciones dominantes. En el caso de que existan en el otro u-pomset indeterminaciones cuyos elementos no estén en las indeterminaciones dominantes elegidas, ésas también pueden ser escogidas como indeterminaciones en el u-pomset resultante. Pero si hubiera solapamiento, se pueden despreciar, de manera que no se pierden soluciones óptimas, más al contrario, se permiten soluciones nuevas al no considerar las restricciones asociadas a las indeterminaciones despreciadas.

El proceso para escoger las indeterminaciones dominantes de ambos u-pomsets sería el que sigue. Sea una indeterminación dominante $S \in \mathcal{U}_i$. Si ningún elemento de S aparece en ninguna indeterminación dominante de $\mathcal{U}_j$, entonces S será una de las indeterminaciones que aparezcan en el u-pomset resultante de la mezcla de $\mathcal{U}_i$ y $\mathcal{U}_j$. Tengamos entonces aquellas otras indeterminaciones dominantes $S' \in \mathcal{U}_i$ tales que contienen algún elemento que aparece en alguna indeterminación dominante de $\mathcal{U}_j$, y viceversa. Las indeterminaciones a escoger de entre las anteriores serán las correspondientes al u-pomset que tenga más dominantes aún no elegidas. En la Figura 4.22 se muestran en trazo continuo las indeterminaciones dominantes que se escogerían según lo indicado, y en discontinuo la que se desecha.

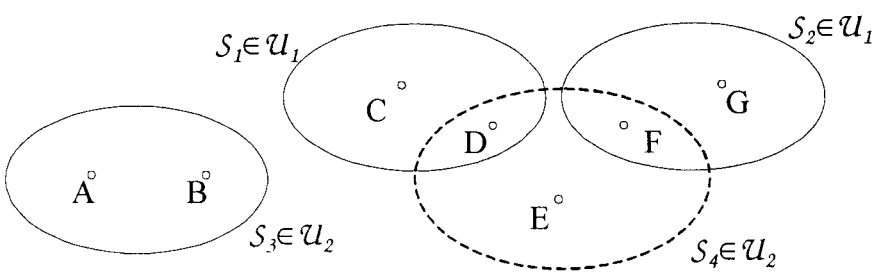


Figura 4.22: Selección de indeterminaciones dominantes (1).

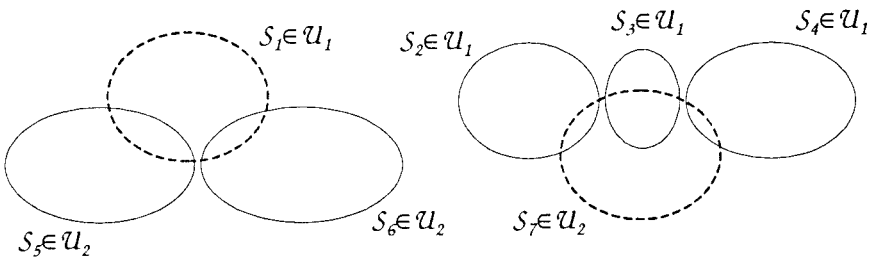


Figura 4.23: Selección de indeterminaciones dominantes (2).

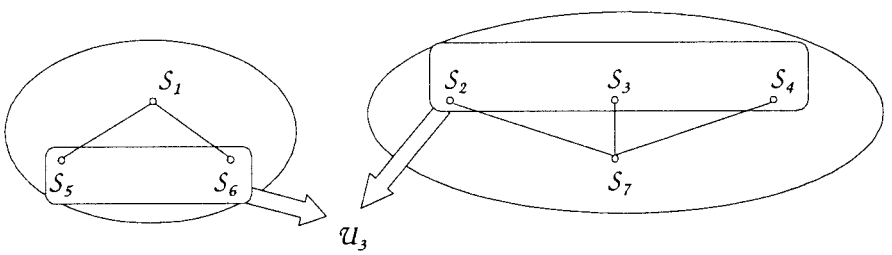


Figura 4.24: Selección de indeterminaciones dominantes (2) – grafo bipartito.

Todavía podría afinarse más si tenemos en cuenta que pueden formarse distintos grupos de indeterminaciones solapadas disjuntos entre sí. En el ejemplo de la Figura 4.23, se podrían escoger las indeterminaciones correspondientes al trazo continuo, rechazando las de trazo discontinuo. El número de indeterminaciones que se obtiene es superior al que se alcanzaría con el criterio anterior, logrando pues mayor exactitud. Esto se puede visualizar mediante un *grafo bipartito*, en donde los nodos correspondientes a un lado del grafo serían las indeterminaciones de uno de los u-pomsets y los del otro lado serían las del otro. Entre dos nodos de ambos opuestos existirá una arista si las indeterminaciones correspondientes incluyen algún elemento común. Para establecer las indeterminaciones que deben ser escogidas para el u-pomset resultante, se determinan los grupos conexos que hay en el grafo, y por cada uno de ellos se escogen las indeterminaciones según el criterio indicado antes: todas las del lado del grafo que supongan mayor (o igual) número de nodos que en el otro. La Figura 4.24 muestra lo anterior para el ejemplo de la Figura 4.23.

De todas las observaciones anteriores se desprende la definición de la operación *merge*, dada por el algoritmo de la Figura 4.25, en el que se usa el algoritmo de la Figura 4.26 para la selección de las indeterminaciones dominantes, *selectU*. En este algoritmo no se han tenido en cuenta las últimas consideraciones sobre grafos bipartitos.

La definición de *ntrans* para el Modelo 2 es ahora:

$$ntrans(\mathcal{P}^u) = \sum_{x \in \mathcal{B}} \#(x, \mathcal{B}) - 1 + \#\mathcal{U} \tag{4.37}$$

Algoritmo *merge* $(\mathcal{P}_1^u, \mathcal{P}_2^u)$ **devuelve** $\mathcal{P}_3^u$

$\mathcal{B}_3 := \mathcal{B}_1 \cup \mathcal{B}_2$

$\mathcal{D}_{\mathcal{U}1} := \mathcal{D}'_{\mathcal{U}}(\mathcal{U}_1, \mathcal{B}_1, \mathcal{B}_2)$

$\mathcal{D}_{\mathcal{U}2} := \mathcal{D}'_{\mathcal{U}}(\mathcal{U}_2, \mathcal{B}_2, \mathcal{B}_1)$

$\mathcal{B}_{\mathcal{FDU}1} := \mathcal{B}_1 \uplus retainAll(\widehat{\mathcal{D}_{\mathcal{U}1}}, \mathcal{F}_1)$

$\mathcal{B}_{\mathcal{FDU}2} := \mathcal{B}_2 \uplus retainAll(\widehat{\mathcal{D}_{\mathcal{U}2}}, \mathcal{F}_2)$

$\mathcal{F}_3 := (\mathcal{F}_1 \cap \mathcal{F}_2) \cup \mathcal{D}'_{\mathcal{F}}(<\mathcal{B}_{\mathcal{FDU}1}, \mathcal{F}_1>, \mathcal{P}_2) \cup \mathcal{D}'_{\mathcal{F}}(<\mathcal{B}_{\mathcal{FDU}2}, \mathcal{F}_2>, \mathcal{P}_1)$

$\mathcal{F}_{m3} := (\mathcal{F}_1 \cup \mathcal{F}_{m1} \cup \mathcal{F}_2 \cup \mathcal{F}_{m2}) - \mathcal{F}_3$

$\mathcal{U}_3 := selectU(\mathcal{D}_{\mathcal{U}1}, \mathcal{D}_{\mathcal{U}2})$

si $\mathcal{F}_3 = \varnothing$

$\mathcal{F}_3 := \mathcal{F}_{m3}$

$\mathcal{U}_3 := \mathcal{U}_3 \uplus \{\mathcal{F}_{m3}\}$

$\mathcal{F}_{m3} := \varnothing$

fsi

devolver $<\mathcal{B}_3, \mathcal{F}_3, \mathcal{F}_{m3}, \mathcal{U}_3>$

Figura 4.25: Definición de *merge* para el *Modelo 2*.

siendo $\#\mathcal{U}$ el número de indeterminaciones del pomset.

MODELO 3

Se distinguirán ahora las indeterminaciones según puedan estar asociadas al conjunto de primeros o no. Así, tendremos un conjunto $\mathcal{U}_{\mathcal{F}}$ que incluirá a aquellas indeterminaciones cuyos elementos tendrán un ejemplar adicional si fueran escogidos como primeros en la secuencia, y en caso contrario mantendrían el número de ejemplares. En este caso, los conjuntos asociados a las indeterminaciones son disjuntos, y la intersección con el conjunto $\mathcal{F}$ también es vacía.

Por otro lado, tendremos un multiconjunto $\mathcal{U}_{\mathcal{R}}$, que incluirá a aquellas indeterminaciones tales que la elección de un ejemplar adicional para sus elementos no esté relacionada con que el elemento aparezca como primero en la secuencia. Más aún, los ejemplares adicionales no se corresponden con el primer elemento de la secuencia. Los conjuntos de $\mathcal{U}_{\mathcal{R}}$ sí podrán intersectar unos con otros, y tendremos los mismos condicio-

Algoritmo *selectU* $(\mathcal{U}_1, \mathcal{U}_2)$ **devuelve** $\mathcal{U}_3$

$\mathcal{U}_3 := \varnothing$

para cada $S \in \mathcal{U}_1$

si $S \cap \widehat{\mathcal{U}_2} = \varnothing$

$\mathcal{U}_3 := \mathcal{U}_3 \cup S$; $\mathcal{U}_1 := \mathcal{U}_1 - \{S\}$

fsi

fpara

para cada $S \in \mathcal{U}_2$

si $S \cap \widehat{\mathcal{U}_1} = \varnothing$

$\mathcal{U}_3 := \mathcal{U}_3 \cup S$; $\mathcal{U}_2 := \mathcal{U}_2 - \{S\}$

fsi

fpara

si $\#\mathcal{U}_1 \geq \#\mathcal{U}_2$

$\mathcal{U}_3 := \mathcal{U}_3 \uplus \mathcal{U}_1$

si no

$\mathcal{U}_3 := \mathcal{U}_3 \uplus \mathcal{U}_2$

fsi

devolver $\mathcal{U}_3$

Figura 4.26: Selección de indeterminaciones dominantes en el *Modelo 2*.

nantes que para $\mathcal{U}$ en el Modelo 2. Nótese que, según vimos en el Ejemplo 4.10, puede haber indeterminaciones en $\mathcal{U}_{\mathcal{F}}$ y $\mathcal{U}_{\mathcal{R}}$ en las que aparezca un mismo elemento, incluso que las indeterminaciones coincidan.

A la estructura resultante la llamaremos *ufr-pomset*, denotándose mediante $\mathcal{P}^{ufr} = < \mathcal{B}, \mathcal{F}, \mathcal{F}_m, \mathcal{U}_{\mathcal{F}}, \mathcal{U}_{\mathcal{R}} >$.

Las ecuaciones (4.24) y (4.25) siguen siendo válidas para la formación de los ufr-pomsets asociados a cada tarea y máquina. Nuevamente, las que cambiarán serán las definiciones de las operaciones *push* y *merge* que los manipulen, así como la operación *ntrans*.

La mayoría de las observaciones realizadas para definir el Modelo 2 son válidas para el nuevo modelo, ya que la única diferencia está en que se distinguen las indeterminaciones según estén ligadas al conjunto de primeros o no.

La definición de *push* correspondiente al Modelo 3 viene dada por la ecuación:

$$push\left(x,<\mathcal{B},\mathcal{F},\mathcal{F}_m,\mathcal{U}_{\mathcal{F}},\mathcal{U}_{\mathcal{R}}>\right)\equiv\begin{cases}<\mathcal{B},\{x\},\varnothing,\varnothing,\mathcal{U}_{\mathcal{R}}\uplus\mathcal{U}_{\mathcal{F}}> & \text{si } x\in\mathcal{F}\\ <\mathcal{B}\uplus\{x\},\{x\},\varnothing,\varnothing,\mathcal{U}_{\mathcal{R}}\uplus(\mathcal{U}_{\mathcal{F}}-\mathcal{S})> & \text{si } \exists\mathcal{S}\in\mathcal{U}_{\mathcal{F}}\,|\,x\in\mathcal{S}\\ <\mathcal{B}\uplus\{x\},\{x\},\varnothing,\varnothing,\mathcal{U}_{\mathcal{R}}\uplus\mathcal{U}_{\mathcal{F}}> & \text{si } x\notin\mathcal{S},\forall\mathcal{S}\in\mathcal{U}_{\mathcal{F}}\end{cases} \quad (4.38)$$

Recuérdese que la operación *push* no produce indeterminaciones. En todo caso, elimina las que puedan existir en $\mathcal{U}_{\mathcal{F}}$, trasladándolas a $\mathcal{U}_{\mathcal{R}}$. Nótese también que, en el caso de que el elemento a añadir esté presente en una indeterminación de $\mathcal{U}_{\mathcal{F}}$, ésta se resolvería en favor de ese elemento, el cual tomaría un ejemplar más en el multiconjunto. Sin embargo, el número total de elementos de las secuencias óptimas del ufr-pomset no cambiaría al desaparecer una indeterminación.

Antes de proceder a la definición de *merge* es necesario realizar algunas observaciones adicionales.

Observación 4.15:

La condición de dominancia para las indeterminaciones de $\mathcal{U}_{\mathcal{F}}$ es la misma que para las de $\mathcal{U}_{\mathcal{R}}$, refiriéndonos al número de ejemplares a considerar para cada elemento de las mismas.

Observación 4.16:

La aplicación de las condiciones de dominancia puede hacer que un elemento se encuentre como dominante en el conjunto $\mathcal{F}$ de uno de los ufr-pomsets y en una indeterminación dominante de $\mathcal{U}_{\mathcal{F}}$ del otro. Tal situación se daría en las siguientes circunstancias. Si $x\in\mathcal{S}\in\mathcal{U}_{\mathcal{F}2}$, siendo $\mathcal{S}$ dominante, entonces deberá ocurrir que $\#(x,\mathcal{B}_2)\geq\#(x,\mathcal{B}_1)$. Para que $x\in\mathcal{F}_1$ sea dominante, deberá ocurrir una de las siguientes condiciones:

- $\#(x,\mathcal{B}_1)>\#(x,\mathcal{B}_2)$
- $\#(x,\mathcal{B}_1\uplus\mathcal{S}')>\#(x,\mathcal{B}_2)$, con $x\in\mathcal{S}'\in\mathcal{U}_{\mathcal{R}1}$, siendo $\mathcal{S}'$ dominante, y por tanto, $\#(x,\mathcal{B}_1)\geq\#(x,\mathcal{B}_2)$.

La primera condición es contradictoria con la que teníamos fijada, y la segunda es compatible en el caso de que $\#(x,\mathcal{B}_1)=\#(x,\mathcal{B}_2)$.

La situación anterior debe ser tenida en cuenta para que no coexista el elemento en $\mathcal{F}$ y $\mathcal{U}_{\mathcal{F}}$ del ufr-pomset resultante de la mezcla. Así, en la definición de *merge* se eliminarán los elementos dominantes de $\mathcal{F}$ en el ufr-pomset resultante que aparezcan en alguna de las indeterminaciones escogidas para el conjunto $\mathcal{U}_{\mathcal{F}}$.

En cuanto a la selección de las indeterminaciones dominantes, el proceso es similar al seguido en el Modelo 2, considerando a todas las indeterminaciones de un mismo ufr-pomset conjuntamente. Sólo en el caso de que el número total de indeterminaciones dominantes en ambos ufr-pomsets sean iguales, cabría preguntarse sobre si es más conveniente escoger indeterminaciones de $\mathcal{U}_{\mathcal{F}}$ o de $\mathcal{U}_{\mathcal{R}}$.

Observación 4.17:

Una indeterminación en $\mathcal{U}_{\mathcal{R}}$ es más restrictiva que si estuviera en $\mathcal{U}_{\mathcal{F}}$, en el sentido de que el elemento escogido no está ligado a que aparezca como primero en la secuencia. Por tanto, el número de posibles secuencias compatibles es menor que si la indeterminación fuera de $\mathcal{U}_{\mathcal{F}}$.

Ejemplo 4.17:

$$\mathcal{P}_1^{ufr} = <\{A,B,C\},\{A\},\varnothing,\varnothing,\{\{B,C\}\}> = A \rightarrow \{2\cdot B,C\} \oplus A \rightarrow \{B,2\cdot C\}$$

$$\mathcal{P}_2^{ufr} = <\{A,B,C\},\{A\},\varnothing,\{\{B,C\}\},\varnothing> =$$

$$A \rightarrow \{2\cdot B,C\} \oplus A \rightarrow \{B,2\cdot C\} \oplus B \rightarrow \{A,B,C\} \oplus C \rightarrow \{A,B,C\}$$

Como se observa, $\mathcal{P}_2^{ufr}$ engloba a $\mathcal{P}_1^{ufr}$, refrendando la observación anterior.

Ejemplo 4.18:

Sean $\mathcal{P}_1^{ufr} = <\{A,B,C\},\{A\},\varnothing,\varnothing,\{\{B,C\}\}>$ y $\mathcal{P}_2^{ufr} = <\{C,D\},\varnothing,\varnothing,\{\{C,D\}\},\varnothing>$. Teniendo en cuenta la Observación 4.3, el resultado de la mezcla de ambos ufr-pomsets resultaría:

$$\mathcal{P}_1^{ufr} \otimes \mathcal{P}_2^{ufr} = \left(A \rightarrow \{2\cdot B,C\} \oplus A \rightarrow \{B,2\cdot C\}\right) \otimes \left(C \rightarrow \{C,D\} \oplus D \rightarrow \{C,D\}\right) =$$

$$A \rightarrow \{2\cdot B,2\cdot C,D\} \oplus A \rightarrow \{2\cdot B,C,2\cdot D\} \oplus A \rightarrow \{B,2\cdot C,D\} \oplus A \rightarrow \{B,2\cdot C,2\cdot D\} \oplus$$

$$C \rightarrow \{A,2\cdot B,C,D\} \oplus C \rightarrow \{A,B,2\cdot C,D\} \oplus D \rightarrow \{A,2\cdot B,C,D\} \oplus D \rightarrow \{A,B,2\cdot C,D\}$$

Dado que las indeterminaciones de ambos ufr-pomsets son dominantes, así como el elemento de $\mathcal{F}$ del primero de ellos, y teniendo en cuenta que las dos indeterminaciones se solapan, el resultado podría venir dado tanto por el ufr-pomset $<\{A,B,C,D\},\{A\},\{C,D\},\varnothing,\{\{B,C\}\}>$ como por $<\{A,B,C,D\},\{A\},\varnothing,\{\{C,D\}\},\varnothing>$, ya que ambos engloban al conjunto de α-pomsets resultantes. Nótese que el primero contiene menos posibles secuencias al escoger la indeterminación de $\mathcal{U}_{\mathcal{R}}$, según se vio en la Observación 4.17. Además, no se rechazan como primeros los elementos de la indeterminación de $\mathcal{U}_{\mathcal{F}}$ para posibles futuras mezclas, al incluirse en $\mathcal{F}_m$, según se mostró que debía hacerse en la definición del Modelo 2 (véanse las conclusiones derivadas de la Observación 4.10).

Así pues, en el caso de que el número de indeterminaciones dominantes totales coincida para los dos ufr-pomsets involucrados en la mezcla, se escogerá aquel grupo que contenga el mayor número de indeterminaciones provenientes de $\mathcal{U}_{\mathcal{R}}$.

La definición de la operación *merge* correspondiente al Modelo 3 viene dada por el algoritmo de la Figura 4.27. La selección de las indeterminaciones dominantes viene dada a su vez por el algoritmo *selectUFR*, definido en la Figura 4.28.

La operación *ntrans* se define para el Modelo 3 como:

$$ntrans(\mathcal{P}^{ufr}) = \sum_{x \in \mathcal{B}} \#(x, \mathcal{B}) - 1 + \#\mathcal{U}_{\mathcal{F}} + \#\mathcal{U}_{\mathcal{R}} \tag{4.39}$$

siendo $\#\mathcal{U}_{\mathcal{F}} + \#\mathcal{U}_{\mathcal{R}}$ el número total de indeterminaciones en el ufr-pomset.

Algoritmo *merge* $(\mathcal{P}_1^{ufr}, \mathcal{P}_2^{ufr})$ **devuelve** $\mathcal{P}_3^{ufr}$

$\mathcal{B}_3 := \mathcal{B}_1 \cup \mathcal{B}_2$

$\mathcal{D}_{\mathcal{UR}1} := \mathcal{D}'_{\mathcal{U}}\left(\mathcal{U}_{\mathcal{R}1}, \mathcal{B}_1, \mathcal{B}_2\right)$

$\mathcal{D}_{\mathcal{UR}2} := \mathcal{D}'_{\mathcal{U}}\left(\mathcal{U}_{\mathcal{R}2}, \mathcal{B}_2, \mathcal{B}_1\right)$

$\mathcal{D}_{\mathcal{UF}1} := \mathcal{D}'_{\mathcal{U}}(\mathcal{U}_{\mathcal{F}1}, \mathcal{B}_1, \mathcal{B}_2)$

$\mathcal{D}_{\mathcal{UF}2} := \mathcal{D}'_{\mathcal{U}}(\mathcal{U}_{\mathcal{F}2}, \mathcal{B}_2, \mathcal{B}_1)$

$\mathcal{B}_{\mathcal{FDU}1} := \mathcal{B}_1 \uplus retainAll\left(\widetilde{\mathcal{D}_{\mathcal{UR}1}}, \mathcal{F}_1\right)$

$\mathcal{B}_{\mathcal{FDU}2} := \mathcal{B}_2 \uplus retainAll\left(\widetilde{\mathcal{D}_{\mathcal{UR}2}}, \mathcal{F}_2\right)$

$< \mathcal{U}_{\mathcal{F}3}, \mathcal{U}_{\mathcal{R}3} > := selectUFR(\mathcal{D}_{\mathcal{UF}1}, \mathcal{D}_{\mathcal{UR}1}, \mathcal{D}_{\mathcal{UF}2}, \mathcal{D}_{\mathcal{UR}2})$

$\mathcal{F}_3 := \left((\mathcal{F}_1 \cap \mathcal{F}_2) \cup \mathcal{D}'_{\mathcal{F}}(< \mathcal{B}_{\mathcal{FDU}1}, \mathcal{F}_1 >, \mathcal{P}_2) \cup \mathcal{D}'_{\mathcal{F}}(< \mathcal{B}_{\mathcal{FDU}2}, \mathcal{F}_2 >, \mathcal{P}_1)\right) - \widetilde{\mathcal{U}_{\mathcal{F}3}}$

$\mathcal{F}_{m3} := \left(\mathcal{F}_1 \cup \widetilde{\mathcal{U}_{\mathcal{F}1}} \cup \mathcal{F}_2 \cup \widetilde{\mathcal{U}_{\mathcal{F}2}}\right) - \left(\mathcal{F}_3 \cup \widetilde{\mathcal{U}_{\mathcal{F}3}}\right)$

si $\mathcal{F}_3 = \varnothing$ Y $\mathcal{U}_{\mathcal{F}3} = \varnothing$

$\mathcal{U}_{\mathcal{F}3} := \{\mathcal{F}_{m3}\}$

$\mathcal{F}_{m3} := \varnothing$

fsi

devolver $< \mathcal{B}_3, \mathcal{F}_3, \mathcal{F}_{m3}, \mathcal{U}_{\mathcal{F}3}, \mathcal{U}_{\mathcal{R}3} >$

Figura 4.27: Definición de *merge* para el *Modelo 3*.

Algoritmo *selectUFR* ($\mathcal{U}_{\mathcal{F}1}, \mathcal{U}_{\mathcal{R}1}, \mathcal{U}_{\mathcal{F}2}, \mathcal{U}_{\mathcal{R}2}$) **devuelve** $< \mathcal{U}_{\mathcal{F}3}, \mathcal{U}_{\mathcal{R}3} >$

$< \mathcal{U}_{\mathcal{F}3}, \mathcal{U}_{\mathcal{R}3} > := < \varnothing, \varnothing >$

para cada $\mathcal{S} \in \mathcal{U}_{\mathcal{F}1} \uplus \mathcal{U}_{\mathcal{R}1}$

si $\mathcal{S} \cap (\widetilde{\mathcal{U}_{\mathcal{F}2} \cup \mathcal{U}_{\mathcal{R}2}}) = \varnothing$

si $\mathcal{S} \in \mathcal{U}_{\mathcal{F}1}$

$\mathcal{U}_{\mathcal{F}3} := \mathcal{U}_{\mathcal{F}3} \cup \mathcal{S}$; $\mathcal{U}_{\mathcal{F}1} := \mathcal{U}_{\mathcal{F}1} - \{\mathcal{S}\}$

si no

$\mathcal{U}_{\mathcal{R}3} := \mathcal{U}_{\mathcal{R}3} \cup \mathcal{S}$; $\mathcal{U}_{\mathcal{R}1} := \mathcal{U}_{\mathcal{R}1} - \{\mathcal{S}\}$

fsi

fsi

fpara

para cada $\mathcal{S} \in \mathcal{U}_{\mathcal{F}2} \uplus \mathcal{U}_{\mathcal{R}2}$

si $\mathcal{S} \cap (\widetilde{\mathcal{U}_{\mathcal{F}1} \cup \mathcal{U}_{\mathcal{R}1}}) = \varnothing$

si $\mathcal{S} \in \mathcal{U}_{\mathcal{F}2}$

$\mathcal{U}_{\mathcal{F}3} := \mathcal{U}_{\mathcal{F}3} \cup \mathcal{S}$; $\mathcal{U}_{\mathcal{F}2} := \mathcal{U}_{\mathcal{F}2} - \{\mathcal{S}\}$

si no

$\mathcal{U}_{\mathcal{R}3} := \mathcal{U}_{\mathcal{R}3} \cup \mathcal{S}$; $\mathcal{U}_{\mathcal{R}2} := \mathcal{U}_{\mathcal{R}2} - \{\mathcal{S}\}$

fsi

fsi

fpara

si $\#\mathcal{U}_{\mathcal{F}1} + \#\mathcal{U}_{\mathcal{R}1} > \#\mathcal{U}_{\mathcal{F}2} + \#\mathcal{U}_{\mathcal{R}2}$

$\mathcal{U}_{\mathcal{F}3} := \mathcal{U}_{\mathcal{F}3} \cup \mathcal{U}_{\mathcal{F}1}$; $\mathcal{U}_{\mathcal{R}3} := \mathcal{U}_{\mathcal{R}3} \cup \mathcal{U}_{\mathcal{R}1}$

si no, si $\#\mathcal{U}_{\mathcal{F}1} + \#\mathcal{U}_{\mathcal{R}1} < \#\mathcal{U}_{\mathcal{F}2} + \#\mathcal{U}_{\mathcal{R}2}$

$\mathcal{U}_{\mathcal{F}3} := \mathcal{U}_{\mathcal{F}3} \cup \mathcal{U}_{\mathcal{F}2}$; $\mathcal{U}_{\mathcal{R}3} := \mathcal{U}_{\mathcal{R}3} \cup \mathcal{U}_{\mathcal{R}2}$

si no, si $\#\mathcal{U}_{\mathcal{R}1} \geq \#\mathcal{U}_{\mathcal{R}2}$

$\mathcal{U}_{\mathcal{F}3} := \mathcal{U}_{\mathcal{F}3} \cup \mathcal{U}_{\mathcal{F}1}$; $\mathcal{U}_{\mathcal{R}3} := \mathcal{U}_{\mathcal{R}3} \cup \mathcal{U}_{\mathcal{R}1}$

si no

$\mathcal{U}_{\mathcal{F}3} := \mathcal{U}_{\mathcal{F}3} \cup \mathcal{U}_{\mathcal{F}2}$; $\mathcal{U}_{\mathcal{R}3} := \mathcal{U}_{\mathcal{R}3} \cup \mathcal{U}_{\mathcal{R}2}$

fsi

devolver $< \mathcal{U}_{\mathcal{F}3}, \mathcal{U}_{\mathcal{R}3} >$

Figura 4.28: Selección de indeterminaciones dominantes en el *Modelo 3*.

MODELO 4 (dos herramientas)

Observación 4.18:

> Es útil diferenciar el caso en que sólo se dispone de dos herramientas en una determinada máquina. En tal situación el análisis resulta más sencillo. Sólo podría generarse una indeterminación, forzosamente asociada al conjunto de primeros. Esto se debe a que la aplicación de *push* no podrá trasladarla a $\mathcal{U}_{\mathcal{R}}$, ya que el elemento que se intenta añadir es uno de los que aparecen en la indeterminación, lo que traería el efecto de resolver la misma en favor de tal elemento, haciendo desaparecer la indeterminación.

El pomset puede definirse mediante un multiconjunto $\mathcal{B}$ y el conjunto $\mathcal{F}$ de primeros elementos posibles en las secuencias óptimas. Para representar a la única indeterminación que puede darse no hace falta ningún elemento adicional. Basta indicar que se producirá si y sólo si el conjunto de primeros ha quedado vacío. La operación *push* viene dada por la ecuación

$$push(x,<\mathcal{B},\mathcal{F}>) \equiv \begin{cases} <\mathcal{B},\{x\}> & \text{si } x \in \mathcal{F} \\ <\mathcal{B} \uplus \{x\},\{x\}> & \text{si } x \notin \mathcal{F} \end{cases} \tag{4.40}$$

Nótese que el caso correspondiente a la indeterminación se resuelve agregando un ejemplar del elemento que se añade, desapareciendo así la indeterminación. Justamente, el caso en que el elemento situado en $\mathcal{F}$ es el contrario al que se añade tiene el mismo efecto. Como la indeterminación se ha representado mediante $\mathcal{F}=\varnothing$, ambos casos tienen la misma definición.

La definición de *merge* viene dada por la ecuación

$$merge(<\mathcal{B}_1,\mathcal{F}_1>,<\mathcal{B}_2,\mathcal{F}_2>) \equiv <\mathcal{B}_1 \cup \mathcal{B}_2, \mathcal{D}_{\mathcal{F}}(\mathcal{P}_1,\mathcal{P}_2)> \tag{4.41}$$

donde $\mathcal{D}_{\mathcal{F}}(\mathcal{P}_1,\mathcal{P}_2)$ viene definido por las ecuaciones (4.31) y (4.32).

Para la definición de *ntrans* hay que observar la posible indeterminación, que se da cuando $\mathcal{F}$ es $\varnothing$. Hay que distinguir el caso en que el propio pomset sea vacío, ya que entonces, aunque $\mathcal{F}$ es también $\varnothing$, no existe indeterminación. Así pues, *ntrans* viene definido por

$$ntrans(<\mathcal{B},\mathcal{F}>) \equiv \begin{cases} \#\mathcal{B} & \text{si } \mathcal{F} \neq \varnothing \text{ ó } \mathcal{B} = \varnothing \\ \#\mathcal{B}+1 & \text{si } \mathcal{F} = \varnothing \text{ y } \mathcal{B} \neq \varnothing \end{cases} \tag{4.42}$$

Es de resaltar que en este modelo, aparte de resultar más sencillo, no se efectúan simplificaciones adicionales debido a las consideraciones realizadas en los Modelos 2 y

3. Además, debido a que se considera la posible indeterminación, el modelo es más exacto que el Modelo 1.

4.3.4 Heurística h_4: modificación de h_1 usando h_2 y h_3

Una vez que disponemos de las heurísticas h_2 y h_3 correspondientes a estimaciones optimistas de los usos de las herramientas en cada máquina para cada subárbol del árbol de precedencias, cabe preguntarse cómo mezclar esas estimaciones con las correspondientes a la heurística h_1, donde se despreció precisamente el uso compartido de los recursos.

Como se ha visto en los subapartados anteriores, las heurísticas h_2 y h_3 se definieron asociadas al nodo de expansión y no a cada una de las tareas particulares que participaban en el mismo como candidatas. Esto se debe a que el interés estaba en sumar los tiempos de uso de los recursos, sin atender a las relaciones de precedencia de las tareas. Sin embargo, se pueden definir las mismas heurísticas para cada tarea, y calcularlas previo a la ejecución del algoritmo A*, mediante un recorrido en profundidad del árbol de precedencias, tal como se hizo para la heurística h_1. De esta forma, h_2 viene definida por la expresión

$$h_2(T) = \max_{máquinas}\left(h_2(T,M_i)\right) = \max_{máquinas}\left(\left(\sum_{H_j \in M_i} h_2(T,H_j)\right) + \Sigma\left(T,\Delta_{cht}(M_i)\right)\right) \quad (4.43)$$

Cuando la duración de los cambios de herramientas no dependía de cuáles eran las herramientas involucradas, podía usarse la heurística h_3 para obtener una estimación más exacta del tiempo mínimo necesario para los cambios de herramientas, $\Sigma\left(T,\Delta_{cht}\left(M_i\right)\right)$. De esta forma, siendo $h_3(T, M)$ definida como

$$h_3(T,M) = \left(\sum_{H_j \in M} h_2(T,H_j)\right) + ntrans\left(\mathcal{P}\left(T,M\right)\right)\cdot\Delta_{cht}\left(M\right) \quad (4.44)$$

la heurística h_3 puede definirse para una tarea T como

$$h_3(T) = \max_{máquinas}\left(h_3(T,M_i)\right) \quad (4.45)$$

Dado que h_2 y h_3 podrían ser estimaciones mejores que h_1, siendo todas ellas optimistas, las primeras pueden tenerse en cuenta en el cálculo de la última, ya que en la expresión recursiva que define a ésta última lo que se requiere es una estimación del tiempo necesario para ejecutar todo el subárbol de tareas. La nueva heurística que se obtiene, h_4, se define basándonos en la ecuación (4.16), sustituyendo h_1 por h_4 y considerando h_2 o h_3, según lo indicado antes. Así, la mejor estimación debido a h_2 o h_3 se propaga hacia las tareas predecesoras en el árbol de precedencias. De esta forma, $h_4(T)$ viene dada por una de las siguientes expresiones, según se use h_2 o h_3 en su definición:

$$h_4(T) = \max\left(dur(T) + \max_{T_i \in suc(T)} \left(h_4(T_i) + \tau_{mov}(T_i, T)\right), h_2(T)\right) \tag{4.46}$$

o

$$h_4(T) = \max\left(dur(T) + \max_{T_i \in suc(T)} \left(h_4(T_i) + \tau_{mov}(T_i, T)\right), h_3(T)\right) \tag{4.47}$$

Nótese que estas heurísticas no anulan a $h_2(n)$ y a $h_3(n)$, ya que éstas consideran a todas las tareas candidatas del nodo de expansión. La definición de $h_4(n)$ sería similar a la de $h_1(n)$ (ecuación (4.11)), y aquí sí se puede decir que $h_4(n)$ es una heurística mejor informada que $h_1(n)$.

4.3.5 Heurística h_5: estimación de los intervalos de tiempos de uso de las máquinas

En la heurística anterior, el tiempo de uso de la máquina más desfavorable, dado por $h_2(T)$ y $h_3(T)$, se supone que puede transcurrir durante la ejecución de todas las tareas del subárbol de precedencias cuya raíz es la tarea T. En algunos casos se podrá determinar, analizando el propio árbol de precedencias, que el intervalo de tiempo de uso es menor, con lo que la estimación del tiempo total para la ejecución de todo el árbol de tareas se podrá mejorar aún más. Definiremos dos nuevas cantidades, δ_b y δ_e, que indicarán los intervalos, al principio y al final de la ejecución de todas las tareas del árbol, en que la máquina no puede ser usada (ver Figura 4.29).

El cálculo de δ_b se realiza atendiendo a su definición recursiva:

$$\delta_b(T, M) = \begin{cases} 0 & \text{si } M = M(T) \\ dur(T) + \min_{T_i \in suc(T)} \left(\delta_b(T_i, M) + \max\left(\Delta_{mov}(\cdot), \Delta'_{cht}(\cdot)\right)\right) & \text{si } M \neq M(T) \end{cases} \tag{4.48}$$

donde $\Delta_{mov}(\cdot) \equiv \Delta_{mov}\left(sa(T_i), M(T_i), M(T)\right)$ y $\Delta'_{cht}(\cdot) \equiv \Delta'_{cht}(T_i, T)$, representando esto último el posible cambio de herramienta necesario entre la ejecución de T_i y T, estando

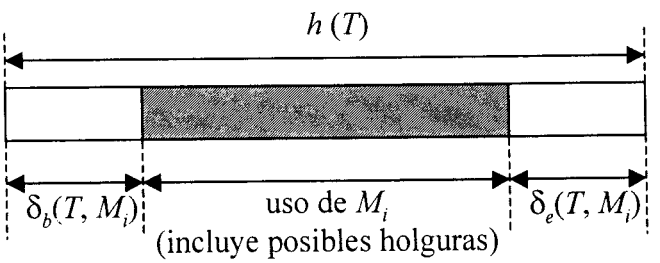


Figura 4.29: Intervalos de uso y no uso de la máquina M_i en la estimación de $h(T)$.

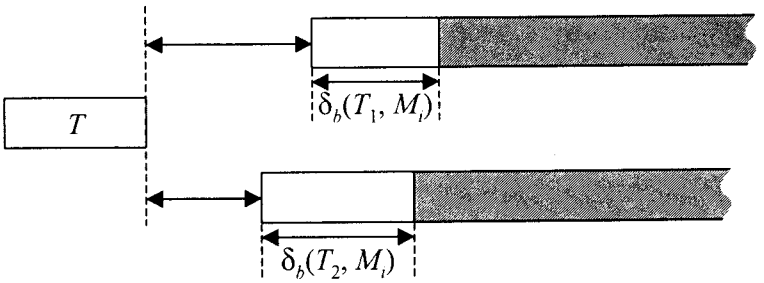


Figura 4.30: Cálculo de $\delta_b(T,M)$.

definido por

$$\Delta'_{cht}(T_i,T)=\begin{cases}\Delta_{cht}\left(M(T),H(T_i),H(T)\right) & \text{si } M(T)=M(T_i)\\ 0 & \text{si } M(T)\neq M(T_i)\end{cases} \tag{4.49}$$

En la Figura 4.30 se muestra el sentido de la definición de $\delta_b(T,M)$ para $M \neq M(T)$. Para que dicha definición sea consistente, se tomará $\delta_b(T,M)=\infty$ si $M \neq M(T)$ y $suc(T)=\varnothing$. De esta forma, cuando una máquina no sea usada por ninguna de las tareas pertenecientes a un subárbol de precedencias, el valor correspondiente a δ_b será infinito, y ello ocurrirá sólo en ese caso.

De la misma forma, δ_e se calcula mediante su definición recursiva:

$$\delta_e(T,M)=\begin{cases}\min\limits_{T_i\in suc(T)}\left(\delta_e(T_i,M),h_5(T)-dur(T)\right) & \text{si } M=M(T)\\ \min\limits_{T_i\in suc(T)}\left(\delta_e(T_i,M)\right) & \text{si } M\neq M(T)\end{cases} \tag{4.50}$$

Nuevamente, para que dicha definición sea consistente, se tomará $\delta_e(T,M)=\infty$ si $M \neq M(T)$ y $suc(T)=\varnothing$. Nótese que, cuando $M=M(T)$, si la máquina es usada por alguna de las tareas sucesoras, $h_5(T)-dur(T)$ será mayor que el δ_e correspondiente, prevaleciendo por tanto éste último. Sólo en el caso de que ninguna de las tareas sucesoras utilice la máquina M, el valor que se toma es $h_5(T)-dur(T)$. En la Figura 4.31 se muestra tal situación.

La definición de la nueva heurística $h_5(T)$ tiene la misma estructura que $h_4(T)$, sólo que en lugar de usar $h_2(T)$ o $h_3(T)$ se tienen en cuenta los incrementos asociados a δ_b y δ_e. Además, es preciso separar las estimaciones de $h_2(T)$ o $h_3(T)$ para cada máquina, usando pues $h_2(T, M)$ o $h_3(T, M)$, resultando:

$$\begin{aligned}h_5(T)=\max\Big(&dur(T)+\max_{T_i\in suc(T)}\left(h_5(T_i)+\tau_{mov}(T_i,T)\right),\\ &\max_{h_2(T,M_i)\neq 0}\left(h_2(T,M_i)+\delta_b(T,M_i)+\delta_e(T,M_i)\right)\Big)\end{aligned} \tag{4.51}$$

o

$$h_5(T) = \max\Big(dur(T) + \max_{T_i \in suc(T)} \big(h_5(T_i) + \tau_{mov}(T_i, T)\big), \max_{h_3(T,M_i)\neq 0} \big(h_3(T,M_i) + \delta_h(T,M_i) + \delta_e(T,M_i)\big)\Big) \quad (4.52)$$

La circularidad que se observa en las definiciones correspondientes a las ecuaciones (4.50), (4.51) y (4.52) se resuelve de la siguiente forma. Como se indicó antes, el término $h_5(T) - dur(T)$ será tomado en $\delta_e(T, M_i)$ sólo cuando la máquina M_i no es usada por ninguna de las tareas sucesoras de T. En tal caso, el término $h_3(T, M_i) + \delta_h(T, M_i) + \delta_e(T, M_i)$ no puede ser el dominante en el cálculo de $h_5(T)$, ya que $h_3(T, M_i) = dur(T)$ y $\delta_h(T, M_i) = 0$. Sólo será pues necesario separar este caso, y no habrá que realizar ajustes posteriores.

Al igual que se indicó para la heurística h_4, la nueva heurística h_5 representa una mejor estimación para $h(T)$ que las definidas con anterioridad. No puede decirse lo mismo al aplicarse a $h(n)$, ya que la forma en que son tenidas en cuenta las estimaciones para cada tarea es similar a la que se indicó para $h_1(n)$, es decir, la estimación total es la correspondiente a la tarea con mayor estimación, contemplando las holguras respecto a $g(n)$. Las heurísticas $h_2(n)$ y $h_3(n)$ pueden ser más realistas, ya que la estimación total tiene un carácter aditivo, por lo que, aunque las estimaciones para las tareas fueran menores que para h_5, en su conjunto podrían superar a la realizada por ésta.

4.3.6 Heurística h_6: consideración de todas las máquinas en h_1, h_4 y h_5

La ecuación (4.11) fue usada para definir $h_1(n)$, y también sirve para definir las correspondientes $h_4(n)$ y $h_5(n)$. Puede observarse cómo en el cálculo de $f(n)$ no se tiene en cuenta el uso de las diferentes máquinas tanto en $g(n)$ como en $h(n)$. En la Figura 4.32 se aprecia el efecto del uso de una máquina distinta a la usada por la tarea candidata considerada. Para poder obtener esta estimación hay que definir una nueva cantidad asociada al retardo máximo que puede tenerse en el uso de cada máquina. Teniendo en cuenta que hay que enlazar con la última herramienta usada en el estado correspondien-

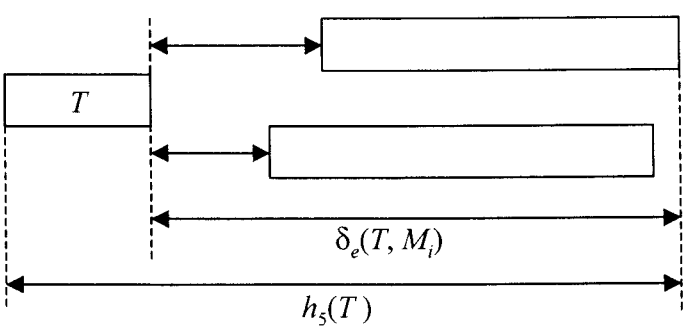


Figura 4.31: Cálculo de $\delta_e(T,M)$.

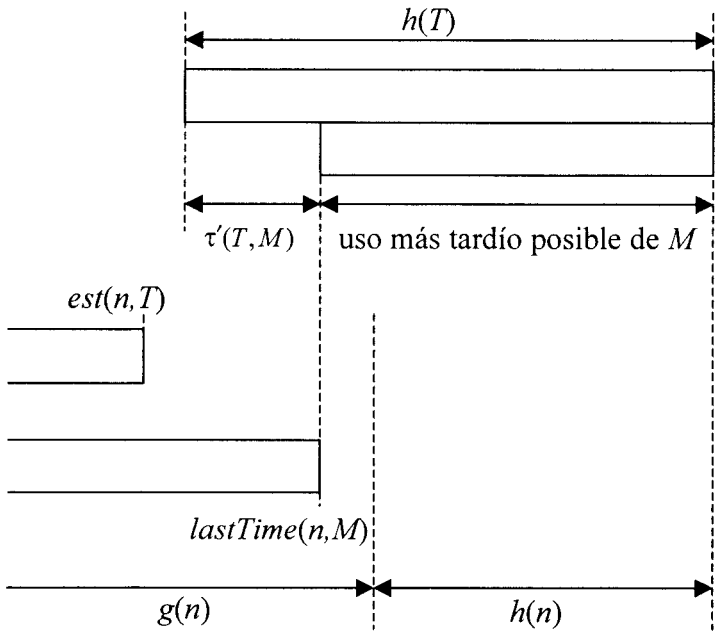


Figura 4.32: Ilustración de $h_6(n)$.

te al nodo de expansión n, la medida concreta que se necesita estará asociada al uso de cada herramienta.

Si observamos la definición que se hizo para $\tau(T, M, H)$ en la ecuación (4.14), cuando $M \neq M(T)$ el retardo no podía ser negativo, por lo que se tomó el valor cero cuando los cálculos asociados a las tareas sucesoras daban un valor negativo. Eso es correcto porque lo que se buscaba era un retardo adicional que podía servir para una mejor estimación de $h(T)$. Las expresiones que se usaban en aquella definición son las que necesitamos ahora. Para la nueva medida, $\tau'(T, M, H)$, permitiremos ahora valores negativos, indicando en tal caso que sobra tiempo para un eventual cambio de herramienta que pudiera necesitarse, o simplemente que, en el uso más tardío posible de la máquina en cuestión (que no altere la estimación realizada), se necesitará tener instalada la herramienta con posterioridad al comienzo de la tarea.

Según lo expresado antes, la definición de $\tau'(T, M, H)$ viene dada por

$$\tau'(T,M,H) = \begin{cases} \Delta_{cht}(M, H(T), H) & \text{si } M = M(T) \\ \max\limits_{T_i \in suc(T)} \left(h(T_i) + \tau'(T_i, M(T), H(T)) - h(T) \right) & \text{si } M \neq M(T) \end{cases} \quad (4.53)$$

e indica el tiempo antes de la ejecución de la tarea T en el que debe estar instalada la herramienta H en la máquina M para que la ejecución de T y todas sus sucesoras en el árbol de precedencias no terminen después de $h(T)$. La Figura 4.33 ilustra el sentido de la definición anterior. Nótese que los valores negativos de τ' se muestran con el sentido de la flecha hacia la derecha y los valores positivos hacia la izquierda.

La nueva heurística $h_6(n)$, basada en h_1 y τ', se define como

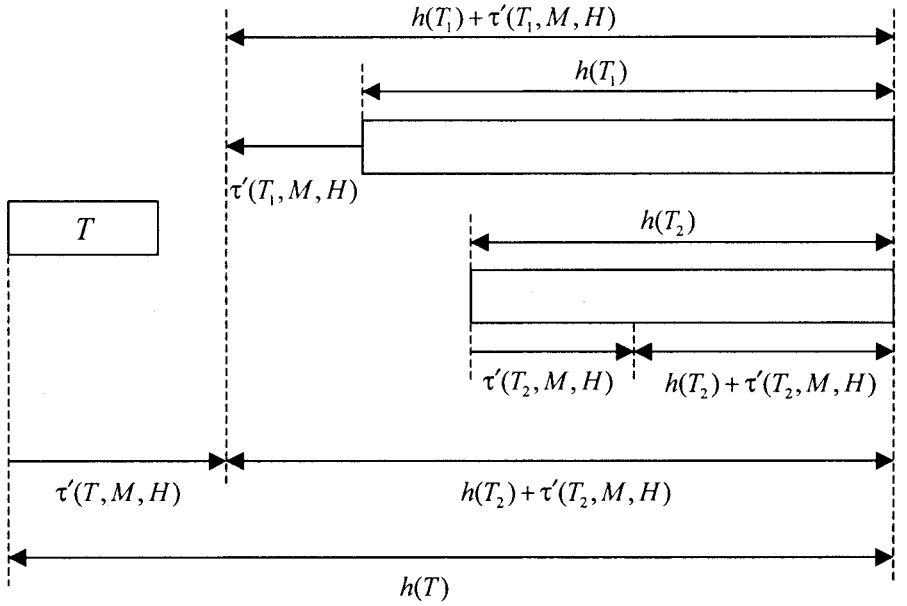


Figura 4.33: Cálculo de τ'(*T*, *M*, *H*) .

$$h_6(n)=\max_{T_i\in cand(n)}\left(\max_{\text{maquinas}}\left(h(T_i)+\tau'\left(T_i,M_j,lastTool\left(n,M_j\right)\right)-e\left(n,M_j\right)\right)\right) \tag{4.54}$$

En la ecuación anterior, $h(T)$ puede venir dada por cualquiera de las heurísticas h_1, h_4 o h_5. Nótese que, en caso de aplicarla sobre h_1, la expresión equivale a

$$h_6(n)=\max\left(h_1(n),\max_{T_i\in cand(n)}\left(\max_{M_j\neq M(T_i)}\left(h_1(T_i)+\tau'\left(T_i,M_j,lastTool\left(n,M_j\right)\right)-e\left(n,M_j\right)\right)\right)\right) \tag{4.55}$$

Las mismas ideas que se usaron para definir tales heurísticas podrían utilizarse para dar una mejor estimación para τ' para el caso $M\neq M(T)$, manteniendo la misma cuando $M = M(T)$. De esta forma, la ecuación (4.54) sigue siendo válida, y lo que cambiaría es la definición de τ'. Los distintos casos se detallan a continuación.

En el caso de la heurística $h_4(T)$ usando $h_2(T)$, definida en la ecuación (4.46), τ'(*T*, *M*, *H*) puede definirse, cuando $M\neq M(T)$, como

$$\begin{aligned}\tau'(T,M,H)=\max\Big(&\max_{T_i\in suc(T)}\left(h_4(T_i)+\tau'\left(T_i,M(T),H(T)\right)-h_4(T)\right),\\ &h_2(T)+\Delta''_{chl}(T,M,H)-h_4(T)\Big)\end{aligned} \tag{4.56}$$

siendo

$$\Delta''_{cht}(T,M,H)=\begin{cases}0 & \text{si } h_2(T,H)>0\\ \min\limits_{h_2(T,H_j)>0}\left(\Delta_{cht}(M,H,H_j)\right) & \text{si } h_2(T,H)=0\end{cases} \tag{4.57}$$

La expresión anterior muestra que será necesario un cambio de herramienta si H ya no va a ser usada.

Si en $h_4(T)$ se usa $h_3(T)$, ecuación (4.47), $\tau'(T, M, H)$ puede definirse, cuando $M \neq M(T)$, como

$$\begin{aligned}\tau'(T,M,H)=\max\Big(&\max_{T_i\in suc(T)}\left(h_4(T_i)+\tau'(T_i,M(T),H(T))-h_4(T)\right),\\ &h_3(T)+\Delta'''_{cht}(T,M,H)-h_4(T)\Big)\end{aligned} \tag{4.58}$$

siendo

$$\Delta'''_{cht}(T,M,H)=\begin{cases}0 & \text{si } H\in\mathcal{P}(T,M).\mathcal{F}\\ \Delta_{cht}(M) & \text{si } H\notin\mathcal{P}(T,M).\mathcal{F}\end{cases} \tag{4.59}$$

La expresión anterior indica que si H no pertenece al conjunto de primeros del pomset asociado al subárbol de precedencias para la máquina en cuestión, será necesario un cambio de herramientas adicional. Recuérdese que en este caso se había realizado la simplificación adicional de considerar la duración de los cambios de herramientas independientes de las propias herramientas involucradas.

En el caso de la heurística $h_5(T)$ usando $h_2(T)$, definida en la ecuación (4.51), $\tau'(T, M, H)$ puede definirse, cuando $M \neq M(T)$, como

$$\begin{aligned}\tau'(T,M,H)=\max\Big(&\max_{T_i\in suc(T)}\left(h_5(T_i)+\tau'(T_i,M(T),H(T))-h_5(T)\right),\\ &h_2(T)+\delta_e(T,M)+\Delta''_{cht}(T,M,H)-h_5(T)\Big)\end{aligned} \tag{4.60}$$

Por último, si en $h_5(T)$ se usa $h_3(T)$, ecuación (4.52), $\tau'(T, M, H)$ puede definirse, cuando $M \neq M(T)$, como

$$\begin{aligned}\tau'(T,M,H)=\max\Big(&\max_{T_i\in suc(T)}\left(h_5(T_i)+\tau'(T_i,M(T),H(T))-h_5(T)\right),\\ &h_3(T)+\delta_e(T,M)+\Delta'''_{cht}(T,M,H)-h_5(T)\Big)\end{aligned} \tag{4.61}$$

4.4 Detección de simetrías

La eficiencia en la resolución de este tipo de problemas puede mejorarse bastante si se detectan las simetrías intrínsecas al mismo. Una parte importante del espacio de búsqueda podría ser ignorada sin perder soluciones. Si se apura aún más, parte del espacio de búsqueda ignorado podría corresponderse con soluciones que con certeza puede asegurarse que no son óptimas.

En el caso que nos ocupa, las simetrías estarían ligadas a la decisión de cómo expandir un nodo de expansión en el algoritmo *paralelo-A**. Sin atender a posibles simetrías, se generaría un nodo sucesor por cada tarea candidata presente en el nodo. Sin embargo, es fácil ver que cuando dos tareas candidatas usen recursos distintos, en principio podría dar igual el orden de incorporación de ambas tareas en el estado correspondiente a la solución, dado que en cada paso de expansión sólo incorporamos una nueva tarea. Esto puede suponer no tener que generar nodos sucesores correspondientes a todas las tareas candidatas. En la generación de todas las soluciones posibles, sin que se pierda el óptimo, habrá que garantizar que las tareas sucesoras a las candidatas no se ven implicadas en un posible retraso de su comienzo.

Para llevar a cabo la mejora asociada a lo anterior, se va a definir el concepto de *compatibilidad de tareas*, de modo que cuando dos tareas sean compatibles, sólo será necesario generar el nodo sucesor para una de ellas.

Definición:

> Dado un nodo de expansión n, cuyo estado queda definido por las tareas candidatas a ser introducidas en el siguiente paso de expansión en la solución, se dice que una tarea candidata T_1 es *compatible con* otra tarea candidata T_2 si la introducción de T_1 en la solución no produce retraso en el tiempo de comienzo más temprano de T_2 ni de las tareas sucesoras de ésta en el árbol de precedencias. Utilizaremos el predicado *compat* (T_1, T_2) para expresar lo anterior, de forma que se cumplirá si ocurre lo anterior. La propiedad de compatibilidad no es simétrica: puede ocurrir que la introducción de J1 no produzca retrasos en J2 ni en sus sucesoras, pero sí a la inversa. En tal caso, *compat* (T_1, T_2) sería *cierto*, pero *compat* (T_2, T_1) sería *falso*.

Para la evaluación de esta propiedad han de tenerse en cuenta los tiempos de comienzo más temprano de las tareas involucradas y de sus sucesoras, los recursos que necesitan y la última herramienta usada en cada máquina, y tiempo de su último uso para el estado correspondiente al nodo de expansión.

En lugar de procesar toda esta información en línea para determinar las tareas compatibles, sería deseable contar con información útil calculada previamente, de manera que pudieran minimizarse al máximo los cálculos durante la fase expansiva del

algoritmo. Hay que tener en cuenta que en muchos casos habría que realizar búsquedas en el árbol de precedencia similares a las realizadas para el cálculo de compatibilidad en otros nodos.

La información necesaria es muy parecida a la que se ha utilizado para definir alguna de las heurísticas recogidas en el apartado anterior. De forma similar a como se hizo entonces, se define $\tau_c(T, M, H)$ como el instante, relativo al comienzo de T, a partir del cual si la herramienta H está situada en la máquina M, se retrasaría alguna de las tareas del árbol de precedencias de T, con respecto a su comienzo más temprano. La definición viene dada por la ecuación:

$$\tau_c(T,M,H)=\begin{cases}\Delta_{cht}(M,H,H(T)) & \text{si } M=M(T)\\ \max\limits_{T_i\in suc(T)}\left(\tau_c(T_i,M,H)-\max\left(\Delta_{mov}(\cdot),\Delta'_{cht}(\cdot)\right)\right)-dur(T) & \text{si } M\neq M(T)\end{cases} \quad (4.62)$$

donde $\Delta_{mov}(\cdot)\equiv\Delta_{mov}\left(sa(T_i),M(T_i),M(T)\right)$ y $\Delta'_{cht}(\cdot)\equiv\Delta'_{cht}(T_i,T)$, representando esto último el posible cambio de herramienta necesario entre la ejecución de T_i y T, definido en la ecuación (4.49). La Figura 4.34 ilustra la definición anterior. Los valores positivos de τ_c se corresponden con las flechas hacia la izquierda, y los negativos hacia la derecha. Nótese que en el ejemplo de la figura, al usar las dos tareas sucesoras inmediatas a T la misma máquina pero distinta herramienta, los valores de τ_c para T y la máquina usada por ambas no se corresponde con el comienzo de las tareas. Si alguna de las herramientas estuviera situada con posterioridad a lo que indica τ_c en M_1, entonces la tarea que usa la otra herramienta se vería forzosamente retrasada en su comienzo más temprano.

La determinación de la compatibilidad de tareas se realizaría de la siguiente forma:

Sean T_i, $T_j \in cand\,(n)$, y $est(n, T_i)$ y $est(n, T_j)$ sus tiempos de comienzo más tem-

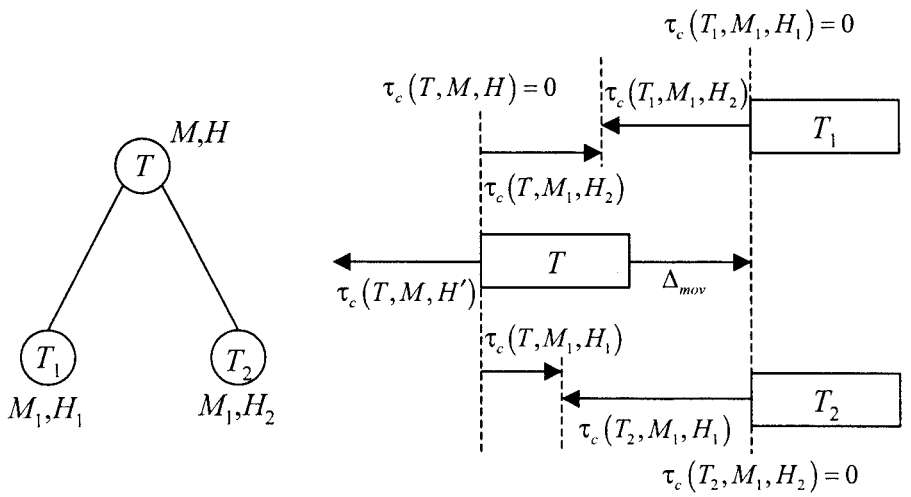


Figura 4.34: Cálculo de $\tau_c(T, M, H)$.

prano. Los tiempos de comienzo más temprano de T_j y sus sucesoras en el árbol de precedencias no se ven retrasados por la elección de T_i si y sólo si

$$est(n,T_i)+dur(T_i)\leq est(n,T_j)-\tau_c(T_j,M(T_i),H(T_i)) \tag{4.63}$$

siendo entonces *compat* (T_i, T_j) = *cierto*, y *falso* en caso contrario.

El sentido de esta propiedad es el de minimizar el número de ramas en la expansión de nodos, de manera que no sea necesario expandir todos los hijos de un nodo por resultar equivalente hacerlo en sucesivos pasos de expansión.

Cuando una tarea es compatible con todas las demás candidatas, puede expandirse un sólo hijo, conteniendo en su estado el anterior más la introducción de esa tarea. Con esto no se pierde ninguna solución mejor que la mejor que se obtendría a partir del nuevo nodo expandido. Si además la propiedad se cumpliera de forma simétrica con todas las tareas, no se estaría perdiendo ninguna solución, incluso peor.

Cuando no haya ninguna tarea compatible con todas las demás candidatas, se necesitará expandir varios nodos para no perder posibles soluciones óptimas. Sería deseable determinar cuáles de las tareas forman el subconjunto mínimo de nodos de expansión necesarios para garantizar que la solución óptima no se pierde. Más deseable aún sería disponer de un criterio que garantice que el número total de nodos generados fuera mínimo. Nos encontramos con varias alternativas contrapuestas:

1. Escoger la tarea con el menor número de tareas incompatibles. De esta forma, el árbol de expansión tiende a crecer estrecho, por lo que es esperable que el número de nodos se acerque más al número de soluciones posibles.

2. Escoger la tarea *crítica*, es decir, la que está involucrada en el valor calculado para la función de evaluación $f(n)$. De esta forma, se favorece que se determinen cuanto antes las ramas que no llevan a la solución óptima. Es de esperar que tenga un mejor comportamiento cuanto mejor sea la heurística utilizada para la estimación.

3. Otro criterio posible es escoger las tareas según sus tiempos de comienzo más temprano, que normalmente tendrán más posibilidad de ser compatibles con más tareas.

En cualquier caso, una vez escogida una tarea T para su expansión, también deberán ser consideradas las tareas con las que no es compatible la elegida, es decir, las tareas T_i que cumplen *compat* (T, T_i) = *falso*. Para ello no será necesario expandir un nodo por cada una de esas tareas, sino que puede volver a repetirse el proceso de selección al igual que se hizo antes, pero restringiendo el conjunto de tareas a las que cumplen lo anterior.

En la Figura 4.35 se muestra el algoritmo de expansión de nodos, utilizado en el algoritmo *paralelo-A**, usando el tercero de los criterios indicados más arriba.

La consideración de la propiedad de compatibilidad de tareas tiene especial importancia si el algoritmo A* trabaja sobre un árbol de expansión y no sobre un grafo. Es fácil que así ocurra debido a que la búsqueda del nodo generado en la expansión puede ser costosa, ya que se debe comparar el estado completo correspondiente al nodo, por lo que puede compensar el uso del árbol de expansión frente al grafo.

Una alternativa a la consideración de la compatibilidad entre tareas pasaría por

```
Algoritmo expandir (n)
   expandir (n, cand(n))
fin

Algoritmo expandir (n, C )
   estMin := min_{T_i ∈ cand(n)} (est(n,T_i))
   si sólo hay una tarea, T_i ∈ C, con est(n,T_i) = estMin
      incluir en ABIERTOS un nodo que incorpore T_i al estado de n
      sea C' = {T_j ∈ C, tales que compat(T_i,T_j) = falso}
      si C' ≠ ∅
         expandir (n, C')
      fsi
   si no
      sea nTarInc_i el número de tareas T_j ∈ C tales que compat(T_i,T_j) = falso
         para cada tarea T_i ∈ C, con est(n,T_i) = estMin, y sea nTarIncMin el
         valor mínimo de esas cantidades
      sea T_i ∈ C, con est(n,T_i) = estMin y nTarInc_i = nTarIncMin
      incluir en ABIERTOS un nodo que incorpore T_i al estado de n
      sea C' = {T_j ∈ C, tales que compat(T_i,T_j) = falso}
      si C' ≠ ∅
         expandir (n, C')
      fsi
   fsi
fin
```

Figura 4.35: Algoritmo para la expansión de nodos.

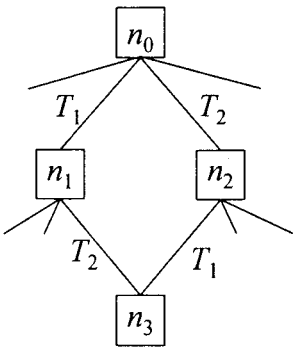


Figura 4.36: Detección de nodos equivalentes.

tener en cuenta todos los nodos, es decir, no se podaría el árbol de expansión por estos motivos, con lo que el algoritmo podría dar muchos pasos en falso durante la búsqueda de la solución óptima. Lo anterior podría ocurrir por dos motivos: por no detectar los nodos equivalentes o por visitar nodos que no conducirán a una solución mejor que la óptima. La expansión, sin embargo, sería más simple.

La detección de nodos equivalentes se puede hacer de forma fácil y eficiente, sin necesidad de realizar una búsqueda exhaustiva entre los nodos correspondientes al mismo nivel. La cuestión clave es que cuando dos tareas candidatas en un nodo no son directamente incompatibles (sin considerar sus tareas sucesoras) en ambos sentidos, en el siguiente nivel de expansión habrá dos nodos equivalentes, fruto de haber incluido en el plan original las dos tareas en los dos órdenes posibles. Lo anterior siempre ocurrirá si los recursos usados por ambas tareas son diferentes.

El proceso para detectar nodos equivalentes se realiza entonces de la siguiente forma (ver Figura 4.36): generados ya los nodos n_0, n_1 y n_3 entre otros, se está expandiendo n_2. En el cálculo de sus sucesores se contempla la tarea candidata T_1. Si T_1 no tiene recursos comunes con T_2, a través de la cual se ha llegado a n_2 desde n_0, entonces el nodo resultante de expandir n_2 considerando T_1 es equivalente al que haya podido generarse desde n_1 (sucesor de n_0 considerando T_1) considerando T_2.

Quedaría por asegurar si con este procedimiento no quedan nodos equivalentes por emparejar.

4.5 Otras mejoras en el algoritmo

En el apartado anterior se vio cómo la relación de compatibilidad de tareas asociada a la detección de simetrías podía usarse en la elección de diferentes estrategias de búsqueda, es decir, en la selección de los nodos sucesores que debían generarse en la expansión, y por consiguiente, ignorando los caminos hacia la solución desde los nodos no escogidos.

Un aspecto desfavorable de los algoritmos A* es el consumo de memoria. Existen variantes del algoritmo A* que mejoran tal aspecto. El algoritmo A* por profundi-

zación iterativa, IDA* [Korf, 1985], fue el primer algoritmo de búsqueda heurística, limitado por memoria, utilizado ampliamente. IDA* es un algoritmo completo y óptimo, con las mismas ventajas e inconvenientes de la búsqueda A*. Sin embargo, puesto que la búsqueda es preferente por profundidad, sólo necesita una cantidad de memoria proporcional a la longitud de la ruta más larga que explore. En [Russell y Norvig, 1996] se indican referencias a otras variantes.

La modificación que se ha usado en este trabajo se basa en el objetivo de descartar lo antes posible a aquellos nodos que no iban a ser visitados por el algoritmo A*, por resultar el valor de la función de evaluación mayor que el de la solución óptima real del problema. De esta forma, se mantiene básicamente el esquema del algoritmo A*, realizándose una búsqueda en profundidad cada cierto tiempo, con el fin de obtener una solución, que pudiera mejorar a la mejor que hubiera hasta el momento. El valor de esa mejor solución encontrada sirve para descartar directamente a aquellos nodos con una estimación que no mejore al de la mejor solución encontrada, y a eliminar los que pudieran estar presentes en la lista de abiertos.

Nótese que el número de nodos visitados no disminuye con la modificación propuesta, sino que más bien podría aumentar. Esto es debido a que, en el recorrido en profundidad, el valor usado para visitar un nodo es el valor correspondiente a la mejor solución encontrada hasta el momento, que no necesariamente tiene por qué coincidir con el óptimo. Así pues, y sólo en la parte correspondiente al recorrido en profundidad, se podrían visitar nodos con un valor de la función objetivo superior al de la solución óptima, que el algoritmo A* no hubiera considerado. A cambio, estamos mejorando el comportamiento del algoritmo en cuanto al consumo de memoria.

El número de nodos adicionales visitados aumentará con el número de hojas visitadas, o lo que es lo mismo, con el número de veces que se realiza el recorrido en profundidad en busca de un límite mejor. Un tema abierto sería cuál es la frecuencia idónea de estos recorridos. En la decisión de tal frecuencia pueden intervenir diversos factores, entre los que podría encontrarse incluso la bondad de la heurística utilizada en el algoritmo. Una buena heurística puede ofrecer mayores garantías de que el camino que estamos siguiendo hacia la solución es acertado, consiguiendo de esa forma soluciones razonablemente buenas. Invertir en tales recorridos puede tener más sentido que si la búsqueda está menos informada.

La misma técnica puede servir para implementar algoritmos de respuesta rápida o limitada en el tiempo (algoritmos *any-time*), en los que se pretende que se ofrezca una solución en un tiempo limitado, normalmente asociados a la propia ejecución del proceso de que se trate, en este caso del ensamblaje del producto en cuestión. Evidentemente, ello se hará a costa de perder la optimalidad. Los algoritmos RTA* [Korf, 1990] representan una variante de los algoritmos A* que trabajan según ese requisito. Hay que observar que el algoritmo descrito, conforme vaya realizando más recorridos en profundidad, puede ir mejorando la mejor solución disponible en ada momento. De esta forma, cuanto más tiempo disponga el algoritmo para devolver la solución, ésta podrá estar más cercana a la óptima, en consonancia con lo que debería esperarse del algoritmo.

4.6 Resultados comparativos

Como se ha indicado, existen muchos factores que afectan a la complejidad del problema planteado. Algunos de los más importantes son el número de piezas del producto a ensamblar, el tamaño y estructura del grafo *And/Or*, y la distribución de las duraciones y recursos entre las tareas candidatas a formar parte de la solución. Los resultados que se muestran en las tablas siguientes, bastante significativos del comportamiento de las heurísticas definidas, se basan en un producto hipotético de 30 piezas, considerando 10 distribuciones diferentes de duraciones y recursos entre las tareas. Para estudiar la dependencia de los recursos disponibles, se muestran cuatro tablas, correspondientes a las cuatro combinaciones posibles para 2 y 4 máquinas y 2 y 4 herramientas por máquina.

Como se ha comentado, uno de los problemas más graves de los algoritmos A* es el gran gasto de memoria que pueden llegar a consumir, por lo que se consideró en el algoritmo el uso de recorridos en profundidad cada cierto tiempo, de manera que las soluciones que se obtienen sirven, además, para acotar el valor de la solución óptima buscada y rechazar los nodos con una estimación peor. La ejecución del algoritmo se limitó a 30 segundos, ya que a partir de ese tiempo se observaba un uso abusivo de la memoria virtual. En los resultados se indican en cuántos de los 10 ejemplares se obtuvo la solución óptima a través de un recorrido en profundidad (N-Pr), el número de ejemplares para los que no se alcanzó la solución óptima (N-F), y la tasa de error de la mejor solución encontrada con respecto al óptimo.

Aparte de las heurísticas descritas en el apartado 4.3, se han considerado las combinaciones entre ellas. Así, por ejemplo, dado que no puede decirse que la heurística $h_1(n)$ sea mejor que $h_2(n)$ ni al revés, otra opción es tomar $h(n)$ como

$$h(n) = \max(h_1(n), h_2(n)) \tag{4.64}$$

de manera que la estimación obtenida es la mejor posible de entre ambas. La misma idea puede aplicarse a las heurísticas no comparables, que estarían englobadas en dos grupos diferentes. En un grupo se situarían $h_1(n)$, $h_4(n)$, $h_5(n)$ y $h_6(n)$, que, basadas en la precedencia de tareas, supusieron una secuencia de mejoras entre ellas apoyándose en información sacada del uso de recursos. Todas esas heurísticas se obtenían a partir de la tarea más desfavorable en cuanto a su tiempo total acumulado. En el otro grupo, $h_2(n)$ y $h_3(n)$ se basaban en contemplar todos los usos pendientes de cada una de las herramientas y máquinas por las tareas restantes, resultando un efecto aditivo que no se tenían en cuenta en las anteriores. Debido a esto, las heurísticas de diferentes grupos no pueden ser comparadas a priori, por lo que tendrá sentido seleccionar el mejor valor como en la ecuación (4.64).

Tabla 4.2: Resultados comparativos para 2 máquinas y 2 herramientas por cada máquina

Heurística	Nodos visitados			Tiempo (ms)			N-Pr	N-F	% Error
	Med	Max	Min	Med	Max	Min			
h_1	41492	89189	2520	19429	30590	180	4	6	0,990
h_2	9316	42775	32	1422	6420	0	5	0	0,000
h_{31}	617	3019	32	176	870	0	2	0	0,000
h_{32}	483	2324	32	216	1710	0	1	0	0,000
h_{33}	938	5826	32	3297	30090	0	1	1	0,330
h_{42}	36730	58113	2016	20062	30520	210	4	6	1,403
h_{431}	40693	70414	1020	21915	30980	160	3	7	1,650
h_{432}	40311	70757	1020	21883	31040	160	3	7	1,650
h_{433}	40257	70209	1020	21897	30920	160	3	7	1,650
h_{52}	41384	70823	1017	23869	30590	270	3	7	2,063
h_{531}	40618	69650	1020	21854	30430	160	3	7	2,558
h_{532}	40607	70135	1020	21860	30540	160	3	7	2,558
h_{533}	40600	70133	1020	21845	30600	170	3	7	2,558
h_{61}	37464	69980	2520	19323	30430	240	4	6	0,990
h_{642}	40624	55579	2016	24874	30820	280	2	8	1,980
h_{6431}	38274	67003	4024	20926	30980	610	4	6	1,485
h_{6432}	37859	67013	4024	20801	30370	600	4	6	1,485
h_{6433}	37732	67576	4024	20772	30370	600	4	6	1,485
h_{652}	39198	54860	1017	21894	30370	220	3	7	2,393
h_{6531}	38834	67179	4024	21987	30920	600	3	7	2,558
h_{6532}	38758	67649	4024	22070	30760	610	3	7	2,558
h_{6533}	38706	68257	4024	21974	30480	600	3	7	2,558

Tabla 4.2 (cont): Resultados comparativos para 2 máquinas y 2 herramientas por cada máquina

Heurística	Nodos visitados			Tiempo (ms)			N-Pr	N-F	% Error
	Med	Max	Min	Med	Max	Min			
$\max(h_1, h_2)$	16385	71585	32	4291	30050	0	4	1	0,248
$\max(h_1, h_{31})$	3422	30506	32	3257	31470	0	1	1	0,248
$\max(h_1, h_{32})$	460	2011	32	411	1600	0	1	0	0,000
$\max(h_1, h_{33})$	847	5781	32	3312	30100	0	1	1	0,330
$\max(h_{42}, h_2)$	6757	23167	32	992	3290	0	3	0	0,000
$\max(h_{431}, h_{31})$	352	1724	32	99	550	0	2	0	0,000
$\max(h_{432}, h_{32})$	352	1724	32	232	1380	0	2	0	0,000
$\max(h_{433}, h_{33})$	928	5790	32	3356	30590	0	1	1	0,330
$\max(h_{52}, h_2)$	11702	63996	32	3904	30050	0	2	1	0,165
$\max(h_{531}, h_{31})$	334	1576	32	98	550	0	2	0	0,000
$\max(h_{532}, h_{32})$	334	1576	32	312	1650	0	2	0	0,000
$\max(h_{533}, h_{33})$	914	5833	32	3351	30540	0	1	1	0,330
$\max(h_{61}, h_2)$	16173	70432	32	4267	30150	0	3	1	0,248
$\max(h_{61}, h_{31})$	3415	30437	32	3186	30260	0	1	1	0,248
$\max(h_{61}, h_{32})$	465	2011	32	380	1490	0	1	0	0,000
$\max(h_{61}, h_{33})$	847	5835	32	3384	30710	0	1	1	0,330
$\max(h_{642}, h_2)$	5458	18254	32	911	3850	0	4	0	0,000
$\max(h_{6431}, h_{31})$	319	1436	32	89	500	0	2	0	0,000
$\max(h_{6432}, h_{32})$	229	1436	32	154	1150	0	1	0	0,000
$\max(h_{6433}, h_{33})$	788	5712	32	3247	30540	0	1	1	0,330
$\max(h_{652}, h_2)$	4688	17858	32	774	3130	0	4	0	0,000
$\max(h_{6531}, h_{31})$	210	1346	32	60	490	0	2	0	0,000
$\max(h_{6532}, h_{32})$	210	1346	32	149	1100	0	2	0	0,000
$\max(h_{6533}, h_{33})$	784	5774	32	3307	30700	0	1	1	0,330

Tabla 4.3: Resultados comparativos para 2 máquinas y 4 herramientas por cada máquina

Heurística	Nodos visitados			Tiempo (ms)			N-Pr	N-F	% Error
	Med	Max	Min	Med	Max	Min			
h_1	40093	56108	3020	25025	30930	710	2	8	2,648
h_2	5311	19398	267	790	4230	50	1	0	0,000
h_{31}	1458	6120	32	198	930	0	1	0	0,000
h_{32}	1377	6120	32	390	2140	0	1	0	0,000
h_{33}	2598	7032	98	4201	30040	110	0	1	1,059
h_{42}	32576	50203	32	21731	30540	0	2	7	2,950
h_{431}	30720	52833	2016	17085	30380	440	4	5	1,967
h_{432}	30669	52941	2016	17063	30320	440	4	5	1,967
h_{433}	30714	52901	2016	17066	30100	440	4	5	1,967
h_{52}	31264	50158	32	21635	30920	0	2	7	2,648
h_{531}	30635	55847	4015	17220	30810	880	4	5	1,891
h_{532}	30616	56304	4015	17091	30100	1480	4	5	1,891
h_{533}	30559	55847	4015	17235	30590	880	4	5	1,891
h_{61}	39383	59355	3020	22469	31360	550	2	7	2,345
h_{642}	32571	47704	32	18829	30160	0	3	6	2,648
h_{6431}	30276	54303	2015	17225	30540	490	4	5	2,496
h_{6432}	30256	54292	2015	17183	30650	500	4	5	2,496
h_{6433}	30245	54331	2015	17132	30320	550	4	5	2,496
h_{652}	32514	47865	32	21656	30420	0	3	6	2,723
h_{6531}	29944	50685	3015	17253	30380	660	4	5	1,967
h_{6532}	29920	50678	3015	17263	30270	660	4	5	1,967
h_{6533}	29869	50611	3015	17179	30270	710	4	5	1,967

Tabla 4.3 (cont): Resultados comparativos para 2 máquinas y 4 herramientas por cada máquina

Heurística	Nodos visitados			Tiempo (ms)			N-Pr	N-F	% Error
	Med	Max	Min	Med	Max	Min			
$\max(h_1, h_2)$	5078	19009	387	1143	4450	60	1	0	0,000
$\max(h_1, h_{31})$	1303	4805	32	418	1870	0	1	0	0,000
$\max(h_1, h_{32})$	1236	4805	32	864	4670	60	1	0	0,000
$\max(h_1, h_{33})$	2478	6622	373	5794	30750	220	0	1	1,513
$\max(h_{42}, h_2)$	3811	18011	32	776	4230	0	2	0	0,000
$\max(h_{431}, h_{31})$	1261	4468	32	645	2530	0	1	0	0,000
$\max(h_{432}, h_{32})$	1194	4468	32	1027	4280	0	1	0	0,000
$\max(h_{433}, h_{33})$	2391	6172	358	5306	30600	220	0	1	1,135
$\max(h_{52}, h_2)$	3508	18005	387	764	4230	110	1	0	0,000
$\max(h_{531}, h_{31})$	1258	4451	32	584	2480	0	1	0	0,000
$\max(h_{532}, h_{32})$	1192	4451	32	1072	4290	50	1	0	0,000
$\max(h_{533}, h_{33})$	1874	4451	358	2314	9660	280	0	1	0,983
$\max(h_{61}, h_2)$	5078	19009	387	984	4450	110	1	0	0,000
$\max(h_{61}, h_{31})$	1303	4805	32	462	1920	0	1	0	0,000
$\max(h_{61}, h_{32})$	1236	4805	32	917	4500	0	1	0	0,000
$\max(h_{61}, h_{33})$	2469	6632	373	5795	31030	220	0	1	1,513
$\max(h_{642}, h_2)$	3592	18167	32	701	4280	0	3	0	0,000
$\max(h_{6431}, h_{31})$	486	2111	32	171	880	0	1	0	0,000
$\max(h_{6432}, h_{32})$	408	2111	32	363	2090	0	1	0	0,000
$\max(h_{6433}, h_{33})$	1713	6016	98	4191	30540	110	0	1	1,059
$\max(h_{652}, h_2)$	3496	18165	295	691	4280	50	2	0	0,000
$\max(h_{6531}, h_{31})$	484	2098	32	175	870	0	1	0	0,000
$\max(h_{6532}, h_{32})$	406	2098	32	367	2090	0	1	0	0,000
$\max(h_{6533}, h_{33})$	1701	6024	98	4130	30160	110	0	1	1,059

Tabla 4.4: Resultados comparativos para 4 máquinas y 2 herramientas por cada máquina

Heurística	Nodos visitados			Tiempo (ms)			N-Pr	N-F	% Error
	Med	Max	Min	Med	Max	Min			
h_1	16905	85540	32	2357	11700	0	1	0	0,000
h_2	2751	7195	32	263	710	0	2	0	0,000
h_{31}	1781	7011	32	533	2030	0	3	0	0,000
h_{32}	3011	13334	32	1389	5880	0	2	0	0,000
h_{33}	1760	7011	32	1133	5000	0	3	0	0,000
h_{42}	16636	84888	32	2719	15600	0	1	0	0,000
h_{431}	16423	82760	32	2758	15600	0	1	0	0,000
h_{432}	16267	81864	32	2747	15600	0	1	0	0,000
h_{433}	15994	80098	32	2747	15600	0	1	0	0,000
h_{52}	16249	83346	32	2713	15600	0	1	0	0,000
h_{531}	16056	82230	32	2745	15600	0	1	0	0,000
h_{532}	15687	80980	32	2746	15600	0	1	0	0,000
h_{533}	15515	79980	32	2746	15600	0	1	0	0,000
h_{61}	15539	80121	32	2368	11810	0	1	0	0,000
h_{642}	12812	80005	32	2269	15600	0	5	0	0,000
h_{6431}	12662	78919	32	2637	15600	0	4	0	0,000
h_{6432}	11914	72099	32	2598	14780	0	4	0	0,000
h_{6433}	11212	65702	32	2549	14170	0	4	0	0,000
h_{652}	6970	38543	32	994	5550	0	4	0	0,000
h_{6531}	7212	37000	32	1124	5980	0	5	0	0,000
h_{6532}	7135	36230	32	1136	5990	0	5	0	0,000
h_{6533}	7066	35540	32	1132	5990	0	5	0	0,000

Tabla 4.4 (cont): Resultados comparativos para 4 máquinas y 2 herramientas por cada máquina

Heurística	Nodos visitados			Tiempo (ms)			N-Pr	N-F	% Error
	Med	Max	Min	Med	Max	Min			
$\max(h_1, h_2)$	804	2098	32	99	220	0	2	0	0,000
$\max(h_1, h_{31})$	794	2098	32	148	330	0	2	0	0,000
$\max(h_1, h_{32})$	794	2098	32	207	550	0	2	0	0,000
$\max(h_1, h_{33})$	794	2098	32	223	600	0	2	0	0,000
$\max(h_{42}, h_2)$	789	2098	32	88	220	0	1	0	0,000
$\max(h_{431}, h_{31})$	789	2098	32	148	330	0	1	0	0,000
$\max(h_{432}, h_{32})$	789	2098	32	208	550	0	1	0	0,000
$\max(h_{433}, h_{33})$	789	2098	32	225	600	0	1	0	0,000
$\max(h_{52}, h_2)$	878	2098	32	104	220	0	1	0	0,000
$\max(h_{531}, h_{31})$	787	2098	32	148	330	0	1	0	0,000
$\max(h_{532}, h_{32})$	787	2098	32	208	550	0	1	0	0,000
$\max(h_{533}, h_{33})$	787	2098	32	224	600	0	1	0	0,000
$\max(h_{61}, h_2)$	804	2098	32	92	220	0	2	0	0,000
$\max(h_{61}, h_{31})$	794	2098	32	148	330	0	2	0	0,000
$\max(h_{61}, h_{32})$	794	2098	32	209	550	0	2	0	0,000
$\max(h_{61}, h_{33})$	794	2098	32	225	600	0	2	0	0,000
$\max(h_{642}, h_2)$	772	2098	32	115	440	0	1	0	0,000
$\max(h_{6431}, h_{31})$	536	2098	32	197	760	0	1	0	0,000
$\max(h_{6432}, h_{32})$	536	2098	32	335	1270	0	1	0	0,000
$\max(h_{6433}, h_{33})$	536	2098	32	377	1480	0	1	0	0,000
$\max(h_{652}, h_2)$	699	2098	32	84	220	0	1	0	0,000
$\max(h_{6531}, h_{31})$	536	2098	32	198	770	0	1	0	0,000
$\max(h_{6532}, h_{32})$	536	2098	32	341	1270	0	1	0	0,000
$\max(h_{6533}, h_{33})$	536	2098	32	381	1490	0	1	0	0,000

Tabla 4.5: Resultados comparativos para 4 máquinas y 4 herramientas por cada máquina

Heurística	Nodos visitados			Tiempo (ms)			N-Pr	N-F	% Error
	Med	Max	Min	Med	Max	Min			
h_1	22697	93376	32	6084	30530	0	4	1	0,302
h_2	1808	5907	128	197	660	0	1	0	0,000
h_{31}	1072	5014	119	396	2260	0	1	0	0,000
h_{32}	1070	5014	119	669	4010	60	1	0	0,000
h_{33}	1071	5014	119	913	5880	160	1	0	0,000
h_{42}	19715	88102	32	5118	30810	0	4	1	0,101
h_{431}	15560	87005	32	4322	30320	0	6	1	0,302
h_{432}	15480	86207	32	4323	30210	0	6	1	0,302
h_{433}	15372	85123	32	4333	30430	0	6	1	0,302
h_{52}	19287	87456	32	5068	30810	0	5	1	0,403
h_{531}	6961	55768	32	1284	10930	0	7	0	0,000
h_{532}	6820	55768	32	1336	11430	0	7	0	0,000
h_{533}	6807	55768	32	1373	11820	0	7	0	0,000
h_{61}	22592	92326	32	7409	31420	0	4	1	0,302
h_{642}	16555	81332	32	4421	30100	0	6	1	0,302
h_{6431}	7854	52011	32	1324	9610	0	8	0	0,000
h_{6432}	7754	52011	32	1296	9550	0	8	0	0,000
h_{6433}	7704	52011	32	1302	9560	0	8	0	0,000
h_{652}	8110	45010	32	1302	8190	0	7	0	0,000
h_{6531}	16362	85114	32	4329	30260	0	7	1	0,201
h_{6532}	7112	44589	32	1302	9610	0	8	0	0,000
h_{6533}	16180	83290	32	4339	30260	0	7	1	0,201

Tabla 4.5 (cont): Resultados comparativos para 4 máquinas y 4 herramientas por cada máquina

Heurística	Nodos visitados			Tiempo (ms)			N-Pr	N-F	% Error
	Med	Max	Min	Med	Max	Min			
max(h_1, h_2)	1765	5907	32	278	980	0	2	0	0,000
max(h_1, h_{31})	1062	5014	119	396	2260	0	1	0	0,000
max(h_1, h_{32})	1060	5014	119	669	4010	60	1	0	0,000
max(h_1, h_{33})	769	2011	119	522	2260	110	1	0	0,000
max(h_{42}, h_2)	1711	5907	32	284	1040	0	2	0	0,000
max(h_{431}, h_{31})	1042	5014	119	413	2430	0	1	0	0,000
max(h_{432}, h_{32})	1040	5014	119	724	4560	60	1	0	0,000
max(h_{433}, h_{33})	758	2014	119	571	2410	110	1	0	0,000
max(h_{52}, h_2)	1710	5907	32	290	1100	0	2	0	0,000
max(h_{531}, h_{31})	1042	5014	119	439	2690	0	1	0	0,000
max(h_{532}, h_{32})	1040	5014	119	744	4760	60	1	0	0,000
max(h_{533}, h_{33})	745	2014	119	565	2580	110	1	0	0,000
max(h_{61}, h_2)	1646	5812	32	290	1100	0	2	0	0,000
max(h_{61}, h_{31})	1002	4920	119	429	2590	0	1	0	0,000
max(h_{61}, h_{32})	996	4920	119	730	4620	60	1	0	0,000
max(h_{61}, h_{33})	689	2011	119	560	2640	110	1	0	0,000
max(h_{642}, h_2)	1073	4833	32	131	600	0	1	0	0,000
max(h_{6431}, h_{31})	544	2937	32	259	1480	0	1	0	0,000
max(h_{6432}, h_{32})	544	2937	32	451	2640	0	1	0	0,000
max(h_{6433}, h_{33})	544	2937	32	510	3020	0	1	0	0,000
max(h_{652}, h_2)	951	4027	32	123	550	0	1	0	0,000
max(h_{6531}, h_{31})	523	2936	32	241	1490	0	1	0	0,000
max(h_{6532}, h_{32})	523	2936	32	423	2580	0	1	0	0,000
max(h_{6533}, h_{33})	523	2936	32	496	3020	0	1	0	0,000

Una de las observaciones más destacadas a partir de los resultados obtenidos es la gran disparidad de tiempos de ejecución y memoria ocupada máxima entre problemas del mismo tamaño para una misma heurística, sobre todo cuando se tienen pocos recursos para compartir entre las tareas de ensamblaje. Por otro lado, para los casos con dos máquinas, las heurísticas de la familia de h_1 obtienen malos resultados, no consiguiendo encontrar la solución óptima en un gran número de casos. No obstante, los resultados alcanzados muestran las mejoras que se han ido consiguiendo al definir las nuevas heurísticas mediante los refinamientos sucesivos. Las heurísticas de la familia h_2 obtienen unos resultados más homogéneos. Otra mejora viene dada por el uso combinado de ambas familias de heurísticas, lo que indica que la mejor elección de la heurística también depende del estado del proceso de búsqueda, es decir, de la zona del grafo *And/Or* que reste por explorar a partir de cada nodo de expansión.

En el análisis de los resultados debe tenerse en cuenta también que en muchas ocasiones la solución óptima se ha logrado mediante un recorrido en profundidad, que se realiza a partir del nodo más prometedor en el momento en que se emprende dicho recorrido, lo que dota a la estrategia de un cierto grado de aleatoriedad. Esto explica que se obtengan en algunos casos resultados aparentemente contradictorios, en los que heurísticas peor informadas obtienen resultados ligeramente mejores que otras mejor informadas.

4.7 Conclusiones

En este capítulo se ha presentado un algoritmo basado en la búsqueda A* para la resolución del problema de planificación propuesto. El algoritmo se compone de dos partes bien diferenciadas. Por un lado, se tratan las tareas que, por las restricciones de precedencia definidas en el grafo *And/Or*, se ejecutarán secuencialmente. Por otro lado, se resuelve la ordenación de las tareas que potencialmente se podrían ejecutar en paralelo. Esta última parte es de hecho la más compleja, y para la que se han realizado los mayores esfuerzos en resolverla.

Se han definido distintas heurísticas admisibles, partiendo de dos modelos relajados del problema. Por un lado, en la heurística h_1 sólo se tienen en cuenta las restricciones de precedencia entre tareas, lo que nos permite obtener una medida del tiempo restante para las tareas contenidas en cada subárbol de precedencia. Por otro lado, en la heurística h_2 sólo se consideran los tiempos totales de uso de cada recurso. A partir de las heurísticas anteriores, se definen otras que integran en la heurística base información parcial del otro tipo, de manera que el refinamiento supone una mejora respecto a una de las heurísticas citadas.

La heurística h_2 es mejorada analizando el árbol de precedencias de tareas para estimar con mayor exactitud el número de cambios de herramientas necesarios. Para ello se han construido varios modelos basados en el uso de multiconjuntos parcialmente ordenados (*pomsets*). Se han considerado también las indeterminaciones en el número de usos no consecutivos de las herramientas que podrían generarse al analizar el árbol

de precedencias. El esfuerzo realizado en la definición de los distintos modelos se ha visto compensado por los resultados obtenidos, bien directamente en su uso como heurística elegida, bien de forma indirecta a través de otras heurísticas que utilizaban sus cálculos.

La heurística h_1 es mejorada de distintas formas, bien usando los resultados obtenidos por h_2 y h_3, bien a través de la determinación de distintos retardos asociados a la imposibilidad de usar un determinado recurso en diferentes intervalos.

El comportamiento de ambos grupos de heurísticas es diferente: las heurísticas de la familia de h_1 afectan a la estimación del nodo a través de la tarea candidata más desfavorable; las heurísticas de la familia de h_2 tienen un efecto aditivo, de manera que se suman los tiempos de uso de los recursos para todas las tareas candidatas. Como se refleja en los resultados obtenidos, un grupo de heurísticas dominará sobre el otro en función de los recursos definidos para el problema y de la estructura del grafo *And/Or*, como factores principales. Así, cuando se disponga de pocos recursos compartidos por muchas tareas, las heurísticas de tipo h_2 serán más relevantes, mientras que cuando se disponga de muchos recursos y/o la estructura del grafo *And/Or* sea poco equilibrada, las heurísticas de tipo h_1 adquieren más peso.

Para mejorar el comportamiento del algoritmo se ha definido, por un lado, la relación de compatibilidad entre tareas, lo cual ha permitido detectar simetrías del problema, así como ignorar partes importantes del espacio de búsqueda en donde no podía encontrarse la solución óptima. Por otro lado, se ha propuesto el uso de recorridos en profundidad para detectar nuevas cotas para la solución óptima, lo que permite un mejor aprovechamiento de la memoria disponible.

Los resultados obtenidos muestran fundamentalmente el efecto de las distintas heurísticas definidas, y cómo una mejor en las mismas conduce a un mejor comportamiento del algoritmo. Ello se consigue, como ha podido observarse en el desarrollo de las heurísticas, a través de la información relevante del problema, lo cual no siempre resulta fácil.

El algoritmo puede ser usado fuera de línea para obtener una solución óptima inicial para el proceso de ensamblaje. Sin embargo, debido por un lado a la flexibilidad para modificar el criterio de convergencia del algoritmo hacia una solución no estrictamente óptima, y por otro al hecho de que conforme avance el proceso de ensamblaje el problema resultante es más pequeño, el algoritmo podría ser aplicable en línea para modificar bien el plan de ensamblaje o bien la secuencia inicial, con objeto de corregir las variaciones con respecto a la solución inicial.

Capítulo 5

Resolución mediante Programación con Restricciones

5.1 Introducción

Este capítulo está dedicado al planteamiento y resolución del problema de planificación propuesto usando el paradigma de la Programación con Restricciones. Se trata de una tecnología emergente que ha conseguido atraer la atención de expertos provenientes de varias disciplinas, de acuerdo a su potencial para resolver problemas complejos. Una de las características más relevantes de la programación con restricciones es la forma declarativa de formular el problema y las soluciones al mismo, es decir, se especifican las relaciones que deben guardar los distintos componentes, sin indicar el modelo computacional que se usa para obtener las soluciones que consigan cumplir dichas relaciones.

Entre los tipos de problemas que han sido abordados utilizando estas técnicas se encuentran problemas de tipo combinatorio, y especialmente los relativos a planificación y *scheduling*, por ejemplo, el ampliamente estudiado *Job Shop Scheduling* [Caseau and Laburthe, 1995] [Esquirol, *et al.*, 1996].

El enfoque utilizado aquí ha sido la expresión en un lenguaje de programación lógica con restricciones, concretamente CLP(R), de las distintas restricciones que definen el problema, y plantear su resolución a través de algoritmos de *branch and*

bound, que, de forma incremental, van incorporando nuevas restricciones y aplicando reglas de consistencia, capaces de limitar el espacio de búsqueda. Se utilizan, además, distintas heurísticas encaminadas a mejorar la eficiencia de la búsqueda.

La estructura del problema, especificado a través del grafo *And/Or* correspondiente al ensamblaje del producto, incluye por un lado las restricciones de precedencia entre las operaciones de ensamblaje. No todas las tareas están ligadas con las demás a través de restricciones de precedencia, de forma que, dado que se está considerando un sistema con múltiples máquinas, existirán conjuntos de operaciones que puedan ejecutarse de forma simultánea.

Por otro lado, el uso de recursos compartidos (las máquinas de ensamblaje) provoca que aparezcan restricciones de tipo disyuntivo entre las tareas que utilizan una misma máquina. Esas restricciones corresponden a que, para cada dos tareas, una podría ejecutarse antes que la otra o viceversa, dando lugar a dos opciones distintas y excluyentes.

Hay un último aspecto que complica aún más el problema, y es el hecho de que el conjunto de tareas que deben componer la solución buscada del problema no está previamente especificado. Así, en el grafo *And/Or* aparecen las tareas alternativas posibles para la construcción de cada submontaje intermedio, y los posibles planes de ensamblaje, que se corresponden con los distintos árboles binarios de tareas que pueden extraerse del grafo. Estas restricciones también poseen una naturaleza disyuntiva, de ahí el factor de complejidad que añaden al problema.

Basado en la estructura en árbol tanto de las restricciones de precedencia como las disyuntivas asociadas a las operaciones alternativas, los algoritmos propuestos realizan un tratamiento incremental de todas las restricciones disyuntivas a través del recorrido del grafo *And/Or*, de forma que se aprovechan los cálculos realizados para las tareas situadas cerca de la raíz, comunes a muchos planes de ensamblaje.

Para conseguir un mejor comportamiento computacional de los algoritmos, se utilizan distintas heurísticas, que ayudarán a acotar el espacio de búsqueda, manteniendo sólo aquellas alternativas que puedan dar lugar a la solución óptima buscada. Tales heurísticas son similares a las propuestas en el Capítulo 4, pero incorporando en su definición la posibilidad de seleccionar tareas alternativas. Las heurísticas desarrolladas no incluyen aún las restricciones asociadas al uso de recursos compartidos entre tareas que no guardan relación de precedencia alguna entre ellas, correspondientes a la familia de heurísticas basadas en h_2 estudiadas en el Capítulo 4.

Las respuestas que se obtienen poseen mayor riqueza que las que se obtuvieron usando las técnicas de los capítulos anteriores, ya que no todas las variables temporales asociadas a los comienzos y finales de las tareas son instanciadas, sino que permanecen las restricciones en forma de intervalos, mostrando las holguras existentes.

El resto del capítulo se estructura de la siguiente manera: en la siguiente sección se realiza una introducción a la programación con restricciones, mostrando los elementos más relevantes del paradigma, e incluyendo al final las características principales del lenguaje CLP(R), en el que se han desarrollado los algoritmos propuestos. En la sección siguiente se muestra la adaptación del problema planteado en CLP(R). A continuación

se muestra el algoritmo de *backtracking* para la resolución del problema. En la siguiente sección se plantea la resolución mediante algoritmos de *branch and bound*, en los que se utilizan distintas heurísticas para mejorar su eficiencia. Los resultados obtenidos son presentados en la siguiente sección, y por último en las conclusiones se realizan algunas puntualizaciones finales.

5.2 Conceptos generales

El concepto clave sobre el que se basa este paradigma de programación es el de *restricción*. Una restricción no es más que es una relación lógica entre variables pertenecientes a distintos dominios, de forma que restringe los valores que pueden tomar. Tales dominios pueden ser de naturaleza heterogénea, desde valores numéricos a simbólicos. Las restricciones expresan las relaciones entre las variables de manera no direccional, pudiendo inferirse resultados a partir de unas variables u otras. Otro aspecto importante es la forma declarativa y aditiva en que se expresan las relaciones entre las variables, sin importar el orden en que aparecen, y sin exteriorizarse el método de resolución de las mismas en el programa.

Estas ideas, ya integradas en la Programación Lógica, fueron explotadas para hacer más eficiente la resolución de problemas, de manera que se incorporaron técnicas de consistencia que ayudaran a descartar la búsqueda sobre aquellas zonas correspondientes a combinaciones de valores que no podían aparecer en la solución, de acuerdo a las deducciones inferidas a partir de las restricciones consideradas.

La mayoría de problemas combinatorios pueden formularse de manera natural a través de problemas de satisfacción de restricciones, utilizando dominios finitos para sus variables. Sobre tales tipos de problemas se centran los distintos métodos de resolución desarrollados en la programación con restricciones. En las siguientes subsecciones se presentan los conceptos más relevantes.

5.2.1 Problemas de satisfacción de restricciones (CSP)

Un problema de satisfacción de restricciones (*Constraint Satisfaction Problem – CSP*) puede definirse a partir de un conjunto de *variables* $X = \{X_1, X_2, ..., X_n\}$, las cuales pueden tomar valores de sus respectivos *dominios* finitos, $D_1, D_2, ..., D_n$, y un conjunto de *restricciones* que especifican, de una manera explícita o implícita, las combinaciones de valores compatibles que pueden tomar las variables. Una solución a un CSP es una asignación de valores a todas las variables, de modo que se satisfagan todas las restricciones del problema. En muchas ocasiones, el objetivo es encontrar el conjunto de todas esas asignaciones. Formulaciones más generales del problema pueden encontrarse en diversas fuentes, como [Mackworth, 1977].

Una restricción puede afectar a un número de variables, desde 1 hasta *n*, el número total de variables del problema. Dos casos particulares son de especial interés, las

restricciones *unarias* y *binarias*. El tratamiento de las restricciones unarias se corresponderá con un procesamiento previo de los dominios de las variables afectadas, por lo que podrán ser ignoradas a partir de entonces. Si todas las restricciones de un CSP son binarias, las variables y restricciones pueden ser representadas a través de un grafo en el que las variables estarán asociadas a los vértices del grafo y las restricciones entre dos variables se representan mediante una arista que une a los vértices asociados. Tal representación es conocida como *grafo de restricciones*, y a los CSP que pueden ser representados mediante ellos (incluyendo también restricciones unarias), se les denomina CSP *binarios*. Los algoritmos de satisfacción de restricciones podrán entonces basarse en los algoritmos de recorridos sobre grafos. Por otro lado, cualquier restricción que involucre a más de dos variables puede ser expresada a partir de restricciones binarias, por lo que los CSP binarios constituyen un marco representativo de todos los CSP. Sin embargo, es preciso señalar que la traducción de un CSP genérico a otro binario podría suponer un coste computacional elevado. La mayoría de las técnicas de consistencia desarrolladas para resolver los CSP consideran restricciones binarias.

El hecho de que los dominios sean finitos permite que los esquemas de búsqueda puedan ser representados a través de un árbol de profundidad limitada. Las decisiones de propagación de restricciones para detectar inconsistencias se corresponden con podas en el árbol de búsqueda.

En cuanto a la complejidad de esta clase de problemas, dado que el problema de la satisfactibilidad de expresiones booleanas (SAT) es NP-completo [Garey and Johnson, 1979], un algoritmo genérico diseñado para resolver cualquier problema de la clase requerirá necesariamente un tiempo que, en el caso peor, crece de forma exponencial con el tamaño del problema.

La resolución de CSP puede abordarse desde dos puntos de vista diferentes. Por un lado, a través de algoritmos de búsqueda sistemática, que recorren el espacio de soluciones en busca de combinaciones de valores que satisfagan el conjunto de restricciones. Por otro lado, a través de la propagación de restricciones, que derivan en problemas más simples definidos por restricciones equivalentes. Ambas estrategias son mezcladas a menudo para obtener mejores algoritmos.

5.2.2 Métodos de búsqueda sistemática

El método más sencillo para resolver un CSP es mediante un algoritmo GT (*generate and test*). Se trata de obtener todas las combinaciones de valores para las variables de forma sistemática, y verificar para cada una de ellas si se satisfacen todas las restricciones del problema. El número de combinaciones que se considerará es el tamaño del producto cartesiano de los dominios de todas las variables. Este método es muy ineficiente, ya que genera muchas asignaciones no válidas que son rechazadas demasiado tarde. Además, el orden en que se generan los valores de las variables no tiene en cuenta los posibles conflictos que puedan darse.

Un método más eficiente que el anterior es mediante el esquema de *backtracking* (BT). En este método, las variables son instanciadas de forma secuencial, obteniéndose

asignaciones parciales que involucran a un número de variables cada vez mayor, con el objetivo de llegar a una asignación de todas las variables que satisfaga el conjunto de todas las restricciones. Cada vez que una variable es instanciada, se comprueba si se satisfacen las restricciones asociadas a la misma. Si una asignación parcial viola alguna de las restricciones, el método hace retroceder la búsqueda, probando una nueva instanciación sobre la variable más reciente que tenga valores alternativos en su dominio por examinar. De esta forma, el método de *backtracking* es capaz de eliminar un subespacio del producto cartesiano de los dominios sin necesidad de examinar todas las combinaciones asociadas al mismo.

Aunque mejor que GT, el método de *backtracking* sigue siendo bastante ineficiente. Su comportamiento patológico es conocido como *thrashing*, y se manifiesta en los siguientes hechos:

- La búsqueda falla repetidamente en diferentes zonas por las mismas causas. Esto ocurre porque el algoritmo de *backtracking* no identifica la verdadera causa del conflicto, es decir, las variables asociadas al mismo.
- Las inconsistencias se detectan tarde, y después de algunas instanciaciones inútiles, incrementándose de esta forma el número de retrocesos que debe realizar el algoritmo.
- El punto de retroceso no es escogido a partir de las causas del fallo, con lo que el algoritmo realiza un trabajo redundante.

Los problemas señalados han ocasionado que el esquema de *backtracking* haya sido objeto de diferentes refinamientos sobre el esquema básico, introduciendo distintos métodos de propagación de restricciones, y poniendo especial énfasis en la detección temprana de inconsistencias.

5.2.3 Técnicas de consistencia

La detección tardía de inconsistencia es el principal problema de los algoritmos de GT y *backtracking*. Para corregir este problema se han introducido distintas técnicas de consistencia que ayudan a podar el espacio de soluciones. Esas técnicas van desde las llamadas *consistencia de nodo* y *consistencia de arco* hasta la más completa pero costosa *consistencia de camino*. La nomenclatura hace referencia al uso de los grafos de restricciones para representar los CSP.

Consistencia de nodo (*node-consistency* – NC)

Se trata de la técnica de consistencia más simple. Un vértice del grafo de restricciones es *nodo-consistente* si todos los valores del dominio de la variable correspondiente satisfacen todas las restricciones unarias impuestas sobre esa variable. El algoritmo NC elimina los valores de los dominios de las variables que sean inconsistentes con las restricciones unarias. Tras la ejecución de NC, que puede realizarse una sola vez al

principio del algoritmo de resolución del CSP, si todos los vértices son nodo-consistentes (sus dominios son no vacíos), se obtendrá un CSP nodo-consistente.

Consistencia de arco (*arc-consistency* – AC)

Se trata de la técnica más usada, dado su buen rendimiento al eliminar inconsistencias con un coste computacional moderado. Básicamente, se trata de eliminar los valores de los dominios de las variables que sean inconsistentes con las restricciones binarias. Una restricción se dice que es *arco-consistente* si para cualquier valor del dominio de cualquiera de las variables que aparece en la restricción, existe un valor en el dominio de la otra que hace que la restricción se satisfaga.

Existen varios algoritmos que verifican la consistencia de arco de un CSP, catalogados los más conocidos desde AC-1 hasta AC-7, cuyo coste computacional es polinómico. Tales algoritmos revisan repetidamente la consistencia de las aristas del grafo, dado que una vez que el dominio de alguna variable ha sido reducido, debe considerarse de nuevo la consistencia de todas las restricciones en donde aparecen. El resultado final es la reducción del dominio de las variables de manera que el grafo es arco-consistente, o, si alguno de los dominios es vacío, la determinación de la inconsistencia del CSP.

Por otro lado, es necesario resaltar que, en general, la determinación de la consistencia de arco de un CSP es insuficiente para determinar las posibles soluciones del problema, o incluso si existe solución al mismo. Sólo en el caso de que todos los dominios de las variables contengan un único valor, o cuando alguno de ellos haya quedado vacío, se podrán obtener resultados definitivos, en un caso dando la solución del CSP, y en el otro determinando la no existencia de solución al mismo.

Consistencia de camino (*path-consistency* – PC)

Dado que las técnicas de consistencia de arco no eliminan todas las inconsistencias del grafo de restricciones, se han propuesto otras técnicas para intentar lograr ese objetivo. La extensión natural a la consistencia de arco es la consistencia de camino.

Se dice que un camino $(X_1,..., X_n)$ es camino-consistente si para cualquier par (x_1, x_n) de valores consistentes (satisfacen todas las restricciones binarias entre X_1 y X_n), existen valores $x_2,..., x_{n-1}$ tales que todas las restricciones X_i, X_{i+1} se satisfacen. Obsérvese que no se indica nada acerca de las restricciones entre X_i y X_j con $|i-j|>1$. Un CSP es camino-consistente si todos sus caminos son camino-consistentes. [Montanary, 1974] mostró sin embargo que para verificar la consistencia de camino de un CSP basta con que lo sean todos los caminos de longitud 2.

Existen diversos algoritmos que verifican la consistencia de camino. Aunque se detectan más inconsistencias que en las técnicas de consistencia de arco, la complejidad computacional tanto en tiempo de ejecución como en memoria es muy superior, por lo que no suelen utilizarse.

k-consistencia

Las consistencias de nodo, arco y camino son casos particulares de un concepto más general, el de *k-consistencia*.

Un CSP se dice que es *k-consistente* si y sólo si todas las instanciaciones consistentes de k–1 variables pueden extenderse a una instanciación consistente incluyendo una variable adicional. Un CSP es *fuertemente k-consistente* si y sólo si es *j-consistente* para todo $j \leq k$.

Según, lo anterior, la consistencia de nodo es equivalente a la 1-consistencia fuerte, la consistencia de arco equivale a la 2-consistencia fuerte y la consistencia de camino a la 3-consistencia fuerte.

Existen también algoritmos que examinan la *k-consistencia* de un grafo de restricciones para $k > 2$, pero en la práctica se utilizan escasamente debido a su alto coste computacional.

Puede observarse que si un grafo de restricciones que contenga n vértices fuera *n-consistente*, entonces la solución al CSP puede ser encontrada sin realizar ninguna búsqueda. Sin embargo, la complejidad en el caso peor para detectar la *n-consistencia* es de orden exponencial. Además, en general, si un grafo es *fuertemente k-consistente*, con $k < n$, la búsqueda mediante *backtracking* no puede ser evitada para encontrar la solución. Sin embargo, hay algunos tipos de CSP para los que lo anterior sí es posible, y es cuando la amplitud del grafo es menor que k. La amplitud de un grafo se refiere al número máximo de arcos que conectan a un nodo con los anteriores en una ordenación óptima del grafo [Kumar, 1992]. Es interesante la apreciación de que la amplitud de un grafo estructurado en forma de árbol es igual a 1 [Freuder, 1982], con lo que puede resolverse sin necesidad de *backtracking* evaluando la consistencia de arco del grafo.

5.2.4 Propagación de restricciones

En las subsecciones anteriores se ha visto cómo la búsqueda sistemática en solitario es muy ineficiente, mientras que las técnicas de consistencia no son completas. El enfoque que suele utilizarse para resolver los CSP es una combinación de la búsqueda mediante *backtracking* con técnicas de consistencia.

Esquemas *Look Back*

Los esquemas de *Look Back* verifican la consistencia entre las variables ya instanciadas. El ejemplo más simple es el propio *backtracking*. Para evitar algunos de los problemas que presenta, se han propuesto otros esquemas.

El método de *Backjumping* (BJ) [Gashnig, 1979] se diferencia del *backtracking* a la hora de realizar los movimientos de retroceso. Cuando encuentra una inconsistencia, analiza la situación para determinar la causa de la misma. La restricción violada indica la variable con la que la asignación de la actual tiene el conflicto. Si, analizados todos los valores del dominio de la variable actual, se determina que no hay ninguno consis-

tente, el retroceso se produce hacia la última variable con la que la actual ha tenido conflicto, a diferencia del *backtracking*, que retrocedía a la variable inmediata anterior.

El algoritmo de *Backmarking* (BM) [Gaschnig, 1979] intenta minimizar la ejecución de pruebas de consistencia redundantes. Se reconocen dos situaciones donde se puede dar esa redundancia. Por un lado, cuando la variable actual va a ser reinstanciada con un valor, y éste entró en conflicto con el valor que todavía mantiene otra variable, con lo que BM no necesita considerar esa instanciación para la variable actual. Por otro lado, cuando se va a reinstanciar la variable actual con un valor para el cual se conoce que es compatible con el valor que mantiene otra variable, por lo que BM no necesita revisar de nuevo esa consistencia.

Esquemas *Look Ahead*

Los esquemas de *Look Ahead* verifican la consistencia de las variables que aún no han sido instanciadas, por lo que permitirán detectar las posibles inconsistencias con mayor antelación.

El método de *Forward Checking* (FC) examina la consistencia de arco entre la variable que está siendo instanciada actualmente con las que aún no lo han sido, de forma que los dominios de las variables no instanciadas se reducen según haya valores que entren en conflicto con el valor asignado a la variable actual. De esta forma, se consigue además que los valores de los dominios de las variables aún no instanciadas satisfagan las restricciones con las variables ya instanciadas, con lo que FC no debe revisar la consistencia con ellas, sino únicamente con las variables no instanciadas.

Otros algoritmos, *Partial Lookahead* (PL), *Full Lookahead* (FL) y *Really Lookahead* (RLA), son extensiones de FC en las que se realizan pruebas de consistencia de arco entre las variables que aún no han sido instanciadas [Nadel, 1988].

5.2.5 Orden de instanciación de variables y valores

La elección del orden en el que las variables son instanciadas, y en que los posibles valores son escogidos para la instanciación puede afectar de forma importante a la eficiencia del proceso de búsqueda.

La ordenación de las variables para su instanciación puede ser *estática* o *dinámica*. En la ordenación estática, el orden de las variables se determina antes de que la búsqueda comience. En la ordenación dinámica, la elección de la siguiente variable a instanciar se determina según el estado actual de la búsqueda.

Se han propuesto distintas heurísticas para determinar el orden en el que las variables son seleccionadas para su instanciación. La más común es la conocida como principio "*first-fail*". Mediante este método, la variable que tenga el menor número de valores alternativos es la que se escoge para siguiente instanciación. Así, el orden de instanciación de las variables puede ser diferente en distintas ramas del árbol de búsqueda, y es determinado dinámicamente. Con este método, lo que se pretende es que las posibles inconsistencias se detecten lo antes posible en cuanto al nivel de profundi-

dad en el árbol, de forma que se consiga un ahorro lo mayor posible en el número de nodos visitados.

Otra heurística consiste en escoger aquella variable que participa en el mayor número de restricciones. De esta forma, también se intenta que las posibles inconsistencias se detecten lo antes posible. Esta estrategia puede ser usada en conjunción con la anterior, cuando las variables tienen el número de valores alternativos.

Con respecto a la ordenación estática, se puede utilizar la siguiente heurística: se escogerá aquella variable que mantenga el mayor número de restricciones con las variables anteriores en la ordenación.

La ordenación de los *valores* también puede tener un fuerte impacto en la eficiencia de la búsqueda. En un caso extremo, si el CSP sólo tuviera una solución, y los valores correspondientes a las variables se escogieran en primer lugar, la solución sería encontrada sin *backtracking*. Una posible heurística sería preferir aquellos valores que maximizaran el número de opciones disponibles para las siguientes instanciaciones [Haralick and Elliot, 1980]. La forma de evaluar esta medida podría ser el producto de los tamaños de los dominios de las variables que restan por instanciar.

Otra heurística consistiría en escoger el valor que conduce a un CSP más simple de resolver. Esto requiere una estimación de la dificultad para resolver un CSP. En [Detcher and Pearl, 1988] se propone un método para convertir un CSP en otro estructurado en forma de árbol eliminando el menor número de restricciones y resolviéndolo para obtener todas sus soluciones. El número de ellas se toma como medida de la dificultad de resolver el CSP original.

5.2.6 Problemas de optimización con satisfacción de restricciones

Hay muchas situaciones en las que no basta con encontrar una solución al problema planteado, sino que se busca la mejor solución, según una medida dada a través de una función objetivo. Por tanto, el problema viene determinado por un conjunto de restricciones que deben satisfacerse y por una función de optimización que evalúe la bondad de cada solución, correspondiente a una asignación total sobre las variables.

El método más utilizado para encontrar soluciones óptimas a problemas de optimización es el de *Branch and Bound* (B&B). Este método necesita una función heurística que estime la calidad de cada asignación parcial de las variables. La función heurística debe representar una subestimación (en el caso de minimización) de la mejor solución que pueda obtenerse a partir de la asignación parcial correspondiente. El algoritmo realiza la búsqueda de forma similar a como lo hace el algoritmo de *backtracking*, pero al realizar una asignación, se evalúa la función heurística y se compara con el límite que se tenga para la mejor solución. Ese límite podrá corresponderse con el valor de la función objetivo de la mejor solución encontrada hasta el momento, o con alguna sobreestimación de la solución óptima. Aparte de la poda asociada a las posibles inconsistencias de las asignaciones parciales, también se producirá en el caso de que el valor correspondiente a la función objetivo exceda del límite.

La eficiencia del algoritmo de *branch and bound* viene determinada por dos factores. Por un lado, por la calidad de la función heurística usada para estimar las asignaciones parciales, y por el otro, por si se encuentra pronto un buen límite del valor óptimo. Normalmente, la parte más costosa computacionalmente del algoritmo es la parte final, correspondiente a las últimas mejoras de la función objetivos, cada vez más lentas, y a la verificación de la optimalidad de la última solución encontrada. De esta forma, estos algoritmos pueden ser usados para encontrar soluciones sub-óptimas, con mayor eficiencia cuanto mejores sean las heurísticas utilizadas.

5.2.7 Problemas de satisfacción de restricciones temporales

Los problemas de planificación y *scheduling*, como el tratado en esta tesis, pueden modelarse a través de redes de restricciones temporales [Detcher, *et al.*, 1991], representando a los llamados TCSP (problemas de satisfacción de restricciones temporales), en donde las variables se corresponden con puntos en el tiempo, y la información temporal viene representada por restricciones unarias y binarias que especifican los intervalos de tiempo permitidos para las variables.

Se distinguen en principio dos tipos de TCSP, según su complejidad. Cuando sólo se admite un intervalo entre cada dos puntos temporales, estamos ante problemas temporales simples (SPT), que pueden ser resueltos en un tiempo polinómico. El caso general viene definido a través de restricciones disyuntivas, correspondientes a distintos intervalos para las variables temporales. Sólo en determinados casos de disyunciones, el problema es tratable [Cohen, *et al.*, 1993], no correspondiéndose con los llamados problemas de *scheduling* disyuntivo.

La primera causa de disyunciones en los problemas de *scheduling* viene dada por el uso de recursos compartidos que sólo pueden ser usados de forma simultánea por una actividad. Cuando dos actividades, que requieren para su ejecución el uso exclusivo de un mismo recurso, no vienen ligadas a través de restricciones de precedencia, se genera una disyunción correspondiente a los dos posibles órdenes en que pueden ser ejecutadas, una antes que la otra o viceversa.

El tratamiento que suele darse a las restricciones disyuntivas pasa por usar algoritmos aproximados para detectar inconsistencias [Schwalb and Detcher, 1997], de forma que ayuden a mejorar la eficiencia de los algoritmos de búsqueda. Para evitar en la medida de lo posible la ramificación en el proceso de búsqueda, suelen aplazarse hasta que exista información suficiente que pudiera resolverlas en favor de alguna de las alternativas. Cuando se hace inevitable la búsqueda mediante *backtracking*, se intenta minimizar el tamaño del árbol de búsqueda ordenando previamente las disyunciones según los tamaños de los dominios [Stergiou and Koubarakis, 2000] y utilizando técnicas de consistencia de apoyo a algoritmos FC y derivados. En [Alfonso y Barber, 1999] se propone un método incremental en donde, conforme se van añadiendo restricciones, se efectúan procesos de clausura (consistencia) que restringen el límite del valor óptimo, utilizando distintas heurísticas para limitar el conjunto de opciones a considerar y reduciendo el número de *backtrackings*.

De entre los métodos más utilizados para detectar inconsistencias y reducir los dominios de las variables temporales se encuentran las conocidas como técnicas de *Edge Finding* [Baptiste and Le Pape, 1996], que razonan acerca del orden en que varias actividades pueden ser ejecutadas sobre un recurso común. El tipo de resultados que se obtienen de estos algoritmos es que alguna actividad no puede ser la primera (o la última) en ejecutarse de entre un grupo de ellas, y de acuerdo con ello, los límites de los dominios correspondientes a los tiempos de ejecución son revisados.

Otra fuente de restricciones disyuntivas es la consideración de tareas alternativas para la construcción de la solución. En [Beck and Fox, 2000] se propone una representación basada en restricciones para modelar problemas en los que se consideran tanto recursos alternativos como actividades alternativas.

5.2.8 Plataformas de implementación

La Programación con Restricciones surgió a partir de la Programación Lógica [Jaffar and Lassez, 1987] mediante el uso de las restricciones para reducir el espacio de búsqueda de soluciones. La Programación Lógica presenta un modelo relacional que facilita la formulación de problemas de tipo combinatorio, así como la expresión de restricciones. Su indeterminismo permite liberar al programador de la búsqueda en árboles, usando mecanismos de *backtracking* que vienen incorporados en su motor de inferencia, en conjunción con técnicas de resolución y unificación. Sin embargo, los lenguajes de Programación Lógica se comportan de manera muy ineficiente al resolver problemas combinatorios debido a que usan las restricciones para verificar si las soluciones generadas las cumplen, aparte del comportamiento patológico de los esquemas de *backtracking* simples.

Los lenguajes de Programación Lógica con Restricciones (CLP) introdujeron el concepto de dominio asociado a las variables frente a la unificación de la Programación Lógica, de manera que era posible aplicar técnicas de consistencia que mejoraran la eficiencia de búsqueda [Van Hentenryck, 1989] [Jaffar and Maher, 1994]. Existen muchas plataformas basadas en CLP, casi todas basadas en el lenguaje Prolog, al que se le ha incorporado un módulo de resolución de restricciones.

Sin embargo, los lenguajes basados en CLP tienen un problema importante. El control del programa está muy ligado a los algoritmos de *backtracking*, resultando muy difícil incorporar otras estrategias de búsqueda. Existen otras plataformas, basadas en otros lenguajes, que facilitan este aspecto. Así, ILOG Solver es un conjunto de bibliotecas sobre C++ que implementan de una forma muy flexible tanto algoritmos de consistencia como de control del programa. Además, se trata de una plataforma abierta, con la posibilidad de definir nuevos algoritmos de propagación de restricciones. Para resolver distintas clases de problemas, se dispone de bibliotecas específicas. Por ejemplo, ILOG Scheduler incluye distintos componentes para resolver problemas de *scheduling*.

El lenguaje CLP(R)

CLP(R) [Jaffar, *et al.*, 1992] es un lenguaje diseñado como una instancia del esquema de CLP(X) definido en [Jaffar and Lassez, 1987], basado en el lenguaje Prolog, y en donde se usa el dominio de los reales. Una importante propiedad de este lenguaje es que las restricciones son tratadas de manera uniforme, en el sentido de que son usadas para especificar los parámetros de entrada al programa y los resultados del mismo. Además, son las únicas primitivas usadas en la ejecución de los programas. La computación comienza con un objetivo y un conjunto de restricciones, inicialmente vacío. El modelo operacional incluye la estrategia de selección de subobjetivos de izquierda a derecha, como en Prolog, mediante la cual van seleccionándose también las restricciones. Cuando una restricción es seleccionada, es añadida al conjunto de restricciones del problema, determinándose si pueden dar lugar a inconsistencias. Se utiliza un algoritmo basado en el *Simplex* para resolver las restricciones lineales, así como un módulo de eliminación de restricciones superfluas. El sistema usa además un mecanismo para retrasar las restricciones no lineales, hasta que la instanciación de variables las convierta en lineales.

5.3 Adaptación del Problema a CLP(R)

El problema a resolver puede representarse a través de una base de hechos que recoja la información relativa al grafo *And/Or*, los recursos y duraciones estimadas para cada tarea, las combinaciones permitidas para máquinas y herramientas, las duraciones de los posibles cambios de herramientas y de los movimientos de los submontajes intermedios entre la máquina donde se formó y aquella en la que va a ser usado para construir otro. En la Figura 5.1 se muestra la representación del grafo *And/Or* de la Figura 2.6. Se utilizan dos relaciones. Por un lado, `nodoOr(SA,L)` indica que `L` es la lista de tareas alternativas para construir el submontaje `SA`. La relación `nodoAnd(T,SA1,SA2)` indica que los submontajes `SA1` y `SA2` son tomados por la tarea `T` para obtener el nuevo submontaje.

Como ya se ha indicado, el grafo *And/Or* recoge las restricciones de precedencia[6] entre las distintas tareas. A partir de las relaciones `nodoOr` y `nodoAnd` puede definirse la relación `prec(T,L1,L2)`, donde `L1` y `L2` representan las listas de tareas correspondientes a los nodos *Or* izquierdo y derecho respectivamente enlazados con la tarea `T`. De esta relación debe interpretarse que, en el caso de que la solución contenga a la tarea `T`, también incorporará a una y sólo una tarea de cada una de las listas `L1` y `L2`, si éstas no son la lista vacía (lo cual indicaría que la tarea `T` añade una pieza a un submontaje). Además, lógicamente, se extraerá la propia relación de precedencia entre `T` y la(s) tarea(s) seleccionada(s) de `L1` y `L2`.

[6] Recuérdese que se está considerando el problema opuesto, el desmontaje (véase el Capítulo 2)

```
nodoOr(abcde,[t1,t2]).
nodoOr(abcd,[t3,t4]).
nodoOr(acd,[t5,t6]).
nodoOr(ab,[t7]).
nodoOr(ac,[t8]).
nodoOr(ad,[t9]).
nodoOr(cd,[t10]).
nodoOr(be,[t11]).
nodoOr(a,[]).
nodoOr(b,[]).
nodoOr(c,[]).
nodoOr(d,[]).
nodoOr(e,[]).

nodoAnd(t1,abcd,e).
nodoAnd(t2,acd,be).
nodoAnd(t3,ab,cd).
nodoAnd(t4,b,acd).
nodoAnd(t5,ac,d).
nodoAnd(t6,c,ad).
nodoAnd(t7,a,b).
nodoAnd(t8,a,c).
nodoAnd(t9,a,d).
nodoAnd(t10,c,d).
nodoAnd(t11,b,e).
```

Figura 5.1: Representación en CLP(R) del grafo *And/Or* de la Figura 2.6.

La solución al problema se va a representar mediante una secuencia de tareas por cada máquina, formando una lista. Por razones de eficiencia en el tratamiento de listas, aparecen al principio las tareas añadidas a la solución en último lugar. Para cada tarea se incluye su identificación, herramienta usada y los tiempos de inicio y finalización de la misma, expresadas mediante el término `t(IdTarea, Herramienta, TIni, TFin)`. Obsérvese que los datos relativos a los tiempos son valores numéricos que estarán involucrados en las restricciones que se planteen para ellos, lo cual conllevará que para la solución (o soluciones) obtenida esos valores estén totalmente determinados para las tareas sin holguras o restringidos a un rango de valores para las tareas que presenten holguras. Este aspecto es uno de los que nos beneficiamos al resolver el problema mediante CLP(R), ya que aparte de ofrecernos la solución en forma de secuencias de tareas, se nos informa de los valores asociados a los tiempos de comienzo y finalización en forma de restricciones cuando para alguna tarea se permite un intervalo de comienzo de la misma.

Las restricciones entre los tiempos asociados a las tareas vienen asociadas a dos circunstancias: las relaciones de precedencia definidas por el grafo *And/Or* y la utilización de recursos comunes por distintas tareas. Esto último es fuente de parte del carácter combinatorio del problema, ya que en general habrá que resolver restricciones de tipo

disyuntivo: si no existe restricción de precedencia entre dos tareas y ambas usan el mismo recurso, hay que decidir cuál de ellas se realiza antes. Cualesquiera de las dos alternativas son en principio válidas, lo cual obliga a contemplarlas por separado para la búsqueda de la solución óptima. Esta problemática ya ha sido abordada en problemas de secuenciamiento similares utilizando técnicas de propagación de restricciones, normalmente retrasando el tratamiento de las restricciones disyuntivas y utilizando técnicas de *Edge Finfing* [Esquirol, *et al.*, 1996] [Caseau and Laburthe, 1995],.

Pero existe otro factor que dota al problema de un carácter combinatorio: para cada nodo *Or* pueden existir varias tareas alternativas que permitan completar el montaje del grupo de piezas asociadas a él. Se trata en definitiva de la propia selección de las tareas que van a componer la solución. Para este tipo de disyunciones no existen tratamientos similares, debido a que ni siquiera puede hablarse de una estructura única del árbol de tareas, ya que para un nodo Or determinado, una tarea podría descomponerlo en dos submontajes con parecido número de piezas, y otra tarea podría hacerlo separando una única pieza del resto. La inexistencia de esa estructura única del árbol de tareas impediría en su caso utilizarla como plantilla donde sustituir las tareas elegidas, así como los valores asociados a los tiempos de comienzo y finalización de las mismas para las variables temporales asociadas con la plantilla.

Así las cosas, cabe plantear la resolución del problema propuesto de dos maneras: (a) extraer todos los árboles de tareas del grafo *And/Or* y resolver el problema planteado para cada uno de ellos, con lo que nos serviríamos de las técnicas conocidas para el *scheduling* disyuntivo; o (b) utilizar como base esquemas de *branch and bound* en los que en los pasos de ramificación se consideren como alternativas aquellas que nos vayamos encontrando en la exploración consiguiente del grafo *And/Or*. Con este último enfoque se pretendería aprovechar los cálculos asociados a las tareas situadas cerca de la raíz del grafo *And/Or*, que serían comunes a muchas de las soluciones posibles. En este trabajo se desarrollan algoritmos que siguen esta última línea.

5.4 Resolución mediante *backtracking*

El esquema más fácil de implementar en un lenguaje de programación lógica con restricciones es obviamente un esquema de *backtracking*. Como se ha indicado en la sección anterior, la solución se compone de una secuencia de tareas por cada máquina. El tratamiento de la solución que realizarán los algoritmos desarrollados en este trabajo requerirá que el orden que siguen las tareas en las secuencias se corresponda con el orden de ejecución de las mismas. En cada paso de expansión se elige una tarea para incluirla en el plan de ensamblaje. Las tareas candidatas a ser incluidas en la solución serán aquellas cuyas predecesoras en el grafo *And/Or* hayan sido ya incluidas. De esta forma, las restricciones de precedencia de tareas definidas en el grafo *And/Or* se cumplen en cuanto al orden de inclusión de las tareas en la solución. Por otro lado, la decisión del orden de inclusión de tareas en la solución se hace corresponder con la decisión del propio orden de ejecución de las tareas que usan la misma máquina para su

```
Algoritmo planBT dev PlanOptimo
   PlanParcial := [ [ ], ..., [ ] ]
   PlanOptimo := indeterminado
   TFinalOptimo := ∞
   LRamasCand := [rama con la lista de tareas del nodo raíz del grafo And/Or]
   completarPlanBT (PlanParcial, LRamasCand)
   devolver PlanOptimo

Algoritmo completarPlanBT (PlanParcial, LRamasCand)
   si RCand = ∅
      TFinal := calcularTiempos (PlanParcial)
      si TFinal < TFinalOptimo
         < PlanOptimo, TFinalOptimo > := < PlanParcial, TFinal >
      fsi
   si no,
      desde i := 1 hasta numRamas (LRamasCand)
         < Rama, RestoRamas > := escogerRama (LRamasCand, i)
         desde j := 1 hasta numTareas (rama)
            Tarea := escogerTarea (Rama, j)
            incluirTareaPlan (Tarea, PlanParcial)    (* añade restricciones *)
            LRamasCandN := incluirSucesoresEnCandidatos (Tarea, RestoRamas)
            completarPlanBT (PlanParcial, LRamasCandN)
            eliminarTareaPlan (Tarea, PlanParcial)  (* elimina restricciones *)
         fdesde
      fdesde
   fsi
```

Figura 5.2: Algoritmo de *backtracking*.

ejecución. Éste es pues el tratamiento que se hace para este tipo de restricciones disyuntivas.

En la Figura 5.2 se muestra el algoritmo de *backtracking* utilizado. Durante la búsqueda, se mantiene una lista con las ramas a explorar, *LRamasCand*. Cada elemento de la misma contiene la información asociada a un nodo *Or*, básicamente la lista de tareas alternativas que pueden escogerse para construir el submontaje asociado. Nótese que deberá escogerse exactamente una tarea de cada rama para completar la solución final. Las distintas alternativas que pueden escogerse vienen dadas a través de las llamadas a *escogerTarea*. Por otro lado, las distintas llamadas a *escogerRama* tienen como objetivo producir distintos ordenamientos de las tareas correspondientes a

distintas ramas del grafo *And/Or*, que, en el caso de compartir recursos comunes, formarán soluciones distintas al problema. Sin embargo, se realizará un trabajo redundante en el caso de tareas que utilicen recursos diferentes. El algoritmo añade y elimina de forma dinámica las restricciones temporales correspondientes a las tareas que se van considerando en la solución.

Aparte de la manera de construir la solución, ésta se completa tras la resolución del conjunto de restricciones temporales asociadas a los tiempos de comienzo y finalización de las tareas que componen la solución. En la Figura 5.3 se muestra la forma en que se añaden estas restricciones mediante la definición en CLP(R) de la relación `incluirTareaPlanMaquina`. Esta relación es usada para construir la secuencia de tareas de ensamblaje para cada máquina, y a ella se llegará cada vez que una tarea deba ser incorporada a la solución, a través de la relación `incluirTarea-Plan`. Al incluir una tarea `T` en la solución, se plantea por un lado la restricción de precedencia con respecto a su tarea predecesora en al grafo *And/Or*. El tiempo de finalización de ésta, `TFinPred`, se usa para acotar el tiempo de inicio de la tarea añadida, `TIni`. Además, se hace uso del posible retardo asociado al eventual transporte del submontaje intermedio entre la máquina donde se construyó (`M`, usada por `T`) y aquella en que se necesita para construir otro (`MPred`, usada por la predecesora de `T`). La relación `movimientoSubmontaje` recoge la información correspondiente a estos

```
% incluirTareaPlanMaquina(T,D,M,H,TFin,TFPred,MPred,SA,LM,LM1):
%    incluye la tarea T en la secuencia LM de la máquina M,
%    obteniéndose la nueva secuencia LM1;
%    D es la duración de T
%    H es la herramienta usada por T
%    TFin es el tiempo final de T
%    TFPred es el tiempo final de la tarea predecesora de T
%    MPred es la máquina usada por la tarea predecesora de T
%    SA es el submontaje obtenido tras la ejecución de T

incluirTareaPlanMaquina(T,D,M,H,TFin,TFPred,MPred,SA,
                        [],[t(T,H,Tini,TFin)]) :- !,
  movimientoSubmontaje(SA,M,MPred,TMov),  % 0 si M=MPred
  TIni >= TFPred + TMov,
  TFin = TIni + D.

incluirTareaPlanMaquina(T,D,M,H,TFin,TFPred,MPred,SA,
                        [t(T1,H1,TIni1,TFin1)|R],
                        [t(T,H,TIni,TFin),t(T1,H1,TIni1,TFin1)|R]) :-
  movimientoSubmontaje(SA,M,MPred,TMov),  % 0 si M=MPred
  TIni >= TFPred + TMov,  % precedencia + movimiento submontaje
  cambioHerramienta(M,H,H1,DCH),  % 0 si H=H1
  TIni >= TFin1 + DCH,    % recurso compartido + cambio herramienta
  TFin = TIni + D.
```

Figura 5.3: Inclusión de restricciones temporales.

```
planBT(NodoOrRaiz,Plan,TFinal) :-
  numeroMaquinas(NMaq),
  creaPlanInicial(NMaq,L),
  nodoOr(NodoOrRaiz,LTareasInicio),
  completarPlanBt(L,Plan,[r(0,null,null,LTareasInicio)]),
  obtenerTFinal(Plan,TFinal).

% completarPlanBT(L,Plan,RCand)
%   completa el plan parcial L con las tareas seleccionadas de RCand

completarPlanBT(L,L,[]) :- !.
completarPlanBT(LM,Plan,RCand):-
  escogerRama(Rcand,Rama,RestoRamas),
  escogerTarea(Rama,T,TFPred,MPred,SAPred),
  incluirTareaPlan(T,TFin,Maq,SA,TFPred,MPred,SAPred,LM,LM1),
  incluirSucesoresEnCandidatos(T,TFin,Maq,SA,RestoRamas,RCand1),
  completarPlanBT(LM1,Plan,RCand1).

obtener_TFinal([],TFinal) :- minval(TFinal).
obtener_TFinal([[]|R],TFinal) :-
  obtener_TFinal([R],TFinal).
obtener_TFinal([[t(_,_,_,TFin)|_]|R],TFinal) :-
  TFinal >= TFin,
  obtener_TFinal(R,TFinal).
```

Figura 5.4: Algoritmo de *backtracking* en CLP(R).

retardos. Por otro lado, el tiempo de comienzo de la nueva tarea añadida a la solución viene también acotada por el tiempo final de la última tarea añadida a la secuencia de la misma máquina más el posible retardo debido al cambio de herramienta que pudiera necesitarse. La relación `cambioHerramienta` recoge la información correspondiente a estos retardos.

En la Figura 5.4 se muestra el algoritmo de *backtracking* propuesto, en CLP(R). En el programa, la relación `planBT` inicializa la solución y usa el algoritmo recursivo `completarPlanBt` para obtener las distintas soluciones al problema. Para el cálculo del tiempo total acumulado para el plan de montaje (relación `obtener_TFinal`), se añaden restricciones adicionales (tiempo final mayor o igual que el acumulado en cada una de las máquinas) y se busca el menor valor que haga satisfacer todas las restricciones acumuladas, a través de la relación `minval`.

Para determinar la siguiente tarea a incluir en la solución existen en principio dos puntos en donde escoger entre distintas alternativas: la elección del nodo *Or* a explorar y por otro lado la elección de la tarea (nodo *And*) para el nodo *Or* seleccionado. Esos puntos de retroceso para buscar soluciones alternativas se corresponden en el programa con las relaciones `escogerRama` y `escogerTarea` respectivamente. Así, la relación `esgogerRama` devuelve cada vez una rama (nodo *Or*) distinta, correspondiendo al tratamiento de las restricciones disyuntivas asociadas al uso de recursos

compartidos, mientras que la relación `escogerTarea` permite obtener cada vez una tarea distinta, correspondiendo al tratamiento de las disyunciones asociadas a las tareas alternativas.

Las tareas *candidatas* a ser incluidas en la solución en cada paso de expansión del algoritmo se recogen en una lista cuyos componentes (referidos aquí indistintamente como ramas o nodos *Or*) son términos de la forma `r(TFinPred,MaqPred,SA, ListaIdTareas)`, donde `TFinPred` se refiere al tiempo final de la tarea predecesora, `MaqPred` es la máquina usada por la tarea predecesora, `SA` es el submontaje correspondiente al nodo *Or*, y `ListaIdTareas` es el conjunto de tareas asociadas al nodo *Or*. Los datos que aparecen de la tarea predecesora, así como el submontaje, se usarán para determinar las restricciones de precedencia y movimiento de submontajes para las tareas candidatas, como se mostró en la Figura 5.3. Una vez escogida una de las tareas y añadida a la solución se usa la relación `incluirSucesoresEnCandidatos` para actualizar la lista de ramas candidatas incorporando las tareas sucesoras de la última tarea incluida en la solución.

Inicialmente, el algoritmo parte de una sola rama, la correspondiente al nodo *Or* raíz (producto completo), y conforme se vayan escogiendo tareas, se incluirán en la lista de candidatas aquellas ramas ligadas a la tarea incluida. Se habrá completado el plan de montaje cuando la lista de ramas de tareas candidatas esté vacía.

5.5 Resolución mediante *branch and bound*

El esquema de *backtracking* presentado en la sección anterior da lugar, como bien es conocido, a un algoritmo muy ineficiente en general. La adaptación de ese esquema a otro de *branch and bound* permite mejorar su comportamiento debido a dos factores: en la fase de ramificación (*branch*) se puede utilizar algún criterio más *inteligente* a la hora de elegir el orden de expansión de entre las diferentes alternativas que encuentra; por otro lado, se puede hacer uso de los valores obtenidos de las soluciones ya generadas para evitar la búsqueda por caminos que no van a proporcionar soluciones mejores (fase de acotación –*bound*–).

La acotación de soluciones se puede implantar, en una primera aproximación, añadiendo simplemente restricciones del tipo `TFin < B` al referirse a los tiempos de finalización de las tareas incorporadas a la solución, siendo B el valor de la mejor solución encontrada hasta el momento.

El uso de funciones que estimen los valores de las soluciones que pueden obtenerse a partir de un punto de la expansión ayudará a podar más el espacio de soluciones en la búsqueda del óptimo. Esas mismas estimaciones pueden servir, además, para elegir con mejor criterio el orden en el que se deben explorar las diferentes alternativas. Todo esto, sin embargo, requerirá un coste computacional añadido que, en algunos casos, puede no ser despreciable.

5.5.1 Heurísticas

Si se pretende realizar estimaciones de las soluciones para eliminar aquellas alternativas que no den lugar a soluciones mejores, tales estimaciones deben ser obligatoriamente optimistas, es decir, deben indicar un límite inferior (ya que se trata de un problema de minimización) del valor de todas las soluciones a las que se puede llegar expandiendo esa solución parcial. Obviamente, ya se está realizando una estimación optimista con lo que ya tenemos, pero se está despreciando el efecto de las tareas que aún faltan por incluir en la solución, por lo que aquella estimación resultaría demasiado optimista. Cuanto más ajustado sea el límite impuesto por la estimación, mejor se comportará el algoritmo en cuanto a soluciones despreciadas, pero existirá un compromiso con el coste computacional que conlleve el cálculo de esas estimaciones más realistas o mejor informadas.

En este trabajo se exponen varias funciones *heurísticas* que realizan estimaciones optimistas de las soluciones. Las heurísticas son similares a las que se definieron en el Capítulo 4 correspondientes a la familia de h_1. Con objeto de minimizar el efecto negativo en el coste computacional, en el cálculo de dichas heurísticas se separa lo que pueda ser *pre*-calculado *fuera de línea* de aquello que deba esperar hasta la ejecución del algoritmo de *branch and bound.*

Heurística *hs*

La primera estimación significativa que vemos considera el tiempo mínimo necesario para ejecutar cada tarea *candidata* y sus sucesoras en el grafo *And/Or*, sin atender a las posibles relaciones disyuntivas por la utilización de recursos comunes entre tareas de distintas ramas del grafo. Por simplicidad, tampoco se tienen en cuenta los posibles cambios de herramientas que debieran producirse en algunos casos.

De esta forma, el tiempo mínimo para ejecutar una tarea T y sus sucesoras viene dado por la expresión:

$$hs(T) = dur(T) + \max(\min_{T_i \in Or_1}(hs(T_i)), \min_{T_j \in Or_2}(hs(T_j))) \tag{5.1}$$

en donde T_i y T_j se refieren a las tareas asociadas a los nodos *Or* enlazados con T por debajo, Or_1 y Or_2. La operación de mínimo se corresponde con la elección de la tarea más favorable del nodo *Or* (sólo se ejecuta una), mientras que la operación de máximo se corresponde con la rama (nodo *Or*) más cargada según la estimación (al menos una tarea de cada rama hay que escoger para completar la solución).

El cálculo de estos valores puede realizarse antes de iniciar el algoritmo de búsqueda de la solución óptima, ya que es independiente de la solución parcial a que se haya llegado.

La estimación a la que se llegaría si se añadiera la tarea *candidata* T a una solución parcial cuyos tiempos acumulados para cada máquina son $TFin(M_i)$ sería

$$TFinMin = \max\left(\max_{\text{maquinas}}\left(TFin(M_i)\right), hs(T) + \max\left(TIni(T), TFin(M(T)) + DCH\right)\right) \quad (5.2)$$

siendo $M(T)$ la máquina que necesita la tarea T y DCH la duración del cambio de herramienta (0 si no lo hubiere). Obsérvese que, además de los tiempos acumulados en las diferentes máquinas, se tienen en cuenta, a través de las restricciones acumuladas para $TIni(T)$, el tiempo de comienzo de la tarea T, las relaciones de precedencia definidas en el grafo *And/Or*.

Los cálculos anteriores permitirán excluir como candidatas a aquellas tareas para las que la solución tuviera un tiempo mínimo acumulado *TFinMin* no inferior al valor de la mejor solución encontrada hasta el momento. Además, si todas las tareas correspondientes a una misma rama han sido eliminadas, la solución parcial debe ser desechada y volver atrás en busca de otras soluciones alternativas.

Heurística *ht*

En la heurística anterior no se han tenido en cuenta los cambios de herramienta necesarios para ejecutar las tareas candidatas y sus sucesoras, ni los retardos asociados a los movimientos de los submontajes intermedios. El cálculo del tiempo mínimo necesario para ejecutar la tarea T y sus sucesoras se consigue adaptando la fórmula indicada antes para *hs*:

$$ht(T) = dur(T) + \max(\min_{T_i \in Or_1}(htc(T_i, T)), \min_{T_j \in Or_2}(htc(T_j, T))) \quad (5.3)$$

siendo

$$htc(T_i, T) = ht(T_i) + \max\left(\tau\left(T_i, M(T), H(T)\right), \Delta_{mov}\left(sa(T_i), M(T_i), M(T)\right)\right) \quad (5.4)$$

con $M(T)$ y $H(T)$ la máquina y la herramienta requeridas para la ejecución de la tarea T, respectivamente. El nuevo factor $\tau(T, M, H)$, similar al definido en el Capítulo 4, representa un tiempo adicional a la ejecución de T y sus sucesoras, que es debido a que, previo a la ejecución de T, la herramienta H se encuentra instalada en la máquina M. Recuérdese que Δ_{mov} se refiere al retardo debido al movimientos de submontajes entre distintas máquinas. El valor de τ viene dado ahora por:

$$\tau(T, M, H) = \begin{cases} \Delta_{cht}\left(M, H(T), H\right) & \text{si } M = M(T) \\ \max\left(0, \tau_1(T, M, H)\right) & \text{si } M \neq M(T) \end{cases} \quad (5.5)$$

siendo

$$\tau_1\left(T, M, H\right) = \max\left(\min_{T_i \in Or_1}\left(\tau_2\left(T, T_i, M, H\right)\right), \min_{T_j \in Or_2}\left(\tau_2\left(T, T_j, M, H\right)\right)\right) \quad (5.6)$$

y

$$\tau_2(T,T_i,M,H)=\tau(T_i,M,H)-(ht(T)-ht(T_i)) \tag{5.7}$$

Nótese que, en el caso $M \neq M(T)$, puede darse un valor positivo para τ ($\tau_1>0$) si el cambio de herramienta necesario para la ejecución de alguna tarea sucesora tiene una duración mayor que la de la propia tarea T.

Utilizando ahora la heurística *ht*, la estimación a la que se llegaría si se añadiera la tarea *candidata* T a una solución parcial cuyos tiempos acumulados para cada máquina son $TFin(M_i)$ sería similar a la de *hs*:

$$TFinMin=\max\left(\max_{\text{maquinas}}(TFin(M_i)),ht(T)+\max(TIni(T),TFin(M(T))+DCH)\right) \tag{5.8}$$

Nótese que ahora *DCH* coincide con el valor de $\tau(T,M(T),UH(M(T)))$, siendo $UH(M(T))$ la herramienta colocada en $M(T)$, correspondiente a la última tarea incluida en la solución parcial con esa máquina.

Los criterios para realizar la poda de soluciones serán evidentemente los mismos que para *hs*, sólo que ahora la estimación puede ser diferente.

Heurística *htm*

El uso de la función τ en la heurística *ht* sirvió para incorporar los retardos asociados a los cambios de herramientas. En las expresiones que la definían se requería que τ fuera no negativa, lo cual significaba que $ht(T) + \tau(T, M, H)$ representaba el tiempo mínimo necesario para ejecutar la tarea T y sus sucesoras, partiendo de que en la máquina M estaba colocada la herramienta H. Cuando M no era la máquina requerida por T, $\tau > 0$ indicaba que el cambio de herramienta necesario en M tardaba más que la propia duración de T, mientras que si esto no ocurría, teníamos $\tau = 0$, lo cual indica que no se requería tiempo adicional (a la ejecución de T) para realizar cambios de herramienta en la máquina M. Las expresiones correspondientes a τ_1 que se calculaban podían dar lugar a valores negativos, lo que significa que *sobraba tiempo* para realizar el cambio de herramienta que debía realizarse en su caso. Esto quiere decir que, permitiendo valores negativos para τ podríamos estimar el momento en el que, para la ejecución de T y sus sucesoras, se necesita la herramienta H en la máquina M. La función τ seguiremos necesitándola tal como está definida, ya que la restricción de ser no negativa está ligada a la precedencia de tareas. Necesitaremos, entonces, definir otra función, τ', en parecidos términos a τ, que sirva para estimar el momento en que se necesita una determinada máquina, teniendo instalada una determinada herramienta, para ejecutar la tarea T y sus sucesoras. De esta forma,

$$htm(T,M,H)=\max(ht(T)+\tau'(T,M,H),0) \tag{5.9}$$

indica el tiempo mínimo adicional al acumulado en la máquina M, si tiene instalada la herramienta H, para completar la ejecución de T y sus sucesoras. Por otro lado, $\tau'(T,M,H)$ viene definida por:

$$\tau'(T,M,H)=\begin{cases}\Delta_{cht}(M,H(T),H) & \text{si } M=M(T)\\ \max\left(\min_{T_i\in Or_1}\left(\tau'_2(T,T_i,M,H)\right),\min_{T_j\in Or_2}\left(\tau'_2(T,T_j,M,H)\right)\right) & \text{si } M\neq M(T)\end{cases} \quad (5.10)$$

siendo

$$\tau'_2(T,T_i,M,H)=\tau'(T_i,M,H)-(ht(T)-ht(T_i)) \quad (5.11)$$

Ahora la estimación del tiempo mínimo requerido por cualquier solución que provenga de añadir la tarea candidata T a una solución parcial en la que los tiempos acumulados para cada máquina sean $TFin(M_i)$ vendrá dada por

$$TFinMin=\max(TIni(T)+ht(T),TFinMinMaq) \quad (5.12)$$

siendo

$$TFinMinMaq=\max_{\text{maquinas}}\left(TFin(M_i)+htm(T,M_i,lastTool(M_i))\right) \quad (5.13)$$

donde $lastTool(M_i)$ es la última herramienta usada en la máquina M para esa solución parcial.

Como se puede apreciar, con esta heurística ampliamos la estimación a todas las máquinas, mientras que en las anteriores sólo se realizaba para la requerida por la tarea candidata en cuestión.

5.5.2 Orden de exploración de alternativas

Los valores obtenidos para las estimaciones de las soluciones que se alcanzarían a partir de cada tarea candidata no sólo pueden servir para podar el espacio de búsqueda, tal como se ha mostrado en el subapartado anterior, sino que además pueden ser usados como criterio para escoger el camino más prometedor para encontrar el óptimo con el menor esfuerzo computacional posible.

Sin olvidar que mantenemos la búsqueda en profundidad debido a las limitaciones (o facilidades) impuestas por la Programación Lógica, el objetivo es escoger una tarea candidata de entre las que nos hemos encontrado tras realizar el último paso de expansión. Según se vio en el esquema de *backtracking* de la Figura 5.4, debían realizarse dos elecciones, una para la rama (o nodo *Or*) y otra para la tarea de la rama seleccionada.

El criterio de elección que se ha utilizado es parecido guarda similitud con el que se toma normalmente para variables y valores en los problemas de satisfacción de restricciones [Haralick and Elliot, 1980]. Para éstos se escoge en primer lugar la variable más restringida, es decir, la que puede hacer fracasar más fácilmente la búsqueda de la solución, con objeto de que los fracasos se detecten lo antes posible. Por otro lado, el valor que se escoge para la variable es el más prometedor, es decir, aquel que tenga más posibilidades para que se satisfagan todas las restricciones. En nuestro caso, las ramas harían las veces de las variables, y las tareas de cada rama serían los valores. De esta forma, en `escogerRama` se da prioridad a las ramas más *críticas*, es decir, las que tienen una estimación superior del valor de la solución. Por otro lado, en `escogerTarea`, para una determinada rama se comienza por las tareas más prometedoras, aquellas con una estimación inferior. De este modo, las tareas candidatas pertenecientes a una misma rama deben ser ordenadas en orden creciente de su estimación. Por otro lado, las ramas deben escogerse en orden decreciente de la mejor estimación existente en ellas.

5.6 Resultados y discusión

Los algoritmos presentados han sido implementados en CLP(R), que ofrece las facilidades de la Programación Lógica para la expresión de este tipo de problemas, así como para la formulación de algoritmos basados en *backtracking*.

Los resultados obtenidos no son, sin embargo, demasiado alentadores. En la Tabla 5.6 se recogen los resultados correspondientes a dos problemas de tamaño pequeño/mediano (15 piezas). El problema G15-MIN se corresponde con un grafo *And/Or* en el que no hay tareas alternativas, es decir, sólo existe un árbol de ensamblaje. En el problema G15-20%, el número de tareas alternativas para cada submontaje es aproximadamente el 20% del máximo número que habría en el caso de todas las

Tabla 5.6: Resultados comparativos.

Problema	Heurística	Nodos	Tiempo (ms)
G15-MIN	BT	167681	153480
	hs	39997	336860
	ht	30333	191800
	htm	29421	184630
G15-20%	BT	339262	323440
	hs	55703	480220
	ht	41084	339650
	htm	34132	265960

descomposiciones fuesen posibles. Aunque el número de nodos (soluciones parciales) generados para las heurísticas más avanzadas resultaba menor que para el esquema de *backtracking*, el coste computacional añadido para evaluar las estimaciones no compensaba el ahorro en el espacio de búsqueda explorado. Teniendo en cuenta que buena parte de la estimación se realiza *fuera de línea*, se encaminaron esfuerzos para mantener las estimaciones de etapas anteriores cuando fuera posible, así como evitar de esta forma volver a ordenar las tareas completamente. Aunque se produjeron progresos, no fueron lo suficientes como para mejorar el comportamiento en tiempo de computación del esquema de *backtracking* para los tamaños del problema señalados. Para tamaños superiores, para los que se prevé que pueda invertirse el comportamiento de los algoritmos, el tiempo de ejecución que resulta es desmesurado.

Cabe atribuir estos pobres resultados a distintas causas. Por una lado, y quizá de forma determinante, a la enorme ineficiencia que este tipo de lenguajes presenta para realizar determinados tratamientos sobre estructuras de datos, lo que ha podido dar lugar, por ejemplo, a un excesivo tiempo en la ordenación de las tareas candidatas. Por otro lado, las heurísticas usadas no parece que hayan sido lo suficientemente fuertes como para guiar de forma más efectiva la búsqueda. El uso de otras heurísticas complementarias, basadas en las que se han propuesto para el uso de recursos en el Capítulo 4 podría mejorar el comportamiento del algoritmo. Sin embargo, el esquema algorítmico usado, basado en *backtracking*, puede dificultar tales mejoras, y esto también es achacable a la plataforma utilizada, basada en CLP.

Así pues, las mejoras vendrían por una conjunción de modificaciones: una nueva plataforma de desarrollo, basada por ejemplo en el conjunto de bibliotecas de ILOG Solver y Scheduler sobre C++, lo cual facilitaría el uso de esquemas algorítmicos más flexibles y con nuevas funciones heurísticas que ayuden a guiar de forma más efectiva la búsqueda.

5.7 Conclusiones

En este capítulo se ha planteado la resolución del problema de planificación propuesto mediante técnicas de satisfacción de restricciones. Basándonos en el uso del lenguaje CLP(R), se han propuesto algoritmos de *branch and bound* para la obtención de la solución óptima al problema. Se obtiene como beneficio al aplicar estas técnicas una mayor facilidad al expresar la información asociada al problema y una mayor información dada por las soluciones, de manera que expresan, en forma de intervalos, los valores posibles para las variables temporales, indicando las holguras existentes para la ejecución de las tareas no críticas.

Se han identificado los distintos tipos de restricciones que definen el problema. Por un lado, las restricciones de precedencia entre las tareas, que vienen definidas a partir del grafo *And/Or*. Los retardos asociados a los cambios de herramientas y al movimiento de submontajes intermedios se incorporan con facilidad a las desigualdades asociadas a la ejecución secuencial de las tareas. Por otro lado, aparecen dos tipos de

restricciones disyuntivas en este problema: aquellas relacionadas con el uso compartido de recursos, y aquellas otras vinculadas a la comentada selección de tareas alternativas. Frente a la opción de enumerar todos los conjuntos de tareas que pueden dar lugar a la solución, y resolver cada problema generado por separado, usando estrategias conocidas de tratamiento de restricciones disyuntivas, aquí se proponen algoritmos que realizan un tratamiento conjunto de todas las restricciones disyuntivas mediante *backtracking*. Este enfoque aprovecha los cálculos asociados a las tareas situadas cerca de la raíz del grafo *And/Or*, que serían comunes a muchas de las soluciones posibles.

Para buscar un comportamiento mejor de estos algoritmos, se han propuesto distintas heurísticas encaminadas a estimar los valores de las soluciones que puedan obtenerse a partir de las soluciones parciales alcanzadas. Tales estimaciones se han usado para estrechar el espacio de búsqueda, así como para buscar una orden de exploración de alternativas más prometedor.

Los resultados obtenidos en CLP(R) parecen indicar que no se trata de la mejor plataforma para resolver este tipo de problemas de manera eficiente. Esto se debe por un lado a las limitaciones asociadas al esquema de *backtracking*, propio de los lenguajes de programación lógica, para controlar el proceso de búsqueda de soluciones. Por otro lado, el tratamiento de estructuras de datos es más ineficiente que en otro tipo de lenguajes de programación. El cambio de plataforma permitiría, entonces, una mayor flexibilidad para utilizar esquemas alternativos al *backtracking*, cuyos problemas son conocidos, así como la incorporación de nuevas heurísticas que ayuden a guiar el proceso de búsqueda.

Capítulo 6

Conclusiones y Trabajo Futuro

En este capítulo se discuten las principales conclusiones de esta tesis, y se identifican distintas líneas de investigación a seguir a partir del trabajo desarrollado.

6.1 Conclusiones

Se ha estudiado en esta tesis el problema de la selección óptima de secuencias de ensamblaje. Se trata de un problema de carácter combinatorio que implica una mayor complejidad que otros problemas ampliamente estudiados, como son el problema del viajante de comercio y el del secuenciamiento de tareas en múltiples máquinas (*Job Shop Scheduling Problem*). La obtención de una solución supone no sólo el secuenciamiento óptimo de las operaciones de ensamblaje, sino también la propia selección de las mismas dentro de un amplio rango de posibles planes de ensamblaje.

6.1.1 Modelo propuesto

Para la resolución del problema se ha propuesto un modelo original que considera la ejecución del proceso de ensamblaje en un entorno con múltiples estaciones de trabajo, de manera que pueden realizarse varias operaciones de ensamblaje asociadas al mismo producto de forma simultánea.

El problema resultante es un problema de planificación, definido a través de un grafo *And/Or*, una representación compacta del conjunto de secuencias de ensamblaje

factibles, que contiene las restricciones de precedencia existentes entre las operaciones de ensamblaje. También se incluyen restricciones de tipo *And/Or*, de forma que para obtener un plan de ensamblaje correcto, las operaciones que lo formen deben pertenecer todas a un mismo árbol dentro del grafo *And/Or*.

La definición del problema se completa con la información asociada al sistema de ensamblaje a emplear. Así, para cada tarea se supone una estimación de su duración y los recursos que necesita, máquina de ensamblaje y configuración de la misma, representada mediante el uso de una determinada herramienta. Teniendo en cuenta que el objetivo a optimizar es el tiempo total del ensamblaje, se han considerado en el modelo propuesto aquellos factores que pueden influir directamente en esa medida, y que dependen de la secuencia de operaciones que se establezca para el montaje. Por un lado, el tiempo correspondiente al cambio de configuración en las máquinas de ensamblaje, y por otro, el tiempo necesario para el transporte de submontajes intermedios entre distintas máquinas de ensamblaje.

La aportación más relevante que supone la propuesta realizada para el modelo es que los resultados pueden ser usados en varias etapas del proceso de planificación. La consideración del secuenciamiento final de las operaciones en el sistema de ensamblaje enriquece las fases previas de diseño, de manera que, considerando el mayor rango posible de opciones, se podrían detectar aquellas configuraciones con mayor y menor interés, potenciando el estudio de las primeras y rechazando las últimas. Teniendo en cuenta la dimensión del problema en estas primeras etapas, los algoritmos a utilizar no perseguirían la obtención de la solución óptima. Los algoritmos genéticos propuestos se ajustarían a tales requerimientos. En etapas posteriores, una vez que el problema se ha simplificado, el objetivo de encontrar el óptimo podría resultar más viable. Incluso, los algoritmos realizados pueden servir de base para el desarrollo de algoritmos que operen en línea durante la propia ejecución del proceso de ensamblaje.

6.1.2 Métodos de resolución empleados

Se han utilizado distintas técnicas algorítmicas para la resolución del problema propuesto, de naturaleza tanto determinista como no determinista. Como técnicas no deterministas se han usado los algoritmos genéticos, planteamiento muy empleado en la resolución de problemas de tipo combinatorio como el que nos ocupa. Como métodos deterministas, se han usado por un lado algoritmos A*, en donde el empleo de heurísticas bien informadas resultan cruciales para su buen comportamiento computacional en la búsqueda del óptimo. Por último, se ha propuesto también el uso de la programación con restricciones, de manera que el problema se visualiza como un problema de satisfacción de restricciones.

Algoritmos genéticos

En la aplicación de los algoritmos genéticos se han propuesto dos modelos de representación cromosómica de las soluciones. El primero de ellos se corresponde con una

representación binaria, lo que ha permitido utilizar los operadores genéticos estándar, aunque su efectividad queda condicionada por la calidad de la información genética asociada a esa representación, que podría resultar deficiente. La segunda de las representaciones propuestas, la representación simbólica, está más cercana a la estructura de las soluciones del problema. Sin embargo, han debido definirse para ella operadores genéticos específicos.

Para la representación simbólica se han considerado dos familias de operadores: una formada por algoritmos encaminados a cambiar la ordenación de las tareas de los individuos para formar otros, y otra familia de operadores para modificar tanto el orden como la composición de tareas en los individuos. La primera familia incluye incluye operadores que buscan nuevas secuencias localmente en un plan de montaje predeterminado, el correspondiente a los cromosomas padre. Estos operadores son similares a los usados para resolver los problemas de TSP y JSSP en la literatura. La otra familia de operadores introduce nuevas tareas en la solución, reemplazando a otras para mantener la validez de los cromosomas. Estos operadores tienen el propósito de buscar secuencias en otros planes de montaje. Estos últimos operadores, los más novedosos, deben involucrar necesariamente al grafo *And/Or* para la construcción de soluciones correctas. Se estudian distintas alternativas, desde aquellos algoritmos que, al introducir una nueva tarea en el plan intenta minimizar el número de tareas adicionales que deben introducirse para obtener un plan correcto, hasta aquellos otros que, a partir de una nueva tarea a introducir, intentan, con el menor coste computacional posible, construir un plan nuevo basándose mínimamente en el anterior.

La primera constatación resultante de la aplicación de algoritmos genéticos ha sido la gran dificultad de definir operadores de cruce eficientes para el problema propuesto. Las fuertes restricciones asociadas a la construcción de un plan de ensamblaje válido, significando que todas las operaciones de montaje deben pertenecer a un mismo árbol del grafo *And/Or*, dificultan enormemente la formación de individuos a partir de partes significativas de dos individuos cualesquiera de la población.

A pesar de ello, los resultados obtenidos para ambos modelos pueden considerarse satisfactorios. Los mejores resultados han correspondido al modelo de representación simbólica, como era de esperar de acuerdo a una correspondencia más directa con la estructura de las soluciones.

Algoritmos A*

En cuanto al uso de algoritmos A*, el mayor esfuerzo se ha dirigido hacia la definición de heurísticas que permitan guiar la búsqueda de manera eficiente. Se han definido varias heurísticas admisibles, partiendo de dos modelos relajados del problema. Dos funciones heurísticas básicas muestran la naturaleza de las restricciones tenidas en cuenta. Por un lado, en la heurística h_1 únicamente se consideran las restricciones de precedencia entre tareas, mientras que en la heurística h_2 sólo se consideran los tiempos totales de uso de cada recurso para la estimación del tiempo necesario para la ejecución de las tareas aún no introducidas en la solución. Ambas heurísticas son mejoradas por

otras que incorporan información asociada a las restricciones no consideradas inicialmente, dando lugar a sendas familias de heurísticas, cada una con un comportamiento diferente en cuanto a su influencia en el cálculo de la función objetivo.

En esas mejoras ha jugado un papel importante la definición de modelos basados en multiconjuntos parcialmente ordenados (*pomsets*) para una estimación más ajustada del número de cambios de herramientas. Se han considerado también las indeterminaciones en el número de usos no consecutivos de las herramientas que podrían generarse al analizar el árbol de precedencias. El esfuerzo realizado en la definición de los distintos modelos se ha visto compensado por los resultados obtenidos, bien directamente en su uso como heurística elegida, bien de forma indirecta a través de otras heurísticas que utilizaban sus resultados.

Como se refleja en los resultados obtenidos, un grupo de heurísticas dominará sobre el otro en función de los recursos definidos para el problema y de la estructura del grafo *And/Or*, como factores principales. Así, cuando se disponga de pocos recursos compartidos por muchas tareas, las heurísticas de tipo h_2 serán más relevantes, mientras que cuando se disponga de muchos recursos y/o la estructura del grafo *And/Or* sea poco equilibrada, las heurísticas de tipo h_1 adquieren más peso.

Para mejorar el comportamiento del algoritmo se ha definido, por un lado, la relación de compatibilidad entre tareas, lo cual ha permitido detectar simetrías del problema, así como ignorar partes importantes del espacio de búsqueda en donde no podía encontrarse la solución óptima. Por otro lado, se ha propuesto el uso de recorridos en profundidad para detectar nuevas cotas para la solución óptima, lo que permite un mejor aprovechamiento de la memoria disponible.

Los resultados obtenidos para los algoritmos A* muestran fundamentalmente el efecto de las distintas heurísticas definidas, y cómo una mejor en las mismas conduce a un mejor comportamiento del algoritmo. Ello se consigue, como ha podido observarse en el desarrollo de las heurísticas, a través de la incorporación de la información relevante del problema, lo cual no siempre resulta fácil.

Programación con restricciones

En cuanto al uso de la programación con restricciones, los principales beneficios que se obtienen son una fácil formulación del problema y una mayor expresividad de las soluciones obtenidas, que incluyen las holguras existentes para las variables temporales asociadas a la ejecución de las operaciones de ensamblaje, lo que permite un análisis más directo de las soluciones. La forma natural en que se expresan las restricciones que definen el problema permite una fácil modificación de las aplicaciones.

Se han definido modelos algorítmicos basados en *branch and bound*, para lo cual se han utilizado distintas heurísticas, similares a las definidas para los algoritmos A*. De la definición del problema propuesto se han identificado los distintos tipos de restricciones. Por un lado, las asociadas a las relaciones de precedencia entre tareas y a los retardos considerados por los cambios de herramientas y el transporte de submontajes intermedios. Por otro lado, las restricciones disyuntivas, debidas de una parte a la

utilización de recursos compartidos, y de otro a la propia estructura del problema, en el que pueden escogerse distintos planes de ensamblaje alternativos. El tratamiento realizado sobre las restricciones disyuntivas ha sido su incorporación dinámica durante el proceso de búsqueda, a través del recorrido del grafo *And/Or*.

Los pobres resultados obtenidos deben buscarse fundamentalmente en el uso de CLP(R) para la implementación de los algoritmos desarrollados, debido a las limitaciones asociadas al esquema de *backtracking*, propio de los lenguajes de programación lógica, para controlar el proceso de búsqueda de soluciones. Además, el tratamiento de estructuras de datos es mucho más ineficiente que en otros tipos de lenguajes de programación.

6.1.3 Resultados comparativos

En los capítulos anteriores se ofrecen resultados parciales para cada una de las técnicas empleadas en la resolución del problema propuesto. Para realizar una comparación entre las distintas técnicas es necesario destacar la distinta naturaleza de las mismas. Dejando de lado los resultados que se han obtenido mediante la programación con restricciones, por las razones expuestas anteriormente, la discusión se centra en la comparación de algoritmos genéticos frente a los algoritmos A*.

Mientras que los algoritmos genéticos son de tipo probabilista, no pudiéndose garantizar la obtención de la solución óptima en un tiempo limitado, los algoritmos A* sí pueden llegar a obtenerla teóricamente. Esto sugiere que para problemas pequeños y medianos, con heurísticas bien informadas y mecanismos adecuados de exploración de las distintas alternativas, los algoritmos A* pueden lograr el óptimo de una manera mucho más eficiente que los algoritmos genéticos. Sin embargo, ya en estos tamaños de problemas se observan grandes diferencias tanto en la memoria ocupada como en el tiempo de ejecución de los algoritmos A* según el ejemplar concreto que se esté resolviendo. Esto no ocurre con los algoritmos genéticos, que ofrecen un comportamiento más homogéneo.

Cuando el tamaño del problema crece, los tiempos de ejecución de ambos algoritmos para alcanzar soluciones óptimas o cercanas al óptimo crece exponencialmente, de acuerdo a la naturaleza del problema. Sin embargo, la memoria ocupada no lo hace del mismo modo. Para los algoritmos A* suponen uno de los más serios inconvenientes para su aplicación, ya que el número de nodos generados también crece de forma exponencial. Sin embargo, los algoritmos genéticos ocupan una cantidad de memoria proporcional al tamaño del problema, si se mantiene el mismo tamaño de la población.

De esta forma, el uso de los algoritmos genéticos puede resultar más adecuado para tamaños del problema elevados, y cuando no se persigue la obtención de la solución óptima, sino simplemente una buena solución, o una colección de buenas soluciones. Estas son las condiciones que suelen darse en las primeras etapas del diseño del producto a ensamblar y del propio proceso de ensamblaje, con muchas tareas alternativas y muchas combinaciones de recursos alternativos.

Por el contrario, para tamaños más moderados, y esto siempre dependerá de cada caso en concreto, y cuando se quiere encontrar la solución óptima, o el conjunto de ellas, el uso de algoritmos A* puede ser más prometedor, si están guiados por funciones heurísticas que recojan de manera adecuada la información relevante del problema.

Para salvar las deficiencias inherentes a los algoritmos A*, o desventajas frente a los algoritmos genéticos, por un lado pueden adecuarse para responder con soluciones tanto más buenas cuanto más tiempo se les permita para ello (algoritmos *any-time*), tal como se ha propuesto en este trabajo. En cuanto a la utilización de memoria, difícilmente puede lograrse un ahorro tan sustancial sin sacrificar la garantía de encontrar el óptimo, aunque, puestos a competir en igualdad de condiciones, los algoritmos genéticos tampoco garantizan tal resultado, y más cuando el tamaño del problema es muy elevado.

6.2 Trabajo futuro

Del trabajo desarrollado en esta tesis pueden derivarse distintas líneas de trabajo futuro. En cuanto al modelo del problema propuesto, se pueden señalar las siguientes:

- Incorporación al modelo de aspectos específicos del sistema de ensamblaje concreto sobre el que se quiere implementar el proceso de ensamblaje. Se ha considerado en particular que el sistema logístico de manipulación y transporte de piezas y submontajes es óptimo. En la realidad, podría ser necesario considerar las limitaciones asociadas a los distintos recursos, como almacenes de piezas y submontajes, alimentadores, dispositivos de fijación, robots móviles, etc.

- Considerar el ensamblaje de lotes de productos, y no sólo de productos individuales.

- Desarrollar algoritmos, basados en los aquí expuestos, que sirvan para controlar el proceso de ensamblaje en línea.

- Desarrollar algoritmos dirigidos a la obtención de planes óptimos de sustitución de piezas defectuosas.

- Extender el modelo de forma que las estimaciones de las duraciones asociadas a las tareas de montaje no tengan un valor fijo, sino un rango de posibles valores ligados a una función probabilística.

En cuanto a las técnicas de optimización utilizadas para la resolución del problema, hemos identificado las siguientes líneas de trabajo futuro:

- Mejora de los algoritmos genéticos, de forma que se consiga aprovechar mejor la información genética almacenada en los individuos. Dicha mejora debe venir por

un lado por la definición de nuevos modelos de representación cromosómica de las soluciones, bien modificando aspectos de los modelos aquí propuestos, o bien incorporando nuevos elementos de la solución en los cromosomas, como pudieran ser los submontajes involucrados en las operaciones de ensamblaje, las restricciones de precedencia entre las tareas, o los recursos utilizados por las mismas. Por otro lado, pueden obtenerse mejoras si en la aplicación de los operadores de cruce son seleccionados individuos que pudieran compartir información genética compatible para construir nuevas soluciones. Esto podría facilitarse a través de modelos de algoritmos genéticos paralelos.

- Profundizar en las modificaciones sobre los algoritmos A*, en la línea seguida en el presente trabajo, para mejorar el comportamiento computacional de los mismos. El problema más importante es el consumo de memoria. Deben buscarse métodos que reduzcan tal consumo, especialmente en lo que se refiere a la generación de los distintos árboles de precedencia de tareas. Por otro lado, el uso de heurísticas no admisibles puede ayudar a encontrar más rápidamente soluciones, si no óptimas, cercanas al óptimo. La combinación de ambos tipos de heurísticas puede ser interesante en la definición de algoritmos *any-time*.

- Detectar las condiciones en las que unas heurísticas tienen más influencia sobre otras, de forma que puedan escogerse de forma dinámica durante la ejecución del algoritmo A*. Esta propuesta está basada en los resultados obtenidos para las dos familias de heurísticas definidas en los algoritmos A*.

- Avanzar en la definición de modelos de satisfacción de restricciones que incorporen nuevos tipos de restricciones y métodos específicos de resolución, basados en las técnicas heurísticas utilizadas en los algoritmos A*.

- Utilización de plataformas más eficientes basadas en la programación con restricciones, como puede ser el conjunto de bibliotecas de ILOG (Solver, Scheduler), desarrolladas sobre C++.

- Combinar los algoritmos genéticos con los A*, de manera que éstos últimos, u otros basados en las heurísticas definidas para ellos, se incorporen como nuevos operadores genéticos o en la definición de la población inicial de individuos.

- Combinar los algoritmos genéticos con la programación con restricciones, de forma que la representación de los individuos incorpore distintos tipos de restricciones que sirvan para construir las soluciones correspondientes.

Bibliografía

[Ahn, *et al.*, 1993] J. Ahn, W. He, and A. Kusiak, Scheduling with Alternative Operations, *IEEE Transactions on Robotics and Automation*, Vol. 9, No. 3, pp. 297-303, 1993.

[Alfonso y Barber, 1999] M. I. Alfonso y F. Barber. Combinación de procesos de clausura y CSP para la resolución de problemas de scheduling. *VIII Conferencia de la Asociación Española para la Inteligencia Artificial CAEPIA'99*, Murcia, Noviembre 1999.

[Altenberg, 1995] L. Altenberg. The Schema Theorem and Price's Theorem. *Foundations of Genetic Algorithms 3*, Whitley L. D. and Vose M. D. (eds.), Morgan Kaufmann Publishers, pp. 23-49, 1995.

[Ames, *et al.*, 1995] A.L. Ames, T.L. Calton, R.E. Jones, S.G. Kaufman, C.A. Laguna and R.H. Wilson. Lessons Learned from a Second Generation Assembly Planning System. *Proceedings of the 1995 IEEE International Symposium on Assembly and Task Planning, ISATP-95*, pp. 41-47, 1995.

[Baker, 1987] J. E. Baker. Reducing bias and inefficiency in the selection algorithm. *Proceedings of the Second International Conference on Genetic Algorithms*, pp. 14-21, Grefenstette J. J. (ed.), Lawrence Erlbaum, 1987.

[Baldwin, *et al.*, 1991] D. F. Baldwin, T. E. Abel, M. M. Lui, T. L. De Fazio and D. E. Whitney. An integrated computer aid for generating and evaluating assembly sequences for mechanical products. *IEEE Transactions on Robotics and Automation*, Vol. 7, No. 1, pp. 78-94, February 1991.

[Baptiste and Le Pape, 1996] P. Baptiste and C. Le Pape. Edge-Finding constraint propagation algorithms for disjunctive and cumulative scheduling. *Proceedings of the Fifteenth Workshop of the U. K. Planning Special Interest Group*, Liverpool, 1996.

[Beck and Fox, 2000] J. C. Beck and M. S. Fox. Constraint-directed techniques for scheduling alternative activities. *Artificial Intelligence*, Vol. 121 (2000), pp. 211-250.

[Boneschanscher, 1993] N. Boneschanscher. *Plan Generation for Flexible Assembly Systems*. Ph. D. thesis Delft University of Technology, Delft, The Netherlands, 1993.

[Bourjault, 1984] A. Bourjault. *Contribution à une Approche Méthodologique de l'Assemblage Automatisé: Elaboration Automatique des Séquences Opératoires*. Thèse d'état, Université de Franche-Comté, Besançon, France, 1984.

[Calton, 1999] T. L. Calton. Advancing design-for-assembly. The next generation in assembly planning. *Proceedings of the 1999 IEEE International Symposium on Assembly and Task Planning*, pp. 57-62, Porto, Portugal, July 1999.

[Calton and Peters, 1999] T. L. Calton and R. R. Peters. A framework for aoutomating cost estimates in assembly processes. dvancing design-for-assembly. The next generation in assembly planning. *Proceedings of the 1999 IEEE International Symposium on Assembly and Task Planning*, pp. 100-105, Porto, Portugal, July 1999.

[Caseau and Laburthe, 1995] Y. Caseau and F. Laburthe. Improving Branch and Bound for Jobshop Scheduling with Constraint Propagation. Proceedings of the 8th Franco-Japanese 4th Franco-Chinese Conference CCS'95.

[Chakrabarty and Wolter, 1997] S. Chakrabarty and J. Wolter. A structure-oriented approach to assembly sequence planning. *IEEE Transactions on Robotics and Automation*, Vol. 13, No. 1, pp. 14-29, February 1997.

[Cohen, *et al.*, 1993] D. Cohen, P. Jeavons and M. Koubarakis. Tractable disjunctive constraints. *Lecture Notes in Computer Science*, 1330 (1997), pp. 478-490.

[Davis, 1991] L. Davis. *Handbook of Genetic Algorithms*. Van Nostrand Reinhold, 1991.

[De Fazio, *et al.*, 1990] T.L. De Fazio, T.E. Abell, G.P. Amblard, D.E. Whitney. Computer-aided assembly sequence editing and choice: Editing criteria, bases, rules, and technique. *Proceedings of the IEEE International Conference on Systems Engineering*, pp. 416-422, 1990.

[De Fazio, *et al.*, 1999] T. L. De Fazio, S. J. Rhee, and D. E. Whitney. Design-Specific Approach to Design for Assembly (DFA) for Complex Mechanical Assemblies. *IEEE Transactions on Robotics and Automation*, Vol. 15, No. 5, pp. 869-881, October 1999. Also, Corrections, Vol. 15, No. 6, pp. 1145, December 1999.

[De Fazio and Whitney, 1987] T.L. De Fazio and D.E. Whitney. Simplified Generation of All Mechanical Assembly Sequences. *IEEE Journal of Robotics and Automation*, Vol. 3, No. 6, pp. 640-658, 1987. Also, Corrections, Vol. 4, No. 6, pp. 705-708, 1988.

[Del Valle and Camacho, 1994] C. Del Valle and E. F. Camacho. Automatic assembly task assignment for a multirobot environment. *IFAC Conference on Integrated Systems Engineering*. Baden-Baden, September 1994.

[Del Valle and Camacho, 1996a] C. Del Valle and E. F. Camacho. Automatic assembly task assignment for a multirobot environment. *Control Engineering Practice*, vol. 4, pp. 915-921, July 1996.

[Del Valle y Camacho, 1996b] C. Del Valle and E. F. Camacho. Algoritmo A* para la determinación de secuencias óptimas de ensamblaje. *II Jornadas de Informática*. Almuñécar, Julio 1996.

[Del Valle y Camacho, 1997] C. Del Valle and E. F. Camacho. Optimización de secuencias de ensamblaje mediante algoritmos genéticos con codificación binaria. *VII Conferencia de la Asociación Española para la Inteligencia Artificial, CAEPIA 97*, pp. 115-124. Málaga, Noviembre 1997.

[Del Valle, *et al.*, 1998] C. Del Valle, P.L. Perejón and E. F. Camacho. Selection of optimal assembly sequences using genetic algorithms. *International ICSC Symposium on Engineering of Intelligent Systems, EIS 98.* Vol. 1, pp. 303-309, February 1998.

[Del Valle and Camacho, 1999a] C. Del Valle and E. F. Camacho. Binary vs symbolic chromosomal encoding in GA-based selection of assembly tasks. *3rd World Multiconference on Circuits, Systems, Communications and Computers*. Athens, July 1999.

[Del Valle and Camacho, 1999b] C. Del Valle and E. F. Camacho. Binary vs symbolic chromosomal encoding in GA-based selection of assembly tasks. *Advances in Intelligent Systems and Computer Science*, pp. 145-150. World Scientific and Engineering Society Press, 1999.

[Del Valle, *et al.*, 1999] C. Del Valle, E. F. Camacho y M. Toro. Búsqueda de secuencias óptimas de montaje mediante Programación Lógica con Restricciones. *IV Taller de Razonamiento Temporal TARRAT'99*, pp. 31-39. Murcia, Noviembre 1999.

[Detcher, *et al.*, 1991] R. Detcher, I. Meiri and J. Pearl. Temporal constraint networks. *Artificial Intelligence*, 49 (1991) 61-95.

[Detcher and Pearl, 1988] R. Detcher and J. Pearl. Network-based heuristics for constraint satisfaction problems. *Artificial Intelligence*, 34 (1) (1988) 1-38.

[Esquirol, *et al.*, 1996] P. Esquirol, H. Fargier, P. Lopez, T. Schiex. Constraint programming. Belgian Journal of Operations Research, Statistics and Computer Sciences, 1996.

[Freuder, 1982] E. C. Freuder. A sufficient condition for backtrack-free search. *Journal ACM*, 29 (1) 1982, 24-32.

[Garey and Johnson, 1979] M. R. Garey and D. S. Johnson. *Computers and intractability: A guide to the theory of NP-completeness*. W. H. Freeman, San Francisco, 1979.

[Gaschnig, 1979] *Performance measurement and analysis of certain search algorithms*. Tech. Rept. CMU-CS-79-124, Carnegy Mellon University, 1979.

[Gillies and Liu, 1995] D. W. Gillies and J. W. S. Liu. Scheduling tasks with And/Or precedence constraints. *SIAM Journal on Computing*, Vol. 24, No. 4, pp. 797-810, August 1995.

[Glover, 1989] F. Glover. Tabu search – part I. *ORSA Journal of Computing*, Vol 1, No. 3, pp. 190-206, 1989.

[Glover, 1990] F. Glover. Tabu search – part II. *ORSA Journal of Computing*, Vol 2, No. 1, pp. 4-32, 1990.

[Goldberg, 1989] D. E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley, 1989.

[Goldwasser and Motwani, 1999] M. H. Goldwasser and R. Motwani. Complexity measures for assembly sequences. *International Journal of Computational Geometry and Applications*, 9:371-418, 1999.

[Gottschlich, *el al.*, 1994] S. Gottschlich, C. Ramos, D. Lyons. Assembly and Task Planning: A Taxonomy. *IEEE Robotics and Automation Magazine*, Vol. 1, No. 3, pp. 4-12, 1994.

[Haralick and Elliot, 1980] R. M. Haralick and G. L. Elliot. Increasing tree efficiency for constraint satisfaction problems. *Artificial Intelligence*, Vol. 14 (1980), pp. 263-313.

[Hart, *et al.*, 1968] P.E. Hart, N.J. Nilsson and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, SSC-4, 2, pp. 100-107, 1968.

[Hart, *et al.*, 1972] P.E. Hart, N.J. Nilsson and B. Raphael. Correction to a formal basis for the heuristic determination of minimum cost paths. *SIGART Newsletter*, 37, pp. 28-29, 1972.

[Henrioud, 1989] J. M. Henrioud. *Contribution à la conceptualisation de l'assemblage automatisé: nouvelle approche en vue de détermination des processus d'assemblage*. Thèse d'état, Université de Franche-Comté, Besançon, France, 1989.

[Henrioud and Bourjault, 1991] J. M. Henrioud and A. Bourjault. LEGA: A Computer-Aided Generator of Assembly Plans. *Computer-Aided Mechanical Assembly Planning* (L. Homem de Mello, S. Lee editors), Kluwer Academic Pub, 1991, pp. 191-215.

[Holland, 1975] J. Holland. *Adaptation in natural and artificial systems*. University of Michigan Press, 1975.

[Holland, *et al.*, 1992] W. van Holland, N. Boneschanscher and W. F. Bronsvoort. Task assignment in a flexible assembly cell using And/Or graphs. *Proceedings of the 23rd International Symposium on Industrial Robots*, pp. 653-558, 642, Barcelona, October, 1992.

[Homem de Mello, 1989] J.M. Homem de Mello. *Task Sequence Planning for Robotic Assembly*. PhD Thesis, Carnegie Mellon University, May 1989.

[Homem de Mello and Sanderson, 1990] L.S. Homem de Mello and A.C. Sanderson. And/Or Graph Representation of Assembly Plans. *IEEE Transactions on Robotics and Automation*. Vol. 6, No. 2, pp. 188-199, 1990.

[Homem de Mello and Sanderson, 1991a] L.S. Homem de Mello and A.C. Sanderson. Representations of Mechanical Assembly Sequences. *IEEE Trans. Robotics Automat.* Vol. 7, No. 2, pp. 211-227, 1991.

[Homem de Mello and Sanderson, 1991b] L.S. Homem de Mello and A.C. Sanderson. A Correct and Complete Algorithm for the Generation of Mechanical Assembly Sequences. *IEEE Transactions on Robotics and Automation*. Vol. 7, No. 2, pp. 228-240, 1991.

[Homem de Mello and Sanderson, 1991c] L.S. Homem de Mello and A.C. Sanderson. Two Criteria for the Selection of Assembly Plans: Maximizing the Flexibility of Sequencing the Assembly Tasks and Minimizing the Assembly Time Through Parallel Execution of Assembly Tasks. *IEEE Transactions on Robotics and Automation*. Vol. 7, No. 5, pp. 626-633, 1991.

[Homem de Mello and Lee, 1991] L.S. Homem de Mello and S. Lee, eds. *Computer-Aided Mechanical Assembly Planning*. Kluwer Academic Publishers, 1991.

[Jaffar and Lassez, 1987] J. Jaffar and J. L. Lassez. Constraint Logic Programming. *Proceedings 14th ACM Symposium on Principles of Programming Languages*, pp. 111-119, Munich, January 1987.

[Jaffar, *et al.*, 1992] J. Jaffar, S. Michaylov, P.J. Stucley and R.H.C. Yap. The CLP(R) Language and System. *ACM Transaction on Programming Languages and Systems*. Vol.14, No. 3, pp. 339-395, July 1992.

[Jaffar and Maher, 1994] J. Jaffar and M.J. Maher. Constraint Logic Programming: A Survey. *The Journal of Logic Programming*: 19, 20, pp. 503-581, 1994.

[Jones and Wilson, 1996] R. E. Jones and R. H. Wilson. A survey of constraints in automated assembly planning. *Proceedings of the 1996 International Conference on Robotics and Automation*, pp. 1525-1532.

[Jones, *et al.*, 1997] R. E. Jones, R. H. Wilson and T. L. Calton. Constraint-based interactive assembly planning. *Proceedings of the 1997 International Conference on Robotics and Automation*, pp. 913-920.

[Kaufman *et al.*, 1996] S. G. Kaufman, R. H. Wilson, R. E. Jones, T. L. Calton, A. L. Ames. The Archimedes 2 Mechanical Assembly Planning System. *Proceedings of*

the 1996 International Conference on Robotics and Automation ICRA'96, pp. 3361-3368.

[Kavraki *et al.*, 1993] L. Kavraki, J. C. Latombe and R. H. Wilson. On the Complexity of Assembly Partitioning. *Information Processing Letters*. Vol. 48, pp. 229-235, 1993.

[Kavraki and Kolountzakis, 1995] L. Kavraki and M. Kolountzakis. Partitioning a planar assembly into two connected parts is NP-complete. *Information Processing Letters*. Vol. 55, pp. 156-165, 1995.

[Kirkpatrick, *et al.*, 1983] S. Kirkpatrick, C. Gellat and M. Vecchi. Optimization by simulated annealing. *Science*, 220 (4598): 671-680, 1983.

[Korf, 1985] R. Korf. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence*, Vol. 27, no. 1, pp. 97-109, 1985.

[Korf, 1990] R. Korf. Depth-limited search for real-time problem solving. *The Journal of Real-Time Systems*, 2, pp. 7-24, 1990.

[Kumar, 1992] V. Kumar. Algorithms for constraint satisfaction problems: a survey. *AI Magazine*, 13 (1): 32-44, 1992.

[Kusiak, 1990] A. Kusiak. *Intelligent Manufacturing Systems*. Prentice-Hall International Series in Industrial and Systems Engineering, 1990.

[Lawler, *et al.*, 1985] G. Lawler, J. K. Lenstra, A. Rinnooy Kan and D. B. Shmoys (eds.). *The Traveling Salesman Problem: A Guided Tour of Combinatorial Optimization*. Wiley, Chichester, 1985.

[Mackworth, 1977] A. K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8 (1977) 99-118.

[Mattfeld, 1996] D. C. Mattfeld. Evolutionary Search and the Job-Shop. Investigations on Genetic Algorithms for Production Scheduling. Physica-Verlag, Heidelberg, 1996.

[Michalewicz, 1992] Z. Michalevicz. *Genetic Algorithms + Data Structures = Evolutionary Programs*. Springer Verlag, 1992.

[Montanary, 1974] U. Montanary. Networks of constraints: fundamental properties and aplications to picture processing. *Information Sciences*, Vol. 7, pp. 95-132, 1974.

[Moradi, *et al.*, 1997] H. Moradi, K. Goldberg and S. Lee. Geometry-based part grouping for assembly planning. *Proceedings of the 1997 IEEE International Symposium on Assembly and Task Planning ISATP'97*, Marina del Rey, CA, August 1997.

[Nadel, 1988] B. Nadel. Tree search and arc consistency in constraint satisfaction algorithms. *Search in Artificial Intelligence*, Springer-Verlag, 1988.

[Nakano and Yamada, 1991] R. Nakano and T. Yamada. Conventional genetic algorithms for job shop problems. In R.K. Belew and L.B. Booker, editors. *Proceedings of the Forth International Conference on Genetic Algorithms, ICGA-91*, pp. 474-479. Morgan Kaufmann.

[Nilsson, 1980] N.J. Nilsson. *Principles of Artificial Intelligence*. Chenanso Forks, NY: Tioga, 1980.

[Pearl, 1984] J. Pearl. *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Reading, MA, Addison-Wesley, 1984.

[Pratt, 1986] V.R. Pratt. Modeling Concurrency with Partial Orders. *International Journal of Parallel Programming*. Vol. 15, No. 1, pp. 33-71, Feb 1986.

[Romney, *et al.*, 1995] B. Romney, C. Godard, M. Goldwasser, G. Ramkumar. An Efficient System for Geometric Assembly Sequence Generation and Evaluation. *Proceedings of the 1995 ASME International Computers in Engineering Conference*, pp. 699-712, 1995.

[Russell y Norvig, 1996] S.J. Russell y P. Norvig. *Inteligencia artificial: un enfoque moderno*. Prentice-Hall Hispanoamericana, 1996.

[Schwalb and Detcher, 1997] E. Schwalb and R. Detcher. Processing disjunctions in temporal constraint networks. *Artificial Intelligence*. Vol. 93 (1997), pp. 29-61.

[Starkweather, *et al.*, 1991] T. Starkweather, S. McDaniel, K. Mathias, D. Whitley, C. Whitley. A Comparison of Genetic Sequencing Operators. In R.K. Belew and L.B. Booker, editors. *Proceedings of the Forth International Conference on Genetic Algorithms, ICGA-91*, pp. 69-76. Morgan Kaufmann.

[Stergiou and Koubarakis, 2000] K. Stergious and M. Koubarakis. Backtracking algorithms for disjunctions of temporal constraints. *Artificial Intelligence*, 120 (2000) 81-117.

[Syswerda, 1991] G. Syswerda. Schedule Optimization Using Genetic Algorithms. In L. Davis, editor. *The Handbook of Genetic Algorithms*, pp. 332-349. Van Nostram Reinhold, 1991.

[Tsao and Wolter, 1993] J. Tsao and J. Wolter. Assembly planning with intermediate states. *IEEE International Coference on Robotics and Automation*, Atlanta, May 1993, pp. 71-76.

[Van Hentenryck, 1989] Pascal Van Hentenryck. *Constraint Satisfaction in Logic Programming*. The MIT Press, 1989.

[Whitley, 1987] L. D. Whitley. Using reproductive evaluation to improve genetic search and heuristic discovery. *Proceedings of the Second International Conference on Genetic Algorithms*, pp. 116-121, Grefenstette J. J. (ed.), Lawrence Erlbaum, 1987.

[Whitley, *et al.*, 1991] D. Whitley, T. Starkweather and D. Shaner. The traveling salesman and sequence scheduling: quality solutions using genetic edge recombina-

tion. In L. Davis, editor. *The Handbook of Genetic Algorithms*, pp. 332-349. Van Nostram Reinhold, 1991.

[Wilson and Rit, 1991] R. H. Wilson and J. F. Rit. Maintaining geometric dependencies in assembly planning. In L. S. Homem de Mello and S. Lee, editors, *Computer-Aided Mechanical Assembly Planning*. Kluwer Academic Publishers, 1991, pp. 217-241.

[Wilson, 1992] R. H. Wilson. On geometric assembly planning. PhD Thesis, Dept of Computer Science, Stanford University, March 1992.

[Wilson, *et al.*, 1995] R.H. Wilson, L. Kavraki, T. Lozano-Pérez and J.C. Latombe. Two-Handed Assembly Sequencing. *International Journal of Robotic Research*. Vol. 14, pp. 335-350, 1995.

[Wolper and Macready, 1995] D. H. Wolpert and W. G. Macready. *No Free Lunch Theorems for Search*. Technical Report SFI-TR-95-02-010, Santa Fe Institute, 1995.

[Wolter, 1988] J. Wolter. *On the automatic generation of plans for mechanical assembly*. Ph.D. thesis. University of Michigan. Department of Computer, Information and Control Engineering, September 1988.

[Wolter, 1992] J. Wolter. A Combinatorial Analysis of Enumerative Data Structures for Assembly Planning. *Journal of Design and Manufacturing*. Vol. 2, No. 2, pp. 93-104, June 1992.

[Wolter and Kroll, 1996] J. Wolter and E. Kroll. Toward assembly sequence planning with flexible parts. 13th *IEEE International Coference on Robotics and Automation*, 1996, pp. 1517-1524.