DEPARTAMENTO DE ESTADÍSTICA E INVESTIGACIÓN OPERATIVA

NUEVOS MÉTODOS DE RESOLUCIÓN DEL PROBLEMA DE SECUENCIACIÓN DE PROYECTOS CON RECURSOS LIMITADOS

FRANCISCO BALLESTÍN GONZÁLEZ

UNIVERSITAT DE VALENCIA
Servei de Publicacions
2002

Aquesta Tesi Doctoral va ser presentada a Valencia el día 20 de febrer de 2002 davant un tribunal format per:

- Jaime Barceló Bugeda
- Juan Larrañeta Astola
- Concepción Maroto Álvarez
- Mª Ángeles Pérez Alarcón
- Ramón Álvarez-Valdés Olaguible

Va ser dirigida per:
Prof. Dr. Vicente Valls Verdejo y Sacramento Quintanilla Alfaro

©Copyright: Servei de Publicacions
Francisco Ballestín González

Depòsit legal:
I.S.B.N.: 84-370-5566-0
Edita: Universitat de València
Servei de Publicacions
C/ Artes Gráficas, 13 bajo
46010 València
Spain
Telèfon: 963864115

VNIVERSITAT DE VALÈNCIA
Facultad de Ciencias Matemáticas
Departamento de Estadística e Investigación Operativa

Nuevos métodos de resolución del problema de secuenciación de proyectos con recursos limitados

TESIS DOCTORAL

Presentada por:

Francisco Ballestín González

Dirigida por:

Dr. Vicente Valls Verdejo

Dra. María Sacramento Quintanilla Alfaro

Valencia, 2001

Dn. Vicente Valls Verdejo profesor titular del Departamento de Estadística e Investigación Operativa y Dna. María Sacramento Quintanilla profesora titular del Departamento de Economía Financiera y Matemática de la Universidad de Valencia

CERTIFICAN que la presente memoria de investigación:

" Nuevos métodos de resolución del problema de secuenciación de proyectos con recursos limitados"

ha sido realizada bajo su dirección en el Departamento de Estadística e Investigación Operativa por Dn. Francisco Ballestín González y constituye su tesis para optar al grado de Doctor en Ciencias Matemáticas.

Y para que así conste, en cumplimiento con la normativa vigente, autorizan su presentación ante la Facultad de Matemáticas de la Universidad de Valencia para que pueda ser tramitada su lectura y defensa pública.

Valencia, 9 de noviembre de 2001

LOS DIRECTORES

Vicente Valls Verdejo María Sacramento Quintanilla Alfaro

UNIVERSIDAD DE VALENCIA

Facultad de Ciencias Matemáticas
Departamento de Estadística e Investigación Operativa

Nuevos métodos de resolución del problema de secuenciación de proyectos con recursos limitados

Memoria presentada por
Francisco Ballestín González
para optar al grado de Doctor
en Ciencias Matemáticas

Dirigida por
Dr. Vicente Valls Verdejo
Dra. María Sacramento Quintanilla Alfaro

A mi padre

Quiero agradecer su ayuda a una serie de personas que han contribuido al desarrollo de esta tesis.

A mis dos directores de tesis, simplemente por todo, y a su grupo de investigación, por ofrecer un ambiente muy agradable de trabajo y ayudar desinteresadamente en numerosas ocasiones.

A las personas que forman parte del Departamento de Estadística e Investigación Operativa: profesores, personal administrativo, becarios nacionales y extranjeros, etc. Por ayudarme a resolver los problemas de diversos tipos que han surgido a lo largo de estos años.

A los otros investigadores del RCPSP y a la justificación, por aportar conocimiento sobre el problema y, especialmente, por añadir interés a esta investigación.

A la Consellería de Cultura, Educación y Ciencia, por proporcionar los recursos ncesarios para la realización de esta Tesis Doctoral. Este trabajo ha sido financiado por la Generalitat Valenciana, con cargo al trabajo de investigación FPI98-CB-12-303.

Por último, pero no menos importante, a mi familia, amigos y compañeros, por su apoyo incondicional.

Índice

Prólogo 1

Capítulo 1. Revisión bibliográfica 5

1. Elementos de los problemas de secuenciación de proyectos. 5
    1.1. Actividades 5
    1.2. Relaciones de precedencia 6
    1.3. Recursos 7
    1.4. Objetivos de la secuenciación de proyectos. Funciones de evaluación 8
    1.5. Elementos aleatorios y deterministas 9
2. El Problema de Secuenciación de Proyectos con Recursos Limitados (RCPSP) 10
3. Generación de problemas test 15
4. Esquemas de generación de secuencias 18
    4.1. Esquema de secuenciación en Serie 18
    4.2. Esquema de secuenciación en Paralelo 20
5. Comentarios sobre Serie y Paralelo. Tipos de secuencias 21
6. Representación de las soluciones 23
    6.1. Lista de actividades 23
    6.2. Vector de prioridades 24
    6.3. Vector de reglas de prioridad 24
    6.4. Vector de traslación 25
    6.5. Vector de tiempos de inicio 25
    6.6. Representación mediante esquemas de secuencias 25
7. Técnicas metaheurísticas 26
    7.1. Búsqueda local 26
    7.2. Temple simulado 28
    7.3. Búsqueda tabú 29
    7.4. Algoritmos genéticos 30
    7.5. GRASP 31
    7.6. Búsqueda dispersa 32
    7.7. El metaheurístico de las hormigas 32
8. Heurísticos para el RCPSP 33
    8.1. Heurísticos basados en reglas de prioridad 33
    8.2. Heurísticos basados en técnicas metaheurísticas 39
    8.3. Otros métodos propuestos 43

Capítulo 2. Dos algoritmos heurísticos: CARA e HIAC 47

1. Introducción 47
2. El algoritmo CARA 50
    2.1. Definiciones preliminares 51
    2.2. Fase 1 de CARA 53
    2.3. Fase 2 de CARA 60
3. El algoritmo HIAC 64
    3.1. El algoritmo de intervalos homogéneos: un procedimiento de mejora 65
    3.2. El Algoritmo de Búsqueda Convexo: un método para combinar secuencias 73
    3.3. Algoritmo HIAC estándar 78
    3.4. Variaciones sobre el algoritmo estándar 80
4. Pruebas computacionales 81
5. Conclusiones 91
6. Distancias en el RCPSP 92
    6.1. Definición de distancia 92
    6.2. Ejemplos no triviales 93
    6.3. Usos de las distancias 101
7. Un esquema algorítmico válido para el RCPSP 103

Capítulo 3. La justificación 111

1. Introducción 111
2. El experimento 111
    2.1. Primer conjunto de experimentos 113
    2.2. Segundo conjunto de experimentos 117
        2.2.1. Operadores binarios 118
        2.2.2. Funciones empleadas 120
        2.2.3. Población inicial 121
        2.2.4. Esquemas algorítmicos 122
        2.2.5. Resultados computacionales 123
    2.3. Aplicación de la doble justificación a una función de mejora 126
        2.3.1. La función de mejora 126
        2.3.2. Resultados computacionales 127
    2.4. La doble justificación en un algoritmo de calidad 128
    2.5. El papel de la justificación en CARA e HIAC 131
3. El tiempo de cómputo de la justificación 132

Capítulo 4. Análisis teórico de la justificación 139

1. Introducción 139
2. La justificación 140
   2.1. La justificación general 140
   2.2. La justificación por extremos 142
   2.3. El algoritmo de justificación general 144
3. Un par de ejemplos prácticos 151
4. La traslación 154
   4.1. La relación entre la traslación y la justificación 155
   4.2. La traslación como caso más general posible 156
   4.3. La traslación como problema de optimización 158
5. Técnicas relacionadas con la traslación/justificación 160
6. La justificación en otros problemas de secuenciación de proyectos con recursos limitados 165
   6.1. El RCPSP con interrupción 166
      6.1.1. Justificación directa 168
      6.1.2. Justificación adaptada 169
      6.1.3. Pruebas computacionales de la justificación en el RCPSP_inter 174
   6.2. El RCPSP con múltiples modos y recursos no renovables 177
      6.2.1. Justificación directa 178
      6.2.2. Justificación adaptada 179
   6.3. El RCPSP con fechas de entrega en la función objetivo 183
      6.3.1. Justificación directa 184
      6.3.2. Justificación adaptada 185

Capítulo 5. HGA, un algoritmo genético híbrido 189

1. Introducción 189
2. HGA, el algoritmo de los picos 190
   2.1. El operador de cruce de los picos 191
   2.2. Una población en el vecindario de una solución 194
   2.3. Esquema algorítmico de HGA 195
3. Resultados computacionales 196

Capítulo 6. Resultados teóricos sobre la combinación de picos 205

1. Introducción 205
2. Resultados teóricos 205
   - 2.1.Cordillera acetable. El grafo de una cordillera 207
   - 2.2. Caracterización de las cordilleras aceptables. El grafo de los picos 210
   - 2.3. Generación de cordilleras aceptables 217
3. Posibles formas de construir una lista de actividades que contenga una cordillera aceptable... 221
4. Posibles técnicas y algoritmos relacionados con los picos 222

Capítulo 7. Conclusiones y líneas futuras de investigación 225

1. Conclusiones 225
2. Líneas futuras de investigación 228

Bibliografía 231

Prólogo

El Problema de Secuenciación de Proyectos con Recursos Limitados (RCPSP), problema objeto de estudio en la presente tesis, está considerado como el problema básico más importante dentro de la secuenciación con recursos limitados. Dada la importancia del problema, tanto en el ámbito práctico como en el teórico, han sido numerosas las investigaciones publicadas que versan sobre él. Entre sus aplicaciones prácticas podemos citar la construcción de edificios, la planificación de la producción y el desarrollo de grandes sistemas de transporte y energía. Dado que es la base de numerosos problemas de secuenciación, cualquier avance en la resolución de este problema puede repercutir rápidamente en la resolución de muchos otros problemas. Así pues, el RCPSP es un "banco de pruebas" donde se experimentan y desarrollan nuevas ideas y técnicas de resolución que después se trasladan a otros problemas. Todas estas consideraciones nos han motivado para estudiar con profundidad este problema.

El RCPSP consiste en realizar un conjunto de actividades sujetas a dos tipos de restricciones. Por una parte, las relaciones de precedencia fuerzan a algunas actividades a comenzar después de la finalización de otras. Por otra parte, procesar cada actividad requiere consumir recursos, los cuales están disponibles en una cantidad fija y limitada en cada unidad de tiempo. El objetivo del RCPSP consiste en encontrar tiempos de inicio para las actividades de manera que se minimice la diferencia entre la finalización de la actividad que termina más tarde y el comienzo de la actividad que comienza a secuenciarse más temprano.

Como extensión del job-shop, el problema que nos ocupa es un problema NP-duro y, actualmente, el límite de los algoritmos exactos se sitúa entre 30 y 60 actividades, lo que implica que son necesarios los algoritmos heurísticos para proporcionar soluciones de buena calidad para instancias grandes.

Los primeros algoritmos heurísticos publicados fueros algoritmos basados en reglas de prioridad que construían una única solución del problema. Posteriormente se desarrollaron distintos algoritmos que proporcionaban la mejor de entre un conjunto de soluciones obtenidas de forma independiente. Actualmente, igual que sucede en muchos problemas NP-duros, el desarrollo de algoritmos metaheurísticos que exploran el espacio de soluciones "guiando" la búsqueda ha permitido mejorar la calidad de las soluciones.

El objetivo del presente trabajo es desarrollar procedimientos eficaces y eficientes para resolver el RCPSP, consiguiendo avances significativos que no sean rápidamente superados. Para ello hemos profundizado en el conocimiento de la estructura del

RCPSP y hemos desarrollado algoritmos híbridos que combinan, de una manera coordinada, las ideas fundamentales de diversos enfoques metaheurísticos con los nuevos procedimientos que el conocimiento del RCPSP nos ha proporcionado.

La memoria se estructura según las siguientes secciones. En el capítulo 1 se describe el problema, englobándolo dentro de la secuenciación de proyectos, y se realiza una revisión comentada de los enfoques heurísticos propuestos para el RCPSP.

En el capítulo 2 se describen dos nuevos algoritmos heurísticos para el problema: CARA e HIAC. Como ya se ha mencionado, las dos restricciones clave del RCPSP son las relaciones de precedencia y las restricciones de recursos. Como punto de partida, hemos diseñado dos algoritmos que se basan por separado en la información proporcionada por las relaciones de precedencia (análisis temporal del proyecto) y en la utilización de recursos; de este modo, se intentará discernir la importancia relativa de estos dos tipos de informaciones.

El algoritmo metaheurístico CARA es el resultado del primer enfoque. El procedimiento consiste en una implementación no estándar, dentro de un marco de poblaciones, de los conceptos fundamentales de la búsqueda tabú, que no utiliza explícitamente las estructuras de memoria.

El segundo procedimiento (HIAC) sigue un enfoque basado en la evolución de poblaciones y usa, básicamente, información sobre utilización de recursos. La evolución de la población es dirigida por la aplicación alternativa de un eficiente procedimiento de mejora para mejorar localmente la utilización de recursos, y de un mecanismo para combinar secuencias que integra características de la búsqueda dispersa y del reencadenamiento de trayectorias.

Una vez descritos ambos algoritmos, se demuestra su calidad comparándolos con los mejores heurísticos conocidos.

Además, dentro del capítulo 2, se definen distintas distancias en el RCPSP y un esquema algorítmico (MetaRCPSP) capaz de producir algoritmos heurísticos para el RCPSP al especificar cada uno de sus componentes.

El análisis de los resultados intermedios obtenidos por los dos algoritmos anteriores indica el potencial de una técnica que ya había sido definida pero casi no ha sido empleada, la justificación. Es una técnica sencilla y muy rápida que al aplicarla a una secuencia produce una solución de, cómo máximo, igual duración y, en muchos casos, de menor duración. La justificación puede ser incorporada fácilmente a muy diversos algoritmos para el RCPSP obteniéndose, de esta manera, mejoras sensibles de la

calidad de las secuencias generadas sin que ello requiera, en general, un incremento en el tiempo de cómputo. En el capítulo 3 se constata este hecho, demostrando la utilidad de esta técnica, no empleada en la mayoría de algoritmos heurísticos. Los resultados obtenidos indican que la justificación aumenta considerablemente la calidad de los algoritmos hasta tal punto que su sola inclusión permite a algoritmos sencillos mejorar a los mejores algoritmos publicados hasta el momento. En concreto, la incorporación de la doble justificación a un algoritmo genético de la literatura que combina las soluciones con un operador muy general (el operador de dos puntos de corte) convierte a este algoritmo en el mejor de los heurísticos para el RCPSP publicados.

En el capítulo 4 se definen diferentes generalizaciones de la justificación, se estudia las relaciones entre ellas y se demuestra que este campo de estudio es atractivo a nivel práctico. Así mismo, se describe cómo adaptar la justificación a otros problemas de secuenciación de proyectos con recursos limitados y se demuestra con un ejemplo que esta técnica puede ser muy útil en algunas extensiones del RCPSP.

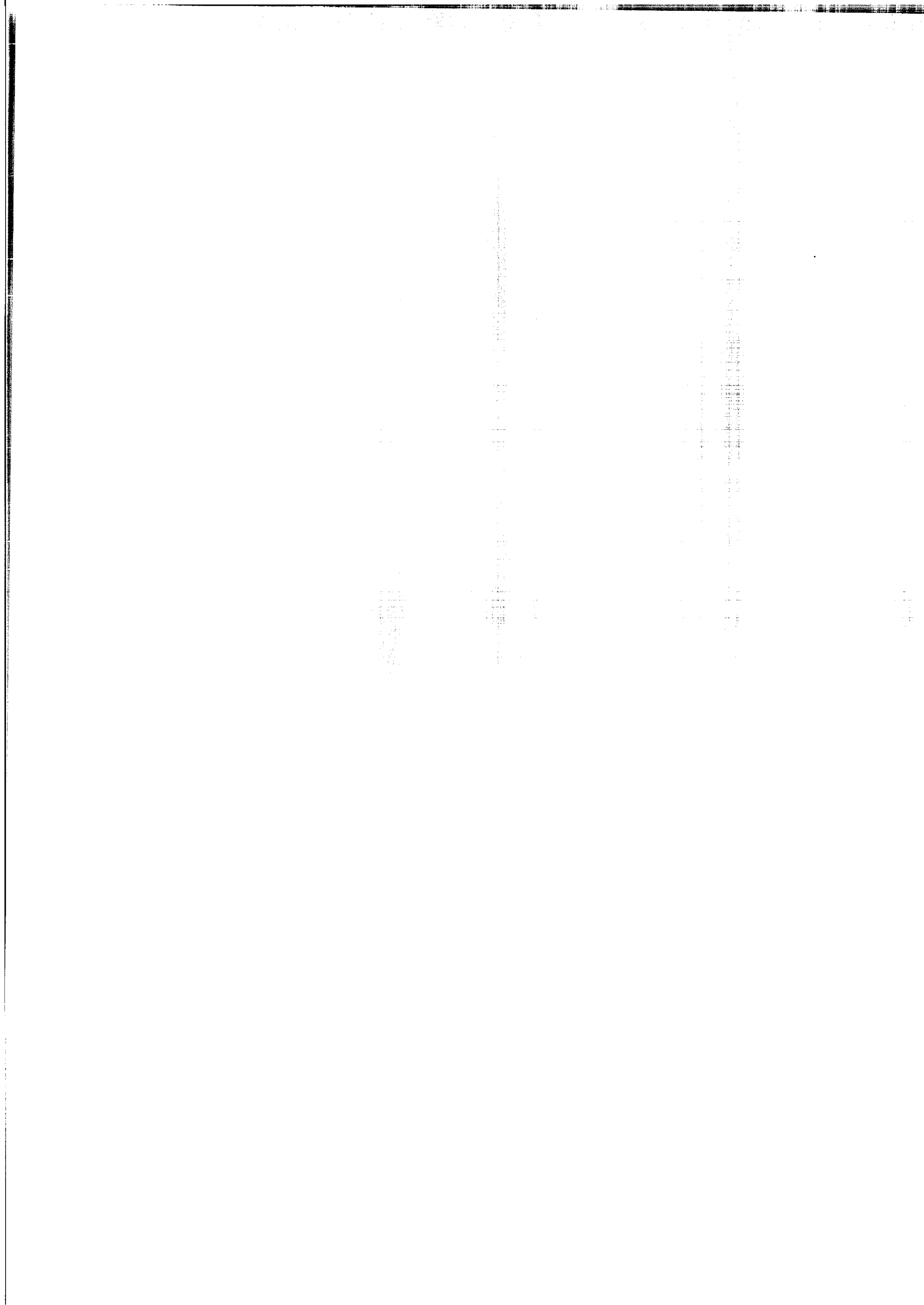
En el capítulo 5 nos planteamos el diseño de un nuevo algoritmo genético basado en una manera de combinar soluciones más específica del problema que pueda superar al operador de dos puntos de corte. El resultado es un algoritmo genético híbrido, HGA, que combina algunos de los procedimientos que han demostrado su eficacia en otros algoritmos. HGA emplea un nuevo operador de cruce (operador de cruce de los picos) cuya finalidad es explotar el conocimiento del problema para identificar y combinar partes de buenas soluciones que realmente han contribuido a su calidad. Al aplicar el operador de cruce de los picos a dos secuencias padres, las secuencias hijos heredan las agrupaciones de actividades que en los padres supusieron una utilización alta de recursos, los "picos" en la gráfica de utilización de recursos. El algoritmo desarrollado es mejor que todos los publicados e, incluso, que los definidos en la tesis.

En el capítulo 6 hemos desarrollado una teoría y una metodología que permite seleccionar un conjunto de picos provenientes de distintas secuencias y construir, a partir de ellos, nuevas secuencias. Los resultados desarrollados constituyen una base teórica para futuras técnicas y nuevos algoritmos donde se combinen picos de varias soluciones.

Por último, en el capítulo 7, se exponen las conclusiones y las líneas futuras de investigación.

Capítulo 1

Revisión bibliográfica



1. ELEMENTOS DE LOS PROBLEMAS DE SECUENCIACIÓN DE PROYECTOS

Los **problemas de secuenciación de proyectos** (PSP) están formados por **actividades**, **recursos**, **relaciones de precedencia** y **funciones de evaluación** (Slowinski et al., 1994). Las primeras aproximaciones fueron desarrolladas en los 50. Algunos artículos y libros históricos son Kelley, 1961, Malcolm et al., 1959, Moder et al., 1983 y Pritsker y Happ, 1966. Los proyectos se pueden encontrar en diferentes áreas; ejemplos típicos incluyen proyectos de construcción como la construcción de rascacielos, puentes y autopistas. Muchos problemas de planificación de producción bajo pedido y de pequeños lotes se pueden formular como proyectos. También deben ser mencionados los proyectos de desarrollo e investigación. Las reuniones masivas como conferencias políticas internacionales o eventos deportivos como los juegos olímpicos y los mundiales de fútbol se pueden ver también como proyectos.

Herroelen et al. han propuesto recientemente (Herroelen et al., 1998) una clasificación de los modelos de secuenciación de proyectos basada en los cuatro elementos mencionados. En Brucker et al., 1999, se ha desarrollado una clasificación compatible con lo generalmente aceptado en la secuenciación de máquinas y en la secuenciación de máquinas con restricciones de recursos.

1.1. Actividades

Un **proyecto** consta de un número de actividades que hay que realizar, también conocidas como trabajos, operaciones y tareas. Generalmente, y según Kolisch, 1995, el término actividades es el empleado para los problemas de secuenciación de proyectos, mientras que los términos trabajos, operaciones y tareas son utilizados en el job shop (estático), flow shop y secuenciación de procesadores respectivamente. Para completar el proyecto de forma satisfactoria es necesario procesar (ejecutar, secuenciar) cada actividad en uno de diversos **modos**, donde cada modo representa una forma distinta de realizar las actividades. Un modo determina la duración de la actividad, medida en número de periodos o unidades de tiempo, que indica el tiempo necesario para completar la actividad, y si existe la posibilidad de interrumpir el proceso de esa actividad. También determina los requerimientos de recursos de varias categorías (cf. apartado 1.3) y los posibles flujos monetarios que ocurren al principio, durante el proceso, o al completar esa actividad (cf. apartado 1.4).

Cada actividad puede llevar asociada una **fecha de entrega**, que indica el tiempo máximo en el que se debe haber completado la actividad o parte de ella, y una **fecha de disponibilidad**, que marca el instante de tiempo a partir del cual se puede procesar la actividad.

Lo que se debe determinar en un PSP es cuándo y en qué modo se va a secuenciar cada actividad.

1.2. Relaciones de precedencia

A menudo, razones tecnológicas derivan en que la secuenciación de una actividad depende de cuándo se ejecute otra. Estas relaciones entre actividades se denominan **de precedencia**, porque la forma habitual (y la más sencilla) en que aparecen se basa en que una actividad no puede comenzar antes de que otra(s) finalice(n). Las relaciones de precedencia entre actividades se pueden visualizar representando el proyecto mediante un grafo dirigido donde cada actividad se representa por un nodo o vértice y una relación de precedencia entre dos actividades, por una arista dirigida. Esta representación se conoce como AON ('Activity-On-Node', actividades en los nodos). A esa arista le corresponde un tipo y un número que caracteriza la relación existente entre las dos actividades. Si, por ejemplo, la actividad j sólo puede empezarse a secuenciar a partir de que finalice la actividad i, el tipo de relación es FS ('Finish-Start', el final de la actividad i restringe el comienzo de la actividad j) y el parámetro es $FS_{ij} = 0$, porque la actividad puede comenzar inmediatamente después (0 unidades de tiempo después) de la finalización de i. Si de la actividad i parte una arista hacia la j, se dice que i es **predecesora inmediata** de j y que j es **sucesora inmediata** de i.

También existe otra representación del problema en forma de grafo dirigido donde las actividades se representan mediante aristas (AOA, 'Activity-On-Arc'), pero sólo ofrece ventajas de visualización cuando es importante representar instantes de completación de conjuntos de actividades (eventos) y para modelizar y analizar un número reducido de tipos de problemas (cf. Elmaghraby, 1995). Entre sus inconvenientes destacan la no unicidad de la representación y la necesidad de incluir vértices y aristas ficticias. Para un resumen de las semejanzas y diferencias entre ambas representaciones ver Kolisch y Padman, 2001.

1.3. Recursos

Generalmente, es necesario realizar algún tipo de consumo para realizar las actividades. En los PSP se modeliza ese consumo a través de los recursos que utiliza cada actividad.

Estos recursos se pueden clasificar según **categorías**, **tipos** y **valores** (Bey et al., 1981).

Las distintas categorías de recursos son: recursos renovables, no renovables, parcialmente renovables y doblemente restringidos.

La disponibilidad de los recursos renovables está limitada en cada unidad de tiempo. Esto quiere decir que, sin tener en cuenta la duración del proyecto, cada recurso renovable está disponible en una cierta cantidad constante en cada unidad de tiempo y que su utilización no puede exceder esa cantidad en ninguna de esas unidades. Como ejemplo de estos recursos están las máquinas. Si disponemos de una única máquina podemos utilizarla en cada unidad de tiempo pero en ninguna de ellas podemos utilizar varias máquinas. Existen problemas (cf. Hartmann, 1999, Klein y Scholl, 1998 y Sprecher, 1994) donde se generaliza este concepto y se considera que las disponibilidades de los recursos e incluso los requerimientos de las actividades pueden variar con el tiempo.

Los recursos no renovables están limitados sobre la duración completa del proyecto, sin restricciones sobre cada periodo. El presupuesto total para la realización de un proyecto es un ejemplo de recurso no renovable.

Los recursos doblemente restringidos están limitados tanto sobre el horizonte de planificación como sobre cada periodo de tiempo. Talbot, 1982, demostró formalmente que cada recurso de este tipo se puede descomponer en uno renovable y uno no renovable. Un ejemplo de recurso doblemente restringido es el presupuesto, si éste está restringido globalmente y además existen restricciones diarias.

Los recursos parcialmente renovables, introducidos por Böttcher et al., 1996, definen la disponibilidad de un recurso para subconjuntos de periodos. Estos mismos autores demostraron que tanto los recursos renovables como los no renovables y los doblemente restringidos se pueden representar por recursos de este tipo.

Dentro de cada categoría, podemos necesitar o utilizar diferentes recursos, cada uno con diferentes funciones a realizar. Se habla entonces de los diferentes tipos de

recursos. La clasificación de tipo distingue, por tanto, dentro de cada categoría a los distintos recursos con respecto a la función que realizan.

Por último, cada tipo de recurso lleva asociado un valor que representa las unidades disponibles de ese tipo de recurso.

1.4. Objetivos de la secuenciación de proyectos. Funciones de evaluación

Todos los problemas de optimización consisten en encontrar una "mejor" solución respecto de un criterio determinado.

Una **función de evaluación** ('performance measure') es una manera de cuantificar la calidad de una solución, el criterio por el cual se mide su bondad. Es lo que se denomina generalmente función objetivo en un problema de optimización general.

Cada función de evaluación define un problema distinto, aunque el conjunto de soluciones posibles sea el mismo. Diferentes funciones de evaluación pueden requerir técnicas distintas para resolver el problema.

La minimización de la **duración del proyecto** ('makespan') es, probablemente, la función objetivo más estudiada y aplicada en el dominio de la secuenciación de proyectos (cf. Kolisch y Padman, 2001). La duración del proyecto se define como el tiempo transcurrido entre el principio y el final del proyecto. Dado que el inicio del proyecto se sitúa habitualmente en $t = 0$, minimizar la duración del proyecto se reduce a minimizar el máximo de los finales de las actividades.

La minimización de la duración del proyecto es una medida regular (Sprecher et al., 1995), así como la minimización de los tiempos de paso (ponderados) de las actividades, o la minimización de los retrasos (ponderados) si existen fechas de entrega (Slowinski, 1989, Sprecher y Drexl, 1998). Una función de evaluación se denomina **(medida) regular** si al comparar dos secuencias para un problema dado que difieran únicamente en el tiempo de completación de una actividad podemos asegurar que la secuencia con el menor tiempo de finalización en esa actividad es al menos tan buena (medida por la función de evaluación) como la otra secuencia, i.e., la primera domina a la última.

Cuando existen niveles significativos de flujos monetarios en el proyecto, en la forma de gastos por realizar actividades y/o pagos por la completación de partes del proyecto, el criterio del **valor neto actualizado** ('net present value') puede ser una

medida más apropiada de la calidad de la solución (Bey et al., 1981). Esta función objetivo es una medida regular si los flujos monetarios son todos positivos (Herroelen et al., 1998). Se obtienen generalizaciones de este problema si se incluyen restricciones en los recursos, consideraciones en las restricciones de material y capital, y consideraciones de intercambios tiempo/coste y aspectos multimodo.

La maximización de la calidad (Icmeli y Rom, 1997 y 1998) y la minimización de los costes son otros objetivos importantes en la secuenciación de proyectos, debido a su importante presencia en las aplicaciones reales.

Los objetivos basados en los costes pueden ser de dos tipos: objetivos basados en los costes sobre las actividades y sobre los recursos.

En los problemas con objetivo basado en los costes sobre las actividades, la manera en que se realizan las actividades resulta en costes directos que deben minimizarse. El problema de intercambio tiempo/coste ('time/cost trade-off') continuo (Elmaghraby, 1977) y sus versiones discretas (Deckro y Hebert, 1989 y 1990, Demeulemeester et al., 1996) son ejemplos de este tipo de problemas.

En los problemas con objetivo basado en los costes sobre los recursos, la secuenciación de las actividades influye indirectamente en el coste a través de los recursos. Ejemplos para este tipo de problemas son el problema de nivelación de recursos (Bandelloni et al., 1994, Brinkmann y Neumann, 1996) y el problema de inversión en recursos ('resource investment') (Demeulemeester, 1995, Möhring, 1984).

1.5. Elementos aleatorios y deterministas

En los apartados anteriores no se ha comentado nada acerca del conocimiento o incertidumbre de los datos iniciales de los problemas. Dependiendo de si todos los datos concernientes al problema están determinados de forma exacta a priori o no, los problemas se dividen en **deterministas** y **estocásticos**.

El ejemplo más antiguo, y uno de los más conocidos y estudiados de los problemas estocásticos de secuenciación de proyectos, es el PERT, donde las relaciones de precedencia son del tipo FS con valor 0 y no hay restricciones de recursos, pero las duraciones de las actividades no son conocidas a priori, sino que únicamente se dispone de tres estimaciones de cada duración, la optimista, la pesimista y la más probable (Malcolm et al., 1959).

2. El Problema de Secuenciación de Proyectos con Recursos Limitados (RCPSP)

El **RCPSP**, a veces denominado SMPSP, de 'Single-Mode Project Scheduling Problem', es uno de los problemas básicos y clave en los PSP. Utilizando los elementos descritos en el apartado anterior, se puede describir el RCPSP como sigue.

Se trabaja con un único proyecto formado por un conjunto de actividades **V** = {1, ... ,n}, que deben ser procesadas. Las actividades 1 y n son actividades ficticias que representan el inicio y final del proyecto, respectivamente.

Las actividades están interrelacionadas por dos tipos de restricciones. Primero, las relaciones de precedencia son del tipo FS con parámetro 0. Para cada actividad j existe un conjunto **Pred**$_j$ de actividades que deben finalizar antes de que j pueda comenzar. A los elementos de $Pred_j$ se les denomina **predecesores inmediatos** de j. Se cumple, sin pérdida de generalidad, que $Pred_j \neq \emptyset \; \forall j \neq 1$, y $\forall j \neq n \; \exists \, i \,/\, j \in Pred_i$. Una actividad i se llama **predecesora** de j si existe $i_0, i_1, ..., i_k, i_{k+1}$ con $i_0 = i$, $i_{k+1} = j$, $i_b \in Pred_{i_{b+1}}$, b = 0, 1, ..., k. La actividad i se dice que es **sucesora (inmediata)** de j si j es predecesora (inmediata) de i. El conjunto de los sucesores inmediatos de j se denota por **Suc**$_j$. Se cumple que 1 (n) es predecesora (sucesora) de toda actividad.

El segundo tipo de restricciones se refiere a los recursos. Realizar las actividades consume recursos de los que se dispone en cantidades limitadas. Disponemos de K tipos de recursos renovables, incluidos en el conjunto K = {1, ..., K}. Se dispone de una cantidad limitada del tipo de recurso k en cada instante de tiempo, llamada R_k. Mientras está siendo procesada, una actividad j necesita $r_{j,k}$ unidades del tipo de recurso $k \in K$ durante cada periodo de su duración d_j (únicamente existe un modo en el que procesar las actividades). Las actividades no se pueden interrumpir una vez comenzadas.

Los parámetros d_j, $r_{j,k}$ y R_k se suponen conocidos, determinados y enteros; además, $d_1 = d_n = 0$ y $r_{1,k} = r_{n,k} = 0 \; \forall \, k \in K$.

El objetivo del RCPSP es encontrar tiempos de comienzo (o de finalización, es equivalente dada la no interrupción de las actividades) para todas las actividades que cumplan las restricciones de precedencia y de recursos, y de tal manera que se minimice la duración del proyecto.

En la Figura 1.1 se muestra un ejemplo (tomado del artículo Sprecher et al., 1995) de un proyecto (lo llamaremos PR1) que contiene n = 7 actividades y K = 1 tipo de

recurso renovable con una limitación de 2 unidades. La figura contiene todos los datos del proyecto. En la Figura 1.2 se representa una secuencia posible con una duración del proyecto de 8 unidades. En la Figura 1.3 se representa la única solución óptima del problema, con una duración de 5 unidades. Este modo de representar una secuencia se denomina **diagrama de Gantt**. El eje horizontal representa el tiempo y, en este caso, el vertical representa la utilización del único tipo de recurso. Si existe más de un tipo de recurso, es necesario dibujar un diagrama para cada tipo de recurso. Cada rectángulo representa la actividad a la que alude el número que aparece dentro de él. La longitud del rectángulo se corresponde con la duración de la actividad que representa, y la altura indica las unidades de ese tipo de recurso que emplea esa actividad. Emplearemos a lo largo de la tesis estas formas de representar los proyectos y las figuras.

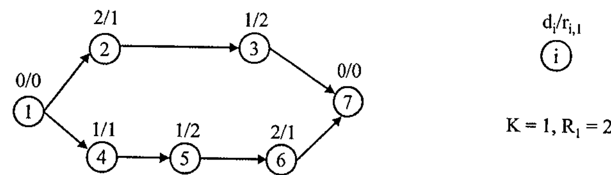


Figura 1.1

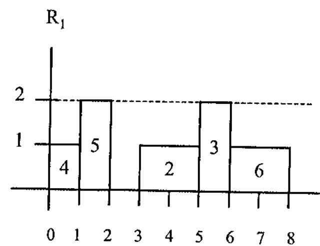


Figura 1.2

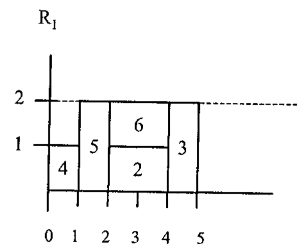


Figura 1.3

Denotaremos a lo largo de la tesis por s_j (f_j) al comienzo (final) de la actividad j en una secuencia S.

Llamaremos **secuencia** o **solución** S a cualquier vector de tiempos de comienzo $\{s_1, s_2, ..., s_n\}$ o, equivalentemente, a cualquier vector de tiempos de finalización $\{f_1, f_2, ..., f_n\}$.

Supondremos, salvo que se especifique lo contrario, que siempre se cumple $s_1 = 0$ y $s_n = \max\{f_i : i = 1, ..., n\text{-}1\}$. Denominaremos **longitud o duración de la secuencia** a f_n, la duración del proyecto si éste se realizara según esa secuencia, y la denotaremos por T(S).

Llamaremos **secuencia posible** o **solución posible** a aquella secuencia donde las actividades cumplen las relaciones de precedencia y las restricciones de recursos.

Dada una secuencia S, sea $A(t) = \{j \in \mathbf{V} / s_j \leq t < f_j\}$ ($= \{j \in \mathbf{V} / f_j - d_j \leq t < f_j\}$) el conjunto de actividades que se están procesando (actividades **activas**) en el instante t.

El RCPSP se puede modelizar de la siguiente forma (cf. Christofides et al., 1987):

$$\begin{aligned}
&\text{Min} \quad f_n && (1)\\
&\text{s.a.}: \quad f_h \leq f_j - d_j \ (= s_j), \ j = 2,\ldots,n, h \in \text{Pred}_j && (2)\\
&\sum_{j \in A(t)} r_{j,k} \leq R_k \qquad k \in K; t \geq 0 && (3)\\
&f_j \geq 0 \text{ y enteras}, j = 2,\ldots,n && (4)
\end{aligned}$$

La función objetivo (1) minimiza el tiempo de finalización de la actividad que marca el final del proyecto y, por tanto, la duración del proyecto. Las restricciones (2) modelizan las relaciones de precedencia, y las restricciones (3) limitan la utilización de recursos de cada recurso k en cada instante t a la limitación existente. Finalmente, (4) define las variables de decisión como enteras no negativas. (1) - (4) es un modelo conceptual ya que los conjuntos $A(t)$ son función de las variables de decisión. Por tanto, si queremos resolver el RCPSP por medio de la programación lineal entera mixta debemos utilizar otro modelo. Más adelante describiremos un modelo de PLE para el RCPSP que necesita de algunas definiciones adicionales.

El RCPSP se puede denotar como m, 1 | cpm | Cmax según la nomenclatura de Herroelen et al., 1998, o como PS | prec | Cmax según la de Brucker et al., 1999.

Blazewicz et al., 1983, demostraron que el RCPSP, generalización del problema clásico del job shop, pertenece a la clase de los problemas de optimización NP-hard. El RCPSP, sin embargo, se convierte en un problema resoluble en tiempo polinomial si eliminamos las restricciones de recursos (restricciones (3) en el modelo conceptual anterior). Este problema irrestringido (cpm | Cmax según la notación de Herroelen et al., 1998) se resuelve utilizando la **recursión hacia adelante** (cf. Elmaghraby, 1977):

(1) $s_1 = 0$;

(2) $s_j = \max\{s_h + d_h / h \in \text{Pred}_j\}$, j = 2, ..., n.

Este método genera una secuencia denominada secuencia **ES** ('Early Start'), donde cada actividad está secuenciada lo más temprano posible, teniendo en cuenta las relaciones de precedencia. Se denota por $\mathbf{ES}_j$ al tiempo de comienzo de la actividad j

en la secuencia ES y $\mathbf{EF}_j$ a $ES_j + d_j$, el tiempo de finalización más temprano de la actividad j.

Existe también el concepto opuesto, una secuencia (posible con respecto a las relaciones de precedencias) con la misma longitud de proyecto que la ES donde cada actividad se secuencia lo más tarde posible. En este caso se utilizan los cálculos o **recursión hacia atrás**, y la secuencia se denomina **LS** ('Latest Start'):

(1) s_n = Longitud de la secuencia ES.

(2) $s_j = \min\{s_h / j \in P_h\} - d_j,\ j = n-1, ..., 1.$

Se denota por $\mathbf{LS}_j$ a los s_j así obtenidos y $\mathbf{LF}_j$ a $LS_j + d_j$, el tiempo de finalización más tardío de la actividad j. Si los cálculos comienzan en otro punto distinto a la longitud de la secuencia ES se obtienen tiempos de comienzo y finalización distintos.

En las Figuras 1.4 y 1.5 se representan las secuencias ES y LS correspondientes a PR1, con una duración de 4 unidades.

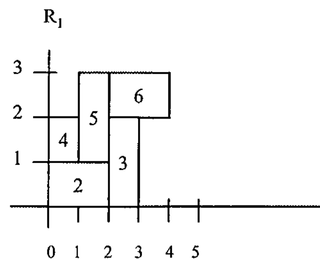


Figura 1.4

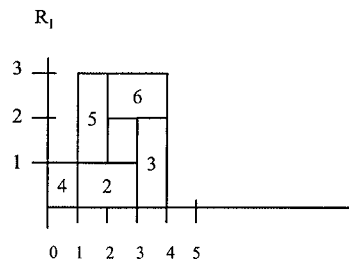


Figura 1.5

Las secuencias ES y LS (o, equivalentemente, los cálculos hacia adelante y hacia atrás) definen un conjunto de actividades llamadas **críticas**, aquellas que cumplen $ES_j = LS_j$ (en PR1 las actividades 4, 5 y 6). Se denominan **caminos críticos** a todos aquellos caminos del grafo cuya longitud (medida por la suma de las duraciones de los nodos que lo componen) es la misma que la de la secuencia ES (y LS). El único camino crítico de PR1 es {4,5,6}. Denotaremos por **CPM** a la longitud común de los caminos críticos. El CPM es una cota inferior para la duración del proyecto en el

RCPSP y la desviación con respecto al CPM se emplea a menudo para la comparación entre heurísticos dado que, para instancias grandes, no se suele conocer la solución óptima.

A continuación se describe el modelo de programación lineal entera del RCPSP propuesto por Pritsker et al. en 1969. Los valores LF_j se calculan tomando como dato inicial T, una cota superior del proyecto, en lugar de la longitud de la secuencia ES. Las variables x_{jt} valen 1 si la actividad j finaliza en el instante t, 0 en otro caso.

$$\text{Min} \quad \sum_{t=EF_n}^{LF_n} t\, x_{nt}$$

$$\text{s.a.} \quad \sum_{t=EF_j}^{LF_j} x_{jt} = 1 \qquad \forall\, j = 1, \ldots, n$$

$$\sum_{t=EF_h}^{LF_h} t\, x_{ht} \le \sum_{t=EF_j}^{LF_j} (t - d_j) x_{jt} \qquad \forall\, j = 2, \ldots, n,\; h \in Pred_j$$

$$\sum_{j=2}^{n} r_{jk} \sum_{b=t}^{t+d_j-1} x_{jb} \le R_k \qquad k \in K,\; t = 1, \ldots, T = \sum_{j=1}^{n} d_j$$

$$x_{jt} \in \{0,1\},\; j \in \mathbf{V},\; t = EF_j, \ldots, LF_j$$

Otras formulaciones del RCPSP se pueden encontrar en Klein, 2000.

Debido a la dificultad del RCPSP, los procedimientos heurísticos son indispensables para resolver problemas grandes como los que suelen aparecer en las aplicaciones. A pesar de ello, se ha experimentado un avance importante en los últimos años respecto a la resolución exacta. En los últimos años ha surgido una amplia variedad de estos algoritmos, demostrando el interés que despierta el problema y su dificultad. Estos últimos enfoques se pueden consultar en los artículos recopilatorios Brucker et al., 1999, Kolisch y Padman, 2001 y Özdamar y Ulusoy, 1995. Otros artículos recopilatorios más antiguos sobre algoritmos exactos son los de Davis 1966 y 1973, Demeulemeester, 1992, Herroelen, 1972 y Patterson, 1984.

Los algoritmos exactos más competitivos (cf. Kolisch y Hartmann, 1999) son los de Brucker et al., 1998, Demeulemeester y Herroelen, 1997, Mingozzi et al., 1998 y Sprecher, 1996. Otros algoritmos exactos recientes son los de Dorndorf et al., 2000 y

Klein y Scholl, 1998. Todos ellos utilizan un algoritmo de ramificación y acotación (B&B).

3. Generación de problemas test

Con el desarrollo de procedimientos heurísticos y exactos para el RCPSP surgió la necesidad de crear instancias para evaluarlos y compararlos. Durante varios años la mayoría de investigadores generaron sus propios problemas test (cf. Alvarez-Valdés y Tamarit, 1989b, Boctor, 1990, Patterson, 1984). Ello dificultaba la comparación entre los métodos ya que, por una parte, no se probaban los algoritmos sobre los mismos problemas (salvo en las contadas ocasiones en que algún autor lograba reunir más de un algoritmo) y, por otra parte, los problemas se generaban a menudo a partir de un conjunto muy restringido de características del proyecto.

En 1984, Patterson recopiló 110 problemas de diversos autores para los que calculó la solución óptima. Los problemas de Patterson, como se denominó ese conjunto de instancias, fueron utilizados durante muchos años para medir la eficacia de los algoritmos (Bell y Han, 1991, Davis y Patterson, 1975, Demeulemeester y Herroelen, 1992, Kolisch, 1996a, Lee y Kim, 1996, Leon y Ramamoorthy, 1995, Patterson, 1976, Patterson y Roth, 1976 , Sampson y Weiss, 1993).

Los 110 problemas se dividen en tres grupos, 10 con menos de 20 actividades y entre 1 y 4 tipos de recursos, 90 con entre 21 y 30 actividades y 2 tipos de recursos, y 10 con n = 51 y 2 ó 3 tipos de recursos.

Estos problemas se convirtieron en estándar, pero presentan dos desventajas importantes (Kolisch et al., 1995):

- Como colección de problemas provenientes de diferentes fuentes no están generados utilizando un diseño controlado con unos parámetros específicos.

- Cuando evolucionaron los algoritmos exactos se demostró que los problemas del conjunto de Patterson era resolubles de forma óptima en muy poco tiempo; hoy en día se resuelven en 0.002 segundos de media (Herroelen et al., 1998). Dado que existen instancias con el mismo número de actividades mucho más difíciles de resolver, no se puede considerar este conjunto como representativo ni puede servir como banco de pruebas.

Como respuesta a estas desventajas, y en vista de la necesidad de un enfoque estructurado para generar instancias, Kolisch et al., 1992 y 1995 propusieron un generador para una clase general de PSP, al que denominaron **PROGEN** ('PROject GENerator').

PROGEN genera proyectos de forma aleatoria, pero de manera que la instancia resultante cumple una serie de restricciones introducidas a priori. Estas restricciones están basadas en una serie de parámetros, que podemos dividir en dos subconjuntos. El primer subconjunto de parámetros consta de:

(i) el número de actividades;

(ii) el número de tipos de recursos existente en el problema;

(iii) la duración de las actividades no ficticias;

(iv) el número de recursos que utiliza cada actividad;

(v) la cantidad de cada recurso que utiliza cada actividad;

(vi) el número de sucesores de 1;

(vii) el número de predecesores de n;

(viii) el número de predecesores y de sucesores del resto de actividades.

El segundo subconjunto de parámetros contiene los siguientes parámetros:

- La **complejidad de la red NC** ('Network Complexity'). Define el número medio de relaciones de precedencia no redundantes por actividad.

- El **factor de recurso RF** ('Resource Factor'). Proporciona la proporción media del número de diferentes tipos de recursos para los que cada actividad no ficticia posee una demanda de recurso no nula, i.e.,

$$\mathrm{RF} = \frac{1}{(n-2)\cdot K}\sum_{j=2}^{n-2}\sum_{k\in K}\begin{cases}1 & \text{si } r_{j,k} > 0\\ 0 & \text{en otro caso}\end{cases}$$

El RF refleja la proporción media de recursos utilizados por actividad. Es una medida de la densidad de la matriz $(r_{j,k})_{j,k}$. RF = 1 implica que cada actividad utiliza todos los tipos de recursos, mientras que cuando RF = 0 nos encontramos ante un problema sin restricciones de recursos.

- La **intensidad de recursos RS** ('Resource Strength'). Es un valor entre 0 y 1 que representa la escasez en la disponibilidad de recursos. Se utiliza de la siguiente manera. Se determina la demanda mínima de cada recurso para que exista alguna solución posible, R_k^{min} y una cantidad máxima R_k^{max} y la disponibilidad se obtiene como $R_k = R_k^{min} + RS(R_k^{max} - R_k^{min})$. R_k^{min} se calcula como $\max\{r_{j,k}, j \in \mathbf{V}\}$, y R_k^{max} como la demanda máxima del recurso k en la secuencia ES.

Para generar una instancia basta con proporcionar unos valores mínimos y máximos para cada uno de los parámetros del primer subconjunto, y un valor para cada uno de los tres parámetros del segundo subconjunto. También se tienen que fijar unos márgenes de error para estos parámetros. El programa intenta construir un grafo que cumpla todas las características especificadas, donde los valores de NC, RF y RS sean iguales a los demandados salvo los márgenes de error, y proporciona un mensaje de error si no lo consigue.

PROGEN se ha empleado para generar cuatro conjuntos de problemas, denominados **j30, j60, j90** y **j120**. Todas las instancias de estos conjuntos han sido generadas según los mínimos y máximos para los parámetros del primer subconjunto que figuran en la Tabla 1.1 (Q = número de tipos de recursos que precisa una actividad):

	K	d_j, $j \neq 1, n$	Q	$r_{j,k}$, $j \neq 1, n$	$\|Suc_1\|$	$\|Pred_n\|$	$\|Suc_j\|$, $j \neq 1, n$	$\|Pred_j\|$, $j \neq 1, n$
min	4	1	1	1	3	3	1	1
max	4	10	4	10	3	3	3	3

Tabla 1.1. Valores de los parámetros para generar j30, j60, j90 y j120.

Las instancias de cada uno de esos conjuntos utilizan el mismo número de actividades no ficticias (j30→30, j60→60, j90→90 y j120→120). Los cuatro conjuntos fueron construidos mediante un diseño factorial completo respecto de los parámetros NC, RF y RS. Los valores para NC fueron 1.5, 1.8 y 2.1, mientras que RF ∈ {0.25, 0.5, 0.75, 1}. En j120 RS ∈ {0.1, 0.2, 0.3, 0.4, 0.5}, mientras que en el resto, RS ∈ {0.2, 0.5, 0.7, 1}. Fijémonos en que las instancias con RS = 1 son triviales, puesto que la secuencia ES es posible y, por tanto, óptima. Se obtuvieron 10 réplicas (con 10 semillas diferentes) para cada combinación de parámetros, por lo que el número de instancias en j120 es 3·4·5·10 = 600 y para el resto de conjuntos es 3·4·4·10 = 480.

En diversos estudios, tanto con algoritmos heurísticos como con métodos exactos, se ha demostrado que el factor RS influye significativamente en la dificultad de las instancias (Hartmann, 1999, Kolisch, 1995 y Kolisch et al., 1995), siendo éstas más

difíciles cuanto menor sea el RS. Esto implica que el conjunto j120 es considerablemente más complicado de resolver que el resto. Si a esto le unimos que contiene más instancias que el resto y ninguna de ellas es trivial (RS = 1), este conjunto es mejor para diferenciar los algoritmos heurísticos.

Todas estas instancias están disponibles en Internet, dentro de una librería sobre secuenciación de proyectos, PSPLIB (http://www.bwl.uni-kiel.de/Prod/psplib). También se puede obtener ficheros de datos con las mejores soluciones posibles conocidas, las cotas inferiores más ajustadas para cada uno de los problemas y los autores de las mismas. Uno de los éxitos de esta librería proviene de la posibilidad de enviar las soluciones posibles vía e-mail, lo que repercute en la actualización constante de los datos de la librería.

Desde su creación en 1996, diversos autores han contribuido con sus soluciones a la mejora de las cotas, especialmente de las cotas superiores (29 investigadores en 13 grupos), y el día 10.9.2001 se podía asegurar la optimalidad de los 480 problemas de j30, de 356 problemas de j60, de 351 problemas de j90 y de 207 problemas de j120.

Una parte o la totalidad de estas instancias ha sido utilizada en la mayoría de artículos con algoritmos para el RCPSP desde la aparición de PSPLIB, tanto para evaluar los algoritmos exactos (Brucker et al., 1999 y Herroelen et al., 1998) como los algoritmos heurísticos (Baar et al., 1997, Hartmann y Kolisch, 2000, Kolisch y Hartmann, 1999, Nonobe e Ibaraki, 1999).

4. Esquemas de generación de secuencias

Los **esquemas de generación de secuencias** o **esquemas de secuenciación SGS** ('Schedule Generation Scheme') son una parte indispensable en muchos procedimientos para el RCPSP. Los SGS construyen una secuencia posible extendiendo paso a paso una secuencia parcial, que inicialmente asigna el inicio 0 a la actividad 1. Una **secuencia parcial** es una secuencia donde únicamente un subconjunto de las n actividades ha sido secuenciado. Existen dos SGS diferentes: **Serie** y **Paralelo**. Serie utiliza el incremento del número de actividades para realizar los pasos, mientras que Paralelo utiliza el incremento temporal.

4.1. Esquema de secuenciación en Serie

Serie consiste en g = 1, ..., n-2 etapas, en cada una de las cuales se selecciona una actividad y se secuencia lo más temprano posible, sin que viole ninguna relación de

precedencia ni restricción de recursos. Existen dos conjuntos disjuntos asociados a cada etapa g. El conjunto de actividades secuenciadas **Sec_g** recoge las actividades que ya han sido secuenciadas, mientras que el conjunto de actividades **D_g** almacena las actividades que son elegibles. Una actividad es **elegible** si no ha sido secuenciada y todos sus predecesores sí lo han sido, i.e., $D_g = \{j \in V \setminus Sec_g \,;\, Pred_j \subseteq Sec_g\}$.

La unión de estos conjuntos no es el conjunto V en general, ya que también puede haber actividades no secuenciadas no elegibles.

Definiendo $\widetilde{\mathbf{R}}_k(\mathbf{t})$ como la disponibilidad restante del recurso de tipo k en el instante t ($\widetilde{R}_k(t) = R_k - \sum_{\substack{j/s_j \le t, \\ f_j > t}} r_{j,k}$, $k \in K$) y $\mathbf{F_g} = \{f_j \,/\, j \in Sec_g\}$ el conjunto de tiempos de completación, se puede describir Serie de la siguiente manera:

Figura 1.6. Algoritmo SERIE

1. Inicialización: $s_1 = f_1 = 0$, $Sec_1 = \{1\}$.
2. Para g = 2 hasta n-1, hacer:
   2.1. Calcular D_g, F_g, $\widetilde{R}_k(t)$ ($k \in K$; $t \in F_g$).
   2.2. Seleccionar $j \in D_g$.
   2.3. $ES_j = \max_{h \in Pred_j} \{f_h\}$.
   2.4. $s_j = \min\{t / t \ge ES_j;\ t \in F_g;\ r_{j,k} \le \widetilde{R}_k(\tau)\ \forall k \in K,\ \tau \in [t, t+d_j[\cap F_g\}$.
   2.5. $f_j = s_j + d_j$.
   2.6. $Sec_g = Sec_{g-1} \cup \{j\}$.
3. $f_n = \max_{h \in Pred_n} \{f_h\}$.

En la primera etapa se asigna el tiempo de inicio y de finalización 0 a la actividad ficticia j = 1, y se la incluye en la secuencia parcial. Al principio de cada etapa g se calcula el conjunto de elegibles D_g, el conjunto de tiempos de finalización F_g y las disponibilidades restantes $\widetilde{R}_k(t)$. Después de ello se selecciona una actividad j del conjunto de elegibles. El tiempo de inicio de j se calcula determinando primero ES_j, el tiempo de comienzo más temprano de j según las relaciones de precedencia y la secuenciación de las actividades predecesoras de j en la secuencia parcial, y, después, calculando el tiempo de comienzo más temprano con respecto a los recursos a partir de ES_j. El tiempo de finalización de j se calcula sumando d_j a s_j.

La Tabla 1.2 muestra como Serie genera la secuencia dada en la Figura 1.3.

g	1	2	3	4	5
Sec_g	{1}	{1,4}	{1,4,5}	{1,4,5,6}	{1,2,4,5,6}
F_g	{0}	{0,1}	{0,1,2}	{0,1,2,4}	{0,1,2,4}
D_g	{2,4}	{2,5}	{2,6}	{2}	{3}
j	4	5	6	2	3

Tabla 1.2. Ejemplo del funcionamiento de Serie.

Existe un número bastante importante de procedimientos heurísticos que siguen este proceso y únicamente varían en el paso 2.2, el modo en seleccionar la actividad a secuenciar.

4.2. Esquema de secuenciación en Paralelo

Este esquema de secuenciación realiza incrementos de tiempos. A cada iteración g le corresponde un tiempo de secuenciación t_g. Las actividades que han sido elegidas para secuenciarse hasta la iteración g son elementos del conjunto de completadas $\mathbf{C_g}$ o del conjunto de activas $\mathbf{A_g}$. C_g contiene todas las actividades que han sido completadas hasta t_g, i.e, $C_g = \{j \in \mathbf{V} / f_j \leq t_g\}$ y el conjunto de activas A_g, para $g \neq 0$, contiene a las actividades que están secuenciándose en t_g, i.e., $A_g = A(t_g) = \{j \in \mathbf{V} / s_j \leq t_g < f_j\}$. Como caso especial, y para simplificar la exposición del algoritmo, definiremos $A_0 = \{1\}$. El conjunto de actividades elegibles $\mathbf{D_g}$ contiene todas las actividades que pueden ser secuenciadas de acuerdo con las relaciones de precedencia y las restricciones de recursos en t_g, es decir, $D_g = \{j \in \mathbf{V} \setminus (C_g \cup A_g); Pred_j \subseteq C_g$ y $r_{j,k} \leq \widetilde{R}_k(t_g)\ \forall k \in K\}$. Una descripción de Paralelo es la siguiente:

Figura 1.7. Algoritmo PARALELO

1. Inicialización: $g = 0$, $f_1 = 0$, $A_0 = C_0 = \{1\}$, $\widetilde{R}_k(0) = R_k$, $\forall k \in K$.
2. Mientras $|A_g \cup C_g| \leq n-2$, hacer
   - 2.1. $g = g+1$.
   - 2.2. $t_g = \min_{j \in A_{g-1}} \{f_j\}$.
   - 2.3. Calcular C_g, A_g, $\widetilde{R}_k(t_g)$, D_g.
   - 2.4. Mientras $D_g \neq \emptyset$, hacer
     - 2.4.1. Seleccionar $j \in D_g$.
     - 2.4.2. $s_j = t_g$.
     - 2.4.3. $f_j = s_j + d_j$.
     - 2.4.4. Actualizar $\widetilde{R}_k(t_g)$, A_g, D_g.
3. $f_n = \max_{h \in Pred_n} \{f_h\}$

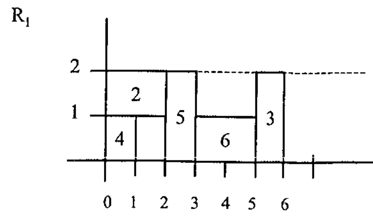


Figura 1.8

g	1	1	2	3	4	5
t_g	0	0	1	2	3	5
A_g	{1}	{4}	{2}	∅	∅	∅
C_g	{1}	{1}	{1,4}	{1,2,4}	{1,2,4,5}	{1,2,4,5,6}
D_g	{2,4}	{2}	∅	{3,5}	{3,6}	{3}
j	4	2	-	5	6	3

Tabla 1.3. Ejemplo del funcionamiento de Paralelo.

Al principio del algoritmo se fija el tiempo de secuenciación a 0, se asigna la actividad ficticia 1 al conjunto de las actividades completadas y se toma la disponibilidad de recursos en 0 como la capacidad total. Cada iteración consiste en dos etapas. En la primera etapa, (2.1 - 2.3), se determina el próximo tiempo de secuenciación t_g, los conjuntos asociados C_g, A_g, D_g y la capacidad disponible restante $\widetilde{R}_k(t_g)$. En la etapa segunda, (2.4), se secuencia un conjunto de actividades en t_g. La Tabla 1.3 muestra la generación mediante Paralelo de la secuencia de la Figura 1.8 para el proyecto de la Figura 1.1. Notar que Paralelo puede tener menos de n-2 etapas, pero que hay exactamente n–2 selecciones (tantas como actividades no ficticias) que se deben realizar.

5. Comentarios sobre Serie y Paralelo. Tipos de secuencias

Los dos esquemas de secuenciación generan siempre secuencias posibles para el RCPSP, óptimas si consideramos únicamente el problema no restringido en cuanto a recursos (Kolisch, 1996b). Se ha estudiado y caracterizado qué tipos de secuencias se pueden obtener con cada uno de los esquemas.

Una secuencia posible $S = (s_1, s_2, \dots, s_n)$ se denomina **activa, activa a la izquierda** o **justificada a la izquierda** si ninguna de las actividades puede empezar antes sin retrasar alguna otra actividad. Análogamente, una secuencia posible se denomina

activa a la derecha o **justificada a la derecha** si ninguna de las actividades puede comenzar después sin retrasar alguna otra actividad o alargar la longitud de la secuencia.

Por otra parte, una secuencia posible se denomina **sin retraso** ('non-delay') si ninguna de las actividades puede empezar antes sin retrasar alguna otra actividad, aunque se permita la interrupción de actividades (suponiendo que cada unidad de duración de cada actividad se secuencia sin interrupción). La secuencia de la Figura 1.3 es activa, mientras que la secuencia de la Figura 1.8 es sin retraso.

Kolisch, 1996b, demostró que Serie genera secuencias activas, mientras que mediante Paralelo se obtienen secuencias sin retraso. En ambos casos se pueden construir todas las soluciones del tipo correspondiente.

El conjunto de las secuencias sin retraso es un subconjunto del conjunto de las secuencias activas por lo que tiene, en general, un cardinal menor. Pero tiene una desventaja importante, y es que puede no contener ninguna solución óptima para una medida regular, algo que no ocurre con el conjunto de las secuencias activas. En PR1, la única solución óptima es la secuencia de la Figura 1.3, que es activa pero no es sin retraso. Kolisch demostró que de 298 problemas del conjunto j30 (cf. apartado 3 y Kolisch et al., 1998), sólo 175, es decir, el 58.72% tenían una solución óptima que era sin retraso. Sin embargo, hay indicios para pensar que la calidad media de las soluciones sin retraso es mejor que la de las soluciones activas, al menos cuando n es lo suficientemente grande (cf. Hartmann y Kolisch, 2000).

Últimamente se han propuesto modificaciones para Paralelo (Cho y Kim, 1997, Leon y Ramamoorthy, 1995), que consisten básicamente en no completar el paso 2.4 en determinados casos, i.e., dejar alguna actividad sin secuenciar en t_g aun cuando existen suficientes recursos para ello. El objetivo es obtener un SGS híbrido que combine las ventajas de Serie y Paralelo. Otra posibilidad similar (Möhring et al., 2000) consiste en restringir el conjunto de elegibles que puede secuenciarse en Paralelo, de acuerdo con unas ciertas prioridades asociadas a las actividades (cf. apartado 8.3).

La complejidad algorítmica de Serie y de Paralelo es $O(n^2K)$ (Kolisch y Hartmann, 1999).

6. REPRESENTACIÓN DE LAS SOLUCIONES

Se han propuesto diferentes representaciones de las soluciones del RCPSP, en especial a partir de la incorporación de las técnicas metaheurísticas como alternativa de resolución. Estas representaciones condicionan las técnicas que se van a poder emplear dentro de un algoritmo basado en una de ellas.

Algunos de los elementos más importantes a tener en cuenta en una representación son: el decodificador, el modo de obtener una solución a partir de la representación y el coste computacional de obtenerla; el subconjunto de soluciones posibles que podemos alcanzar; la calidad de ese conjunto y si siempre contiene algún óptimo; la redundancia, el número de representaciones diferentes que codifican la misma solución.

En la mayoría de heurísticos el decodificador es Serie o Paralelo (o versiones de los mismos), especialmente el primero.

6.1. Lista de actividades

Una **lista de actividades** es una permutación de las actividades $\lambda = (j_1\ j_2\ ...\ j_n)$ compatible con respecto a las relaciones de precedencia, es decir, a cada actividad le corresponde una posición u orden mayor que el de cualquiera de sus predecesoras. En particular $j_1 = 1$ y $j_n = n$ $\forall\lambda$ lista de actividades.

Serie puede transformar cualquier lista de actividades en una secuencia posible. Para ello basta cambiar el paso 2.2. 'Seleccionar $j \in D_g$' por '$j = j_g$' (Hartmann, 1997). Serie transforma la lista de actividades λ = (1 4 5 2 6 3 7) para PR1 en la secuencia de la Figura 1.3. Paralelo también se puede aplicar, calculando un vector de prioridades (cf. apartado 6.2) a partir de la lista de actividades, mediante la fórmula $p_i = \text{orden}(i,\lambda)$ (orden o posición de i en λ), y seleccionando aquella actividad en D_g de menor prioridad p_i. Esta modificación conlleva que, generalmente, no se elegirán las actividades para secuenciarse en el orden en que marca la lista de actividades por lo que, habitualmente, se utiliza Serie como esquema de secuenciación cuando se implementa esta representación.

Con esta codificación y Serie (Paralelo) se pueden obtener todas las secuencias activas (sin retraso). En general, una secuencia activa admite más de una representación como lista de actividades y, a priori, no es controlable la redundancia

porque, dada una lista de actividades λ, no se sabe qué secuencia codifica antes de secuenciarla. La lista λ = (1 4 5 2 3 6 7) también codifica la solución de la Figura 1.3.

6.2. Vector de prioridades

Esta representación (denominada en la literatura en inglés como 'random key', 'priority value' y 'problem-space based') está basada en un vector de n números (habitualmente reales y, en múltiples ocasiones, entre 0 y 1), $\rho = (p_1, p_2, \ldots, p_n)$, que asigna el número p_j a la actividad j.

Para obtener una secuencia a partir de un vector de prioridades es suficiente aplicar cualquier SGS seleccionando en cada etapa g aquella actividad j con el p_j mayor (o menor, según los casos) de entre el conjunto de elegibles D_g. Los valores p_j realizan el papel de las prioridades que se calculan mediante una regla de prioridad (cf. apartado 8.1). Si aplicamos Serie a ρ = (1, 0.75, 0.1, 0.9, 0.8, 0.5, 0) en PR1 (escogiendo el de mayor prioridad) obtendremos la secuencia de la Figura 1.3.

Fijémonos en que el orden en que se secuencian las actividades está determinado por las magnitudes relativas de las prioridades, no por sus valores absolutos. Dicho de otro modo, la calidad de la secuencia está determinada por la importancia relativa de las actividades indicada por las prioridades asignadas.

Para cada secuencia (activa en el caso de que empleemos Serie, sin retraso si empleamos Paralelo) existen infinitas representaciones como vector de prioridades. La seguridad de poder acceder a alguna solución óptima dependerá de si empleamos Serie o Paralelo como decodificador.

Cualquier vector de números reales que cumpla que si el inicio de la actividad i es menor que el de la actividad j, p_i es mayor que p_j es una codificación de una secuencia (activa si vamos a utilizar Serie, sin retraso si Paralelo) dada, si se escogen primero las actividades con mayor prioridad.

6.3. Vector de reglas de prioridad

La representación mediante un **vector de reglas de prioridad** consiste en una lista $\pi = (\pi_1, \pi_2, \ldots, \pi_n)$, donde cada π_i es una regla de prioridad (rdp, cf. 8.1). Tanto Serie como Paralelo pueden usarse como procedimientos de decodificación, seleccionando la actividad i-ésima a ser secuenciada de acuerdo con la rdp π_i.

En esta representación pueden haber, en general, secuencias posibles que no se pueden codificar por lo que, en particular, el conjunto de secuencias accesibles mediante esta representación puede no contener ningún óptimo (cf. Hartmann, 1997). Una secuencia puede admitir más de una representación como vector de prioridades.

6.4. Vector de traslación

Una secuencia puede ser codificada por un **vector de traslación** $\sigma = (\sigma_1, \sigma_2, ..., \sigma_n)$, donde σ_j es un entero no negativo. Como procedimiento de codificación y decodificación se utiliza una modificación de la recursión clásica hacia adelante (cf. apartado 2), donde el tiempo de comienzo de la actividad j, s_j, se calcula como el máximo de los tiempos de finalización de sus predecesores más la traslación σ_j de la actividad j. Es decir, $s_1=0$ y $s_j = \max\{s_h + d_h / h \in Pred_j\} + \sigma_j$ para $j = 2, ..., n$. La traslación tiene un significado muy claro, representa lo alejado que se encuentra el inicio de una actividad j con respecto al máximo de las finalizaciones de sus predecesoras (que es ES_j, el primer instante en que se podría secuenciar j sin violar las relaciones de precedencia).

Las secuencias obtenidas de este modo pueden no ser posibles, debido a que las restricciones de recursos no se cumplen en general. Los algoritmos basados en esta representación deben compensar esta falta de atención a los recursos, por ejemplo penalizando en la función objetivo la violación de las restricciones de recursos.

Como contraprestación todas las soluciones posibles son accesibles, entre ellas todas las óptimas, y no existe redundancia.

6.5. Vector de tiempos de inicio

La codificación de una solución se realiza mediante un vector de enteros no negativos $\tau = (t_1,...,t_n)$, donde t_j representa el tiempo de comienzo de la actividad j. Es decir, se 'codifica' la solución mediante la misma solución. Obviamente no existe redundancia y todas las soluciones son accesibles, incluso las imposibles.

6.6. Representación mediante esquemas de secuencias

Un **esquema de secuencias** ('schedule scheme') (C,D,N,F) consiste en cuatro conjuntos disjuntos de relaciones (pares de actividades). $(i,j) \in C$ implica que la actividad i tiene que finalizar antes de que comience la j (**conjunciones**). $(i,j) \in D$

implica que la realización de las actividades i y j no pueden coincidir en el tiempo (**disyunciones**). (i,j) ∈ N implica que las actividades i y j tienen que procesarse en paralelo en al menos una unidad de tiempo (**relaciones paralelas**). Para las actividades i y j con (i,j) ∈ F no existen restricciones (**relaciones flexibles**).

Un esquema de secuencias representa aquellas secuencias no necesariamente posibles donde todas las relaciones exigidas se cumplen. Debido a que el problema de determinar si existe una secuencia posible dentro de un esquema de secuencias es NP-duro (Krämer et al., 1995), el decodificador que se utiliza (Baar et al., 1997) construye una secuencia posible no necesariamente activa donde se satisfacen todas las relaciones de C y D y un número "grande" de las relaciones paralelas de N. Esto quiere decir que se pueden obtener secuencias que no correspondan a los esquemas de secuencias con lo que se está trabajando.

En esta representación, la redundancia existe en el sentido de que una secuencia pertenece, en general, a varios esquemas de secuencias por lo que podría ser alcanzada a partir de aplicar un cierto decodificador a dos o más de esas representaciones.

Para cada secuencia posible existe al menos un esquema de secuencias que la contiene. Sin embargo, puede ocurrir que no exista ningún esquema de secuencias para el que el heurístico empleado proporcione una solución posible dada. En ese caso no se podría asegurar en general la posibilidad de encontrar la solución óptima.

7. TÉCNICAS METAHEURÍSTICAS

Antes de describir los heurísticos empleados para el RCPSP realizaremos un resumen de las técnicas metaheurísticas que han sido utilizadas para crear algoritmos para el RCPSP y algunas otras que serán de utilidad en posteriores capítulos. Para entender mejor estas técnicas empezaremos por analizar el concepto de búsqueda local. Supondremos que el problema que estamos considerando es el de minimizar una cierta función objetivo f, con valores reales. Consideraremos que una solución x es **mejor** que otra solución y si f(x) < f(y).

7.1. Búsqueda local

En una gran cantidad de problemas de optimización, entre ellos el RCPSP, existen formas sencillas de construir soluciones posibles. Sin embargo, las soluciones

obtenidas mediante estos métodos constructivos suelen ser en general de una calidad moderada, lo que lleva a desarrollar algoritmos que actúen sobre una solución inicial dada e intenten mejorarla.

En cada **procedimiento de búsqueda local**, también llamados de mejora, cada solución x del conjunto de soluciones posibles X tiene un conjunto de soluciones asociadas $N(x) \subseteq X$, denominado **entorno** o vecindad de x. Cada solución x' perteneciente a N(x) se puede obtener directamente a partir de x mediante una operación llamada **movimiento**. Estos métodos se basan en explorar los entornos de las soluciones.

Un proceso de búsqueda local parte de una solución inicial x_0, calcula su entorno $N(x_0)$ y escoge una nueva solución x_1 en él, es decir, realiza el movimiento m_1 que aplicado a x_0 proporciona x_1. Este proceso se aplica reiteradamente, obteniendo

$$x_0 \xrightarrow{m_1} x_1 \xrightarrow{m_2} x_2 \xrightarrow{m_3} \ldots \xrightarrow{m_{k-1}} x_{k-1} \xrightarrow{m_k} x_k .$$

Un procedimiento de búsqueda local queda completamente determinado al especificar el entorno y el criterio de selección de una solución dentro del entorno, siempre y cuando no se empleen componentes aleatorias ni en la definición del entorno ni en la selección.

Una de las estrategias de búsqueda local más utilizadas es la **BFS** ('Best Fit Strategy', Estrategia de la Mejor Mejora). Este método consiste en aplicar en cada iteración el mejor movimiento posible respecto de la función objetivo, hasta que la solución actual no se pueda mejorar. Así, en la etapa k, el algoritmo calcula la solución $x_{k+1} \in N(x_k)$ de manera que no existe ningún elemento en ese entorno mejor que x_{k+1}. Si x_{k+1} mejora a x_k, el método acepta esta nueva solución (se "mueve" a ella), y realiza una nueva iteración, con x_{k+1} como solución de partida. El algoritmo termina cuando ninguna solución del entorno de la solución actual en esa iteración mejora dicha solución (lo que se correspondería con $f(x') \geq f(x_k)\ \forall x' \in N(x_k)$).

Otra estrategia ampliamente utilizada es la **FFS** ('First Fit Strategy', Estrategia de la Primera Mejora). En esta estrategia se dispone de un mecanismo para determinar un orden en el cálculo de las soluciones del entorno de una solución. FFS consiste en calcular según ese orden las soluciones del entorno de la solución actual y aceptar la primera solución que la mejore. El criterio de terminación es el mismo que en el BFS.

En ambos métodos la solución final x cumple $f(x) \leq f(y)\ \forall y \in N(x)$. En ese sentido se considera que se obtiene un **óptimo local con respecto al entorno definido**.

Los métodos de búsqueda local suelen ser muy rápidos y proporcionan soluciones relativamente cercanas al óptimo global, al menos para tamaños de problemas pequeños.

7.2. Temple simulado

El **temple simulado** (SA, 'Simulated Annealing') fue introducido por Kirkpatrick, Gelatt y Vecchi en 1983. Este procedimiento se basa en una analogía con el comportamiento de un proceso físico en el que se enfría un sólido fundido.

Empezando con una solución inicial x, se genera una solución x' $\in$ N(x) (previamente se ha definido un entorno para cada solución). Si x' es mejor que x, se acepta x' y se continua el proceso con x'. En otro caso, si x' no mejora x, x' se acepta según una cierta probabilidad. Esta probabilidad depende de la diferencia en la función objetivo entre x y x' y de un parámetro llamado **temperatura**, que varía durante la ejecución del algoritmo. A mayor diferencia f(x') - f(x) menor probabilidad de aceptación, y cuanto menor es la temperatura más difícil es aceptar una solución que no mejore. El parámetro temperatura es, en general, alto cuando comienza el algoritmo, de manera que se permiten empeoramientos en la función objetivo con cierta frecuencia. A medida que transcurre el algoritmo se reduce la temperatura paulatinamente y, por tanto, se limitan gradualmente los movimientos a movimientos de mejora. Entre los posibles criterios de parada está el de definir a priori un cierto límite inferior de la temperatura, y finalizar el algoritmo la primera vez que ésta alcance ese límite (**congelación**).

SA se puede ver como una extensión del FFS, puesto que acepta todas las mejoras de forma inmediata, aunque a diferencia de esa estrategia sí acepta movimientos de no mejora. De este modo, se intenta no 'caer' demasiado pronto en óptimos locales respecto del entorno definido. Un esquema general de un SA podría ser el siguiente:

Figura 1.9. SA(solución inicial,T inicial,r,L)

1. x = solución inicial.
2. Mientras no se cumpla el criterio de parada:
   - 2.1. Realizar L veces:
     - 2.1.1. Calcular x' $\in$ N(x). Sea dif = f(x') – f(x).
     - 2.1.2. Si (dif < 0): x = x'.
     - 2.1.3. En otro caso: x = x' con probabilidad igual a $e^{-dif/T}$.
   - 2.2. T = rT.
3. Devolver la mejor solución calculada en todo el algoritmo.

El parámetro T representa la temperatura y r es un parámetro real, que se supone menor que 1, de manera que cada vez que realizamos T = rT estamos disminuyendo la temperatura.

7.3. Búsqueda tabú

La **búsqueda tabú** (TS, 'Tabu Search') fue desarrollada por Glover (Glover, 1989 a y b) y es una extensión de la búsqueda local BFS que intenta no quedar atrapada en un óptimo local.

Un algoritmo TS comienza desde una solución posible $x_0 = x$ y, en cada iteración k, se mueve desde una solución x_k a una solución del entorno de x_k mediante un movimiento. Un TS simple realiza el movimiento a la mejor solución x_{k+1} de $N(x_k)$ (según la función de evaluación), aunque x_{k+1} sea peor que x_k, pero asegurándose hasta cierto punto que x_{k+1} no haya sido visitada con anterioridad, para evitar el ciclado. Para ello se restringen los movimientos a aplicar, algunos de los cuales son prohibidos (se definen como **tabú**). Estos movimientos, usualmente repetición e/o inversos de movimientos aplicados anteriormente, se almacenan ordenados (respecto a la iteración en que son definidos tabú) en una lista denominada tabú. Esta lista es limitada; cuando está llena y se introduce un elemento nuevo el movimiento más antiguo almacenado desaparece de ella y, por tanto, deja de ser tabú.

Las soluciones a las que se accede mediante movimientos tabú se dice que tienen estatus tabú. A menudo se incluyen condiciones (llamados **niveles de aspiración**) que permiten moverse a una de estas soluciones, normalmente cuando ésta mejora a la mejor solución almacenada.

Este almacenamiento y uso de la información de las soluciones recientemente visitadas se denomina empleo de la memoria a corto plazo. Las memorias en un plazo intermedio y a largo plazo se pueden emplear para **intensificar** (buscar en regiones "atractivas" ya visitadas) y **diversificar** (buscar en regiones nuevas) la búsqueda. A menudo se utilizan los atributos de las soluciones para guiar esa búsqueda. En la descripción de nuestros algoritmos emplearemos esos términos, aún en los casos en que no se trabaje con un algoritmo tabú.

En el **reencadenamiento de trayectorias** ('path relinking') se dispone de una solución inicial A y una solución guía B. El proceso consiste en ir aplicando movimientos a A que le acerquen a B, introduciendo cada vez más características de B en A. Así se construyen k soluciones $A_1, A_2, ..., A_k$ que trazan un camino de A a B en el conjunto de

soluciones (encadenan A y B). También se puede trabajar con un conjunto de soluciones guía.

7.4. Algoritmos genéticos

Los **algoritmos genéticos** (GA, 'Genetic Algorithms', Holland, 1975) son algoritmos de búsqueda basados en los mecanismos de selección natural y la genética. Establecen una analogía entre el conjunto de soluciones del problema a resolver y el conjunto de individuos de una población natural, codificando la información de cada solución en una cadena ('**string**') de números denominada **cromosoma**. El potencial de un cromosoma como solución se mide mediante una función de evaluación (en la que forma parte importante la función objetivo del problema), y se denomina **calidad** ('fitness') de la solución.

Un GA calcula un conjunto de cromosomas llamado **población inicial**, y simula la evolución de esa población repitiendo los siguientes pasos mientras no se cumpla un determinado criterio de parada:

i) se eligen dos cromosomas de la población (**padres**), de acuerdo con su calidad;

ii) se obtienen y evalúan uno o dos cromosomas nuevos (**hijos**) a partir de aplicar un operador a los padres. Al operador se le denomina **operador de cruce** y debe combinar las características de los padres al crear los hijos.

iii) se reemplazan individuos de la población mediante los recién creados.

Entre los operadores de cruce más utilizados para strings binarios se hallan los operadores de uno y dos puntos de corte, y el de cruce uniforme.

Dados dos cromosomas progenitores, una madre $M = (m_1, m_2, ..., m_n)$ y un padre $P = (p_1, p_2, ..., p_n)$, el **operador de un punto de corte** consiste en escoger aleatoriamente un entero q (denominado **punto de corte**) con $1 \leq q < n$. Mediante este operador se obtienen dos hijos, a y b, definidos de la siguiente manera:

$a_i := m_i, i = 1, ..., q$; $a_i := p_i, i = q+1, ..., n$; $b_i := p_i, i = 1, ..., q$; $b_i := m_i, i = q+1, ..., n$.

Es decir, se copian las q primeras posiciones de la madre en las q primeras posiciones del primer hijo, y los n-q últimos bits los obtiene el primer hijo de los últimos n-q del padre. La obtención del segundo hijo se realiza intercambiando los roles entre la madre y el padre.

La elección de los progenitores y de los individuos a eliminar se realiza de forma aleatoria, pero los cromosomas de mayor calidad tienen más posibilidades de ser seleccionados como padres, y menos de ser eliminados (la ley de supervivencia del más fuerte) al introducir el hijo o los hijos en la población. En algunas ocasiones, gobernadas por la aleatoriedad, un cromosoma sufre un cambio pequeño en sus genes, simulando la **mutación** en la genética (de donde hereda el nombre), lo que introduce diversidad en la población.

Los principales elementos que caracterizan una aplicación GA son el esquema de codificación de las soluciones, los operadores de cruce que se utilizan y la función de evaluación. Otras decisiones también importantes que han de realizarse para construir un GA son el modo de obtener la población inicial, los operadores de mutación (cuáles y cuándo aplicarlos), la regla de elección de los padres y de eliminación de uno o varios individuos al entrar el hijo o los hijos calculados y el criterio de terminación del algoritmo.

7.5. GRASP

El procedimiento GRASP ('Greedy Randomized Adaptive Search Procedure', Feo y Resende, 1989) consta de dos fases, la fase de construcción y la de mejora.

En la fase de construcción se construye de forma iterativa una solución posible para el problema; en cada iteración se añade un elemento a la solución, llevándose a cabo la selección de ese elemento de la siguiente manera. Se dispone de una función de evaluación de las distintas alternativas. Una posibilidad agresiva pero miope ('greedy') sería la de escoger en cada iteración el elemento con mejor valor en esa función. Lo que se realiza es emplear la probabilidad ($\rightarrow$ 'randomized') y la función para elegir el elemento, de manera que sea más probable introducir aquellos elementos con mejor evaluación. Una vez escogido el elemento, los valores de los elementos restantes cambian, se adaptan a la nueva situación ($\rightarrow$ 'adaptive').

Una vez construida la solución posible, se le aplica una búsqueda local o una función de mejora, y estos dos pasos se repiten un número de veces predeterminado. Al emplear componentes aleatorios, en general se obtendrán diferentes soluciones en cada iteración.

7.6. Búsqueda dispersa

La búsqueda dispersa ('scatter search') es un procedimiento evolutivo consistente en 5 elementos (cf. Glover, 1998).

1. Un método de generación diversificada para generar una colección de soluciones de prueba diversas.

2. Un método de mejora que transforma una solución de prueba en una o más soluciones mejoradas.

3. Un método para actualizar el conjunto de referencia, que construye y mantiene un conjunto de referencia consistente en b (generalmente pequeño) "mejores" soluciones encontradas.

4. Un método de generación de subconjuntos, que produce subconjuntos del conjunto de referencia a partir de los cuales crear soluciones combinadas. El más usual es el que genera todos los subconjuntos de orden 2.

5. Un método de combinación de soluciones que transforma un subconjunto dado de soluciones en una o más soluciones. El método de combinación es análogo al operador de cruce en los algoritmos genéticos, pero debe ser capaz de combinar 2 o más soluciones.

7.7. El metaheurístico de las hormigas

En el metaheurístico de las hormigas ('Ant Colony Optimization', Dorigo y Di Caro, 1999) un cierto número de generaciones de hormigas artificiales exploran el espacio de soluciones en búsqueda de buenas soluciones. Cada hormiga de cada generación construye una solución paso a paso a través de diferentes decisiones probabilísticas. En general, las hormigas que encuentran buenas soluciones marcan los caminos recorridos poniendo una cierta cantidad de feromona en las aristas del camino. Las hormigas de la siguiente generación son atraídas por la feromona, por lo que buscarán cerca de buenas soluciones. Además de por las feromonas, las hormigas son usualmente guiadas también por heurísticos específicos del problema.

8. Heurísticos para el RCPSP

Los heurísticos que han sido o están siendo más utilizados para el RCPSP se pueden dividir en dos metodologías: heurísticos basados en rdps (apartado 8.1) y algoritmos obtenidos a partir de metaheurísticos (apartado 8.2). Otros enfoques menos utilizados están recopilados en el apartado 8.3. En Davis, 1966 y 1973, Herroelen, 1972 y Kolisch y Hartmann, 1999, se pueden encontrar recopilaciones de los enfoques heurísticos para el RCPSP.

8.1. Heurísticos basados en reglas de prioridad

Una **regla de prioridad** define un procedimiento para seleccionar una actividad del conjunto de elegibles D_g para cada g (cf. apartado 4).

Una rdp está formada por tres componentes, (i) una función que asigna a cada actividad j del conjunto de elegibles D_g un valor real v(j), (ii) la manera de seleccionar el extremo del conjunto, bien la actividad que posea el menor valor v(j) o la que posea el mayor valor v(j) y (iii) la regla que permite desempatar entre las actividades de igual valor.

Una de las maneras más habituales de desempate es la de elegir, de entre las actividades con mismo valor, aquella con etiqueta menor. Se dice entonces que se está utilizando una regla "pura".

Las reglas de prioridad se pueden clasificar según diferentes criterios:

i) Con respecto al tipo de información que utilizan.

   Podemos distinguir reglas basadas en la red, en el tiempo, en los recursos, en cotas superiores y en cotas inferiores.

ii) Con respecto a la cantidad de información utilizada.

   Si emplean información únicamente de la actividad j para calcular v(j), se denominan reglas **locales**; si emplean otro tipo de información, **globales**.

iii) Con respecto a las actualizaciones de v(j)

   Una regla se dice **dinámica** si v(j) puede cambiar durante las iteraciones del SGS y **estática** en otro caso.

iv) Con respecto al SGS utilizado.

Existen reglas que sólo admiten uno de los SGS mientras que otras se pueden emplear con los dos.

Para una lista de artículos que tratan sobre rdps para el RCPSP, consultar Kolisch y Hartmann, 1999, mientras que en Alvarez-Valdés y Tamarit, 1989b, Kolisch, 1995 y Lawrence, 1985, se pueden encontrar algunas revisiones de las rdps para el RCPSP. La Tabla 1.4 recoge algunas de las rdps más conocidas y utilizadas en la literatura, así como sus definiciones. En la Tabla 1.5 se resumen sus características.

La regla MTS emplea **TSuc**$_j$, el conjunto de todos los sucesores de la actividad j (análogamente se define TPred$_j$, el conjunto de todos los predecesores de j). Las reglas WCS, IRSM y RSM emplean **AP** = $\{(i,j) \in D_g x D_g \,/\, i \neq j\}$, el conjunto de todos los pares de actividades (i,j) que están en el conjunto de elegibles D_g. Finalmente, las reglas IRSM y WCS utilizan **E(i,j)**, el tiempo de comienzo de la actividad j más temprano de acuerdo con las relaciones de precedencia y los recursos, si la actividad i comienza en el tiempo de secuenciación t_g. La rdp WRUP es en realidad una familia de reglas, una para cada par de valores que se asignen a w_1 y w_2 (se debe cumplir $w_1 + w_2 = 1$).

Acrónimo		v(j)
GRPW	Greatest Rank Positional Weight	$d_j + \sum_{i \in Suc_j} d_i$
IRSM	Improved Resource Scheduling Method	$\max_{(i,j) \in AP}\{0, E(j,i)-(LF_i-d_i)\}$
LFT	minimum Latest Finish Time	LF_j
LST	minimum Latest Start Time	LF_j-d_j
MSLK	Minimum Slack	LF_j-EF_j
MTS	Most Total Successors	$\lvert TSuc_j \rvert$
RSM	Resource Scheduling Method	$\max_{(i,j) \in AP}\{0, t_g+d_j-(LF_i-d_i)\}$
SPT	Shortest Processing Time	d_j
WCS	Worst Case Slack	$LF_j - d_j - \max_{(i,j) \in AP}\{E(i,j)\}$
WRUP	Weighted Resource Utilization ratio and Precedence	$w_1 \lvert Suc_j \rvert + w_2 \sum_{k \in K} \frac{r_{j,k}}{R_k}$

Tabla 1.4. Acrónimos y definiciones de algunas rdps.

Acrónimo	Información que utilizan	local/global	estática/ dinámica	Serie/Paralelo	min/max
GRPW	red, tiempo	G	E	S/P	max
IRSM	red, tiempo, cota inferior, recursos	G	D	P	min
LFT	red, tiempo, cota inferior	G	E	S/P	min
LST	red, tiempo, cota inferior	G	E	S/P	min
MSLK	red, tiempo, cota inferior, recursos	G	D	S	min
MTS	red	G	E	S/P	max
RSM	red, tiempo, cota inferior	G	D	P	min
SPT	tiempo	L	E	S/P	min
WCS	red, tiempo, cota inferior, recursos	G	D	P	min
WRUP	red, recursos	G	E	S/P	max

Tabla 1.5. Características de las rdps definidas en la Tabla 1.4.

Los heurísticos basados en rdps emplean uno o ambos SGS para construir una o más secuencias. Hasta hace pocos años, estos heurísticos eran los más empleados y efectivos (cf. Kolisch, 1996b), aunque en los últimos años han sido superados por algoritmos obtenidos a partir de metaheurísticos (cf. apartado 8.2).

Los heurísticos basados en rdps se dividen en dos tipos, los que generan una única secuencia o **métodos de un paso**, y los que generan más de una, en cuyo caso se denominan **métodos multipaso**.

Los métodos de un paso (consistentes en aplicar una rdp con un SGS) fueron los primeros heurísticos que se utilizaron para resolver el RCPSP, fruto de la experiencia positiva de la utilización de rdps en problemas de secuenciación de máquinas. Su rapidez, facilidad de implementación y de comprensión llevaron a estos métodos a ser los más utilizados y comentados.

De hecho, no se ha dejado completamente de proponer nuevas rdps, cada vez más elaboradas (Kolisch, 1996a, Özdamar y Ulusoy, 1996a, Thomas y Salhi, 1997 y Ulusoy y Özdamar, 1994). Se sigue pensando (cf. Kolisch y Hartmann, 1999) que es productivo investigar en rdps y en heurísticos basados en ellas, puesto que son una herramienta útil para proporcionar soluciones iniciales a los algoritmos basados en metaheurísticos (cf. apartado 8.2) con bajo coste computacional.

El fracaso en encontrar una rdp que dominara claramente a las demás, agravado por la existencia, para cada rdp, de problemas donde no funcionaban demasiado bien, desembocó en los métodos multipaso, los cuales se pueden dividir en métodos que incluyen varias rdps, métodos de secuenciación adelante-atrás y métodos de muestreo.

Los **métodos que incluyen varias rdps** se propusieron a partir de la experiencia de que unas rdps funcionaban mejor que otras en unos problemas y otras, en otros, y aprovechando la rapidez con que se ejecuta una rdp. Consisten en generar m secuencias utilizando m rdps diferentes, en un intento de aprovechar las cualidades de cada una por separado (Boctor, 1990), y de tener una cierta calidad garantizada en todos los problemas. También se puede intentar juntar sus cualidades por medio de combinaciones convexas de m rdps. Este método consiste en calcular la prioridad de j según $v(j)=\sum_{i=1}^{m} w_i\, v_i(j)$, con $w_i \geq 0\ \forall i$ y $\sum_{i=1}^{m} w_i = 1$, donde $v_i(j)$ es el valor obtenido de aplicar la rdp i-ésima sobre la actividad j. Ha sido utilizado entre otros, por Thomas y Salhi, 1997 y Ulusoy y Özdamar, 1989.

Los **métodos de secuenciación adelante-atrás** emplean la secuenciación en diferentes direcciones, no sólo la secuenciación hacia adelante como el resto de procedimientos. Dado que las técnicas empleadas en estos métodos serán claves en el desarrollo de nuestros heurísticos, vamos a describir más detalladamente algunos de sus aspectos.

Mientras que la secuenciación hacia adelante consiste en aplicar un SGS como está descrito en el apartado 4, la secuenciación hacia atrás consiste en aplicarlo a la red de precedencias inversa (grafo **inverso**), en la que i es predecesor de j si y sólo si j es predecesor de i en la red original, y donde la actividad ficticia n (1) se convierte en el inicio (fin) del proyecto. Este tipo de secuenciación se puede ver de dos formas equivalentes. En la primera, el proceso de secuenciar comienza asignando un tiempo de finalización a n, una cota superior de la duración del proyecto UB_T (por ejemplo, la suma de las duraciones). Después, se procede de forma análoga a la secuenciación hacia adelante pero de forma simétrica, una actividad es elegible si sus sucesores han sido secuenciados, al procesar una actividad se realiza lo más tarde posible, etc. Una vez finalizada la secuenciación, el comienzo de la actividad 1 no será, en general, 0. Para que todas las secuencias con las que se trabaja comiencen en 0 se reducen los tiempos de inicio y finalización en s_1 unidades. La secuencia obtenida S' está justificada a la derecha. En la segunda forma se trabaja con la red inversa como si fuera el proyecto original, secuenciando la actividad n en 0, y secuenciando hacia adelante – pero sobre la red inversa. La secuencia resultante S se puede convertir

fácilmente en una secuencia S^r para el proyecto original mediante la transformación $s_j^r = T(S) - f_j$. Mientras S es justificada a la izquierda en el proyecto inverso, S^r es justificada a la derecha para el original y, además, $S^r = S'$.

Existe una tercera posibilidad al secuenciar, la bidireccional. En la secuenciación bidireccional se construye una secuencia simultáneamente en las dos direcciones. Se comienza la secuencia procesando la actividad 1 en 0 y la n en UB_T. En cada iteración se selecciona, mediante una regla de prioridad, una actividad para ser secuenciada, junto con la dirección en la que se va a secuenciar. Dependiendo de si es por el método hacia adelante o hacia atrás se secuenciará lo más pronto o lo más tarde posible. En el primer caso, todas las predecesoras de la actividad habrán sido procesadas hacia adelante y, en el último, lo habrán sido todas sus sucesoras, pero hacia atrás. La secuencia final se obtiene de justificar a la izquierda las actividades secuenciadas mediante el método hacia atrás, justificándolas en orden creciente de inicio. La **justificación a la izquierda (derecha)** de una actividad $i \neq 1$ (n) en una secuencia S consiste en secuenciarla lo más pronto (tarde) posible dentro de S, sin cambiar el resto de actividades y de manera que la secuencia resultante sea posible. Si se justifican las actividades de una secuencia activa (a la derecha) en orden decreciente de los finales (creciente de los inicios) se obtiene una secuencia justificada a la derecha (izquierda).

Li y Willis, 1992, utilizan la secuenciación hacia adelante y hacia atrás junto con Serie, así como la justificación de secuencias activas y activas a la derecha. Özdamar y Ulusoy, 1996a y 1996b, emplean exclusivamente la secuenciación hacia adelante y atrás junto con Paralelo. Sólo Klein, 1998 y 2000, trabaja con la secuenciación bidireccional, además de con las otros dos direcciones. No emplea la justificación de secuencias, sólo la de las actividades procesadas hacia atrás en el caso bidireccional. Con estos tres métodos compara la eficacia de diferentes rdps con la secuenciación hacia adelante, atrás y bidireccional por separado, y crea un heurístico (Klein, 1998) en los que se combinan varias rdps y varias direcciones de secuenciación. Tormos y Lova, 2001, construyen secuencias con un método de muestreo aleatorio sesgado basado en la peor elección (ver más abajo) y a cada una de ellas la justifican a derecha e izquierda. La regla utilizada en el muestreo es la LFT y emplean tanto Serie como Paralelo.

Los **métodos de muestreo** utilizan generalmente un SGS y una rdp, y obtienen diferentes secuencias introduciendo un matiz probabilístico en la selección de las actividades. En lugar de una prioridad v(j) se calcula una probabilidad de selección

p(j), que representa la probabilidad de que se seleccione la actividad j del conjunto de elegibles D_g cuando se ha de seleccionar una actividad para secuenciar.

Dependiendo de cómo se calculen las probabilidades se puede distinguir entre muestreo aleatorio, muestreo aleatorio sesgado y muestreo aleatorio sesgado basado en la peor elección.

El **muestreo aleatorio** asigna a cada actividad del conjunto de elegibles D_g la misma probabilidad $p(j) = 1/|D_g|$. El **muestreo aleatorio sesgado** emplea directamente las prioridades para calcular la probabilidad de selección. Si la rdp selecciona la actividad con la prioridad más grande, la probabilidad se calcula como $p(j) = v(j)/(\sum_{i \in D_g} v(i))$. Si la rdp selecciona la de menor prioridad, algunas de las propuestas para calcular las probabilidades han sido:

$$p(j) = \frac{1/v(j)}{\sum_{i \in D_g} 1/v(i)}, \text{cuando } v(j) > 0\ \forall j, \text{ o } p(j) = \frac{M - v(j)}{\sum_{i \in D_g}(M - v(i))},$$

donde $M > \max\{v(j)/\, j \in D_g\}$.

Estos métodos han sido aplicados por Álvarez-Valdés y Tamarit, 1989b y Cooper, 1976.

Schirmer y Riesenberg, 1997, propusieron una modificación llamada **muestreo aleatorio sesgado normalizado** que, esencialmente, asegura que la probabilidad de selección de la actividad con la prioridad más pequeña (grande) cuando se busca el mínimo es la misma que la de la actividad con la prioridad mayor (más pequeña) si se buscara el máximo.

El **muestreo aleatorio sesgado basado en la peor elección** utiliza las prioridades indirectamente a través de unos valores denominados 'regret'. Los valores **regret**, r_j, comparan la prioridad de la actividad j con la peor elección del conjunto de elegibles:

$$r_j = \begin{cases} (\max_{i \in D_g} v(i)) - v(j) & \text{si extremo} = \min \\ v(j) - \min_{i \in D_g} v(i) & \text{si extremo} = \max \end{cases}$$

La probabilidad de selección se calcula como $p_j = \frac{(r_j + \varepsilon)^\alpha}{\sum_{i \in D_g}(r_i + \varepsilon)^\alpha}$, con $\varepsilon > 0, \alpha > 0$.

Mediante el parámetro α se puede controlar la cantidad de sesgo; un α grande

conducirá a una selección casi determinista mientras que un α = 0 producirá una selección aleatoria.

Según Kolisch, 1995, y Schirmer y Riesenberg, 1997 (quienes determinan el ε dinámicamente), este método es el mejor de entre los de muestreo.

En los últimos años se han desarrollado algoritmos basados en intentar aprovechar la información que proporciona la instancia a resolver para determinar las técnicas a aplicar. Son algoritmos multipaso autoregulables que eligen la rdp, el SGS y el sistema de muestreo a emplear en el problema. Esta elección depende del número de iteraciones efectuado y de los resultados de un análisis de ciertos parámetros del problema que se está resolviendo, que se realiza dentro del propio algoritmo (Kolisch y Drexl, 1996, Schirmer y Riesenberg, 1998). Estos algoritmos utilizan cotas inferiores para descartar secuencias parciales, acelerando así el proceso. Estos métodos son de los mejores heurísticos no obtenidos a partir de técnicas metaheurísticas (cf. Kolisch y Hartmann, 1999).

Un algoritmo multipaso distinto a los anteriores lo tenemos en Artigues et al., 2000. Proponen un procedimiento exacto de $O(n^2K)$ para insertar una actividad dentro de una secuencia parcial S de manera que la secuencia resultante aumente en longitud lo mínimo posible. Para cada actividad i, i = 2, ..., n–2, el algoritmo heurístico considera el proyecto obtenido al extraer la actividad i, aplica una rdp para ese nuevo proyecto y después añade i a la secuencia resultante mediante el procedimiento exacto.

1.2. Heurísticos basados en técnicas metaheurísticas

Al igual que en muchos otros problemas, en los últimos años se han desarrollado algoritmos basados en metaheurísticos para el RCPSP que han superado los heurísticos conocidos anteriormente (cf. Cho y Kim, 1997, Kolisch y Hartmann, 1999).

En este apartado vamos a describir únicamente alguno de los últimos algoritmos basados en técnicas metaheurísticas (no están incluidas las aproximaciones para las extensiones del RCPSP). En la Tabla 1.6 se recoge un resumen de muchos de los heurísticos existentes; en Ballestín, 1999, Hartmann, 1999, Hartmann y Kolisch, 2000 y Kolisch y Hartmann, 1999 se ofrece una descripción de algunos de los mismos.

Artículo	Metaheurístico	Representación	Decodificador
Alcaraz y Maroto, 2001	GA	lista act.	Serie adelante/detrás
Baar et al., 1997	TS	lista act.	Serie
	TS	esquema sec.	específico
Bouleimen y Lecocq, 1998	SA	Lista act.	Serie
Cho y Kim, 1997	SA	v. prioridades	Paralelo modificado
Hartmann, 1997	GA	lista act.	Serie
	GA	v. prioridades	Serie
	GA	v. rdps	Serie
Hartmann, 2000	GA	lista act.	Serie y Paralelo
Kohlmorgen et al., 1999	GA	v. prioridades	Serie
Lee y Kim, 1996	SA; TS; GA	v. prioridades	Paralelo
Leon y Ramamoorthy, 1995	FFS; BFS; GA	v. prioridades	Paralelo modificado
Merkle et al., 1999, 2000	hormigas	lista act.	Serie
Naphade et al., 1997	BFS	v. prioridades	Paralelo modificado
Pinson et al., 1994	TS; TS; TS	lista act.	Serie
Sampson y Weiss, 1993	variante SA	v. traslación	extensión recursión
Thomas y Salhi, 1998	TS	v. inicios	específico

Tabla 1.6. Resumen de las estrategias metaheurísticas para el RCPSP.

El genético de Hartmann

En 1998, Hartmann publicó un artículo donde definía y comparaba varios genéticos basados en diferentes codificaciones. El mejor de ellos, que desde ahora denominaremos el genético de Hartmann (o genético de Hartmann (1)), es capaz de mejorar (en calidad) a varios heurísticos de la Tabla 1.6 que habían sido aplicados sobre los problemas de Patterson: Cho y Kim, Lee y Kim, Leon y Ramamoorthy, Özdamar y Ulusoy, Sampson y Weiss y Thomas y Salhi. Además, fue considerado en Kolisch y Hartmann, 1999, como el mejor heurístico junto al SA de Bouleimen y Lecocq, aunque este último era inferior en j120, el conjunto más difícil. Este SA está basado, exclusivamente, en el operador de traslación simple que selecciona una actividad j_q y la inserta inmediatamente después de otra actividad j_s si las relaciones de precedencia lo permiten.

En ese artículo de Kolisch y Hartmann se incluyen varios de los mejores heurísticos para el RCPSP que no siguen un esquema metaheurístico (en especial varios del apartado anterior), y fueron dominados tanto por el SA de Bouleimen y Lecocq como por el genético de Hartmann. Teniendo en cuenta estos datos y, dado que va a jugar

un papel importante a lo largo de la tesis por diferentes motivos, vamos a describir este último con más profundidad que el resto. En la Figura 1.10 se puede ver un esquema de ese genético, donde POPsize es el tamaño de la población y nsche el número de secuencias máximo que se va a calcular.

Figura 1.10. GA_Hartmann(POPsize,nsche)

1. Construir la población inicial de POPsize individuos.
2. niter = $\lfloor nsche/POPsize \rfloor - 1$.
3. Desde i = 1 hasta niter, hacer
   - 3.1. Particionar aleatoriamente la población en parejas de padres.
   - 3.2. Para cada pareja de padres
     - 3.2.1. Obtener hijos aplicando el operador de cruce de dos puntos de corte y la mutación. Secuenciarlos mediante Serie.
     - 3.2.2. Añadir los hijos a la población.
   - 3.3. Reducir la población por selección.
4. Devolver la mejor solución.

La población inicial se construye mediante un muestreo sesgado basado en la peor elección sobre LFT y $\alpha = 1$ (ver apartado anterior), aplicado POPsize veces. El punto 3.3 se lleva a cabo seleccionando los POPsize mejores individuos, desempatando de forma aleatoria.

El operador de dos puntos de corte que se emplea es el adaptado a permutaciones. Dadas dos listas de actividades, una madre $\lambda^M = (\lambda_1^M \lambda_2^M \ldots \lambda_n^M)$ y un padre $\lambda^P = (\lambda_1^P \lambda_2^P \ldots \lambda_n^P)$, la lista de actividades hija $\lambda^H = (\lambda_1^H \lambda_2^H \ldots \lambda_n^H)$ se define como sigue: se calculan dos enteros, $1 < q_1 < q_2 < n$, y se toma $\lambda_i^H = \lambda_i^M$ para i = 1, ..., q, i.e., las primeras posiciones se toman de la madre. Las $q_2 - q_1$ posiciones siguientes provienen de actividades del padre, pero no se puede tomar λ_i^P con $i = q_1 + 1, \ldots, q_2$, porque en general no obtendríamos una permutación. Lo que se hace es calcular λ_i^H como λ_k^P, donde k es el menor índice tal que $\lambda_k^P \notin \{\lambda_1^H, \lambda_2^H, \ldots, \lambda_{i-1}^H\}$. Las últimas posiciones se rellenan de nuevo con actividades de la madre, calculándolas de una manera análoga. Hartmann demostró que la permutación resultante era una lista de actividades.

La mutación consiste en recorrer la lista de actividades (obtenida por el operador) e intercambiar dos actividades adyacentes con una cierta probabilidad, si esas actividades no están relacionadas.

Un metaheurístico de las hormigas para el RCPSP

El algoritmo basado en el metaheurístico de las hormigas de Merkle et al., 1999, se puede describir empleando listas de actividades. En el método se realizan un cierto número de generaciones, en cada una de las cuales M hormigas construyen una secuencia cada una. Cada hormiga construye una lista de actividades a la que se le aplica Serie. La selección de la actividad j para la posición i se realiza, de entre las elegibles, de acuerdo con unas probabilidades p_{ij}, basadas en dos tipos de información. La primera fuente de información es heurística, a partir de la regla de prioridad LFT. El segundo tipo de información es histórico, denominado en este contexto información por feromonas: se favorece que se repitan decisiones tomadas en el pasado por buenas soluciones.

En cada iteración, además de evaporarse una cierta cantidad de feromona, la mejor solución obtenida y la mejor solución global añaden feromonas a las posiciones concretas en que se han colocado las actividades. Algunas características adicionales de este algoritmo son: (1) el uso combinado de la evaluación directa (sólo se tiene en cuenta la cantidad de feromona almacenada en ij para el cálculo de p_{ij}) y la de la evaluación sumada (también se tiene en cuenta las cantidades de feromonas en kj, $k < i$) y (2) se 'olvida' la mejor solución global con una cierta probabilidad para que no condicione en exceso la búsqueda. Este algoritmo estándar lo denominaremos Merkle et al. (1). Además del algoritmo estándar, los autores realizan pruebas añadiendo una búsqueda local (2-opt) a la mejor solución en cada iteración o en cada cierto número de iteraciones. Esto produce dos nuevas versiones que denotaremos por Merkle et al. (2) y (3), dependiendo del número de iteraciones y de hormigas que empleen.

En Merkle et al., 2001, se describe una versión mejorada de Merkle et al. (1). En las primeras iteraciones dos colonias de hormigas trabajan de forma independiente. En una colonia las hormigas obtienen las soluciones secuenciando hacia adelante y, en la otra, hacia atrás. Tras un cierto número de iteraciones el proceso continúa con una única colonia, la que ha proporcionado mejores resultados en las últimas iteraciones. Algunos cambios importantes con respecto a Merkle et al. (1) es que se emplea la rdp LST y que después de un número fijo de iteraciones se aplica una búsqueda local (2-opt) a la mejor solución obtenida. En total se evalúan como máximo 5000 secuencias. A este algoritmo lo denominaremos Merkle et al. (4).

El genético de Hartmann (2)

Hartmann mejora su genético en Hartmann, 2000, añadiendo un gen más a cada solución, cuyo valor depende de si la solución se obtiene aplicando Serie o Paralelo.

La solución hija recibe este gen de la madre y el hijo del padre, por lo que a través de la supervivencia de las mejores soluciones y de la combinación se emplea mayoritariamente en cada instancia el método de generación de secuencias que mejor resulta. Esta nueva posibilidad conduce a un cambio en la población inicial, que se modifica para contener también soluciones obtenidas mediante Paralelo, y añade una nueva mutación, la que cambia ese gen a su opuesto con una cierta probabilidad.

Un genético hacia adelante y hacia atrás

Alcaraz y Maroto, 2001, generalizan de otra forma el genético de Hartmann. A cada codificación por listas de actividades le añaden un gen más, que determina si la lista de actividades se secuencia hacia adelante o hacia atrás. Los diferentes componentes del genético se modifican para ajustarse a este cambio. En el artículo se realiza un amplio estudio estadístico para escoger entre diferentes alternativas de mantenimiento de la población, operadores de cruce y operadores de mutación.

1.3. Otros métodos propuestos

Los algoritmos que utilizan principalmente ideas no basadas en rdps ni en metaheurísticos se pueden dividir en: métodos truncados de B&B, métodos basados en arcos disyuntivos, heurísticos basados en la programación entera, algoritmos basados en la secuenciación por bloques y procedimientos basados en la relajación lagrangiana.

Métodos truncados de B&B

Pollack–Johnson, 1995, utiliza un B&B en profundidad mediante un árbol de soluciones parciales. El algoritmo es esencialmente un heurístico de secuenciación en paralelo. En lugar de secuenciar siempre la actividad con mayor prioridad, el algoritmo ramifica en ciertas ocasiones, de manera que en una rama secuencia la actividad con mayor prioridad y en la otra, la de segunda mayor prioridad. Debido al uso de Paralelo el algoritmo obtiene soluciones sin retraso, por lo que la solución óptima podría no alcanzarse.

Sprecher, 1996, emplea su B&B en profundidad como heurístico, imponiendo un tiempo máximo de computación. El proceso de enumeración se guía por el árbol de precedencias que esencialmente ramifica según las actividades del conjunto de elegibles de Serie. A través de la vuelta atrás ('backtracking') se enumeran implícitamente todas las listas de actividades. Se aplican rdps para seleccionar la

actividad más prometedora para ramificar primero, en un intento de obtener buenas soluciones en un periodo corto de tiempo.

Métodos basados en arcos disyuntivos

La idea básica de los métodos basados en arcos disyuntivos es ampliar las relaciones de precedencia (el conjunto de **arcos conjuntivos**) añadiendo arcos adicionales (**arcos disyuntivos**) de manera que los **conjuntos prohibidos minimales**, conjuntos de actividades tecnológicamente independientes que no se pueden secuenciar simultáneamente debido a restricciones de recursos, son destruidos. De este modo se consigue que la secuencia de los inicios más tempranos sea posible respecto a los recursos además de respecto a las relaciones de precedencias. Tres ejemplos de estos métodos son Shaffer et al., 1965, Alvarez-Valdés y Tamarit, 1989b y Bell y Han, 1991.

Heurísticos basados en la programación entera

Oguz y Bala, 1994, utilizan la formulación del RCPSP propuesta por Pritsker et al., 1969, para resolverlo como un problema de programación entera. El horizonte de planificación se divide en T periodos de longitud igual y los tiempos de proceso d_j tienen que ser dados como múltiplos discretos de un periodo. La variable de decisión binaria es $x_{j,t} = 1$ si la actividad j se termina al final del periodo t, 0 en otro caso.

Algoritmos basados en la secuenciación por bloques

Mausser y Lawrence, 1995, utilizan estructuras de bloques para mejorar la duración de los proyectos. Empiezan generando una solución posible mediante Paralelo. Después de esto identifican bloques que representan intervalos de tiempo que contienen completamente todas las actividades procesadas dentro de él. Cada uno de estos bloques se considera de forma independiente con respecto a los demás, y se intenta resecuenciar las actividades que lo forman para así acortar la longitud total del proyecto.

Métodos basados en la relajación lagrangiana

Möhring et al., 2000, diseñan un método basado en la relajación lagrangiana para proporcionar al mismo tiempo cotas inferiores y superiores. Sus resultados son competitivos con los obtenidos con técnicas metaheurísticas. Las soluciones proporcionadas al resolver cada relajación – cada vez que los multiplicadores se actualizan – son secuencias compatibles con las relaciones de precedencias, no así

con las restricciones de recursos. Los autores emplean los tiempos de α-finalización ('α-completion times'), $C_j(\alpha) = s_j + \alpha d_j$, con diferentes α's entre 0 y 1, de estas soluciones como prioridades para obtener secuencias posibles. Para ello emplean un método híbrido entre Serie y Paralelo que calcula, en cada iteración, el menor tiempo de comienzo en que se secuenciaría cada una de entre las k (parámetro entero positivo) elegibles con menor prioridad, y se escoge la de menor tiempo de comienzo.

En este heurístico se emplea el método de relajación lagrangiana considerado en Christofides et al., 1987, basado en la formulación de Pritsker et al., 1969.

Capítulo 2

Dos algoritmos heurísticos: CARA e HIAC

1. INTRODUCCIÓN

Las dos restricciones clave del RCPSP son las relaciones de precedencia y las restricciones de recursos. Nuestro objetivo es construir dos funciones intensificadoras, cada una de ellas basada exclusivamente en uno de esos elementos, y diseñar un algoritmo con cada función como elemento destacado. Estos procedimientos compartirán elementos comunes, tanto técnicas relacionadas con el RCPSP como ideas provenientes de los metaheurísticos. En este capítulo desarrollaremos estos dos algoritmos, CARA e HIAC, y los comparemos con algunos de los mejores heurísticos de la literatura (apartado 4). Los apartados 5 y 6 servirán para profundizar en algunos aspectos más generales surgidos a raíz de los puntos anteriores.

Antes de proceder a la explicación de CARA e HIAC vamos a presentar las dos codificaciones duales que van a emplear todos los procedimientos desarrollados a lo largo de la tesis, las listas de actividades y los órdenes topológicos. Veamos cada una de ellas.

La representación por listas de actividades

Una lista de actividades (cf. apartado 6.1 capítulo 1) es una permutación de n actividades $\lambda = (j_1\ j_2\ \dots\ j_n)$ compatible con las relaciones de precedencia. Si $i = j_p$ diremos que la actividad i está en la posición p, o que el orden de i en λ (**orden**(i,λ)) es p.

Dada una secuencia S, una lista de actividades λ que satisfaga $s_i < s_j \rightarrow$ orden$(i,\lambda) <$ orden(j,λ) se denomina **representación por lista de actividades (representación AL) de S**. Al ser una lista de actividades se puede emplear Serie para obtener una secuencia activa $S(\lambda)$ a partir de λ. Cada secuencia S puede ser representada por varias listas de actividades de este tipo pero, a diferencia de lo que ocurre en el caso de las listas habituales, la redundancia está completamente controlada, ya que todas las representaciones AL de S sólo se diferencian en la posición asignada a las actividades que comienzan al unísono. Además, se puede escoger una representación única si se determina una forma de desempatar, por ejemplo, por menor número de actividad. Obviamente, si S es activa y $\lambda(S)$ es una representación por lista de actividades de S, se cumple $S(\lambda(S)) = S$.

Una lista de actividades no es necesariamente una representación AL de una secuencia activa (ver ejemplo posterior). Sin embargo, si λ' es una lista de actividades,

entonces $\lambda = \lambda(S(\lambda'))$ es una representación AL de la secuencia $S(\lambda) = S(\lambda')$. A esta transformación $\lambda' \rightarrow \lambda$ la denominaremos **redefinir** λ'. Cuando se redefine se desempata según λ', escogiendo primero las actividades con menor orden. De este modo se modifica λ' exclusivamente lo necesario para que las actividades estén en orden creciente de inicios.

A lo largo de la tesis supondremos que, salvo cuando estemos redefiniendo una lista de actividades, la regla de desempate es escoger primero la actividad de menor número, y denotaremos por $\lambda(S)$ a la única representación AL de S. Dado que trabajaremos con secuencias activas emplearemos S y $\lambda(S)$ indistintamente.

La representación por órdenes topológicos

Un **orden topológico** de las actividades de un proyecto es un orden compatible con las relaciones de precedencia, i.e., un vector $\gamma = (\gamma_1, \gamma_2, ..., \gamma_n) = (\gamma(1), \gamma(2), ..., \gamma(n))$ de enteros distintos entre 1 y n que satisface que si $(i,j) \in A$ entonces $\gamma_i < \gamma_j$. Los órdenes topológicos son un caso especial de los vectores de prioridades (cf. apartado 6.2 capítulo 1), y por tanto se puede emplear Serie para su decodificación. Asumiremos que una actividad es más importante cuanto menor sea su prioridad.

La codificación mediante órdenes topológicos es una codificación dual de las listas de actividades porque para cada lista de actividades λ codificación de una secuencia S, existe un único orden topológico γ asociado a λ, el dado por $\gamma_i = k$ sii $j_k = i$. Además se tiene $S(\gamma) = S(\lambda) = S$. El recíproco también es cierto.

Dada una secuencia S, denominaremos **representación por ordenes topológicos (representación OT) de S** a un orden topológico que cumpla $s_i < s_j \rightarrow \gamma_i < \gamma_j$. También emplearemos el término **vector OT**. Una secuencia puede ser representada por varias representaciones OT, pero la redundancia está completamente controlada, ya que las representaciones OT de S sólo se diferencian en la prioridad asignada a las actividades que comienzan al unísono. Además, se puede escoger una representación única si se determina una forma de desempatar, por ejemplo, por menor número de actividad. Obviamente, si S es activa y $\gamma(S)$ es una representación por orden topológico de S, se cumple $S(\gamma(S)) = S$.

Un orden topológico no es necesariamente una representación OT de una secuencia activa (ver ejemplo posterior). Sin embargo, si γ' es un orden topológico, entonces $\gamma = \gamma(S(\gamma'))$ es una representación OT de la secuencia $S(\gamma) = S(\gamma')$. A esta transformación $\gamma' \rightarrow \gamma$ la denominaremos **redefinir** γ'. Cuando se redefine se desempata según γ', eligiendo primero las actividades con menor prioridad.

Redefinir un orden topológico o una lista de actividades no es computacionalmente costoso pero requiere un esfuerzo que, en ocasiones, puede no ser necesario. En algunos procedimientos que describiremos más adelante y que, en principio, devuelven la redefinición de un orden topológico o lista de actividades calculado, evitaremos redefinir en el caso en que las soluciones obtenidas no vayan a ser empleadas más por su baja calidad.

Ejemplo

Consideremos el problema dado por la Figura 2.1. La secuencia S de la Figura 2.2 es posible para este problema.

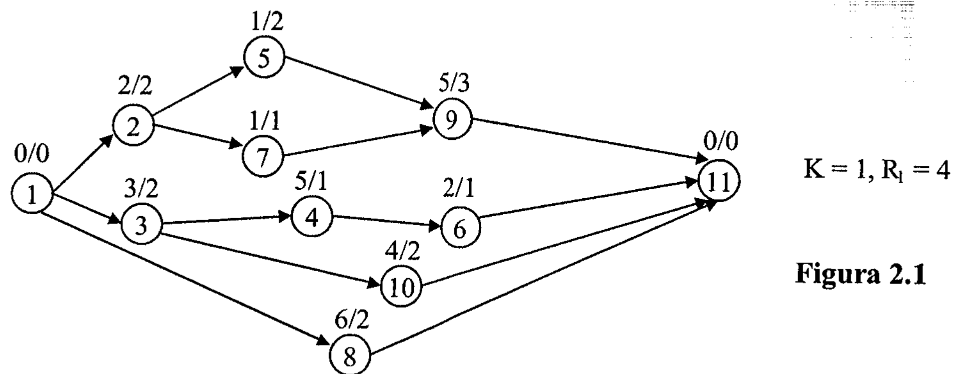


Figura 2.1

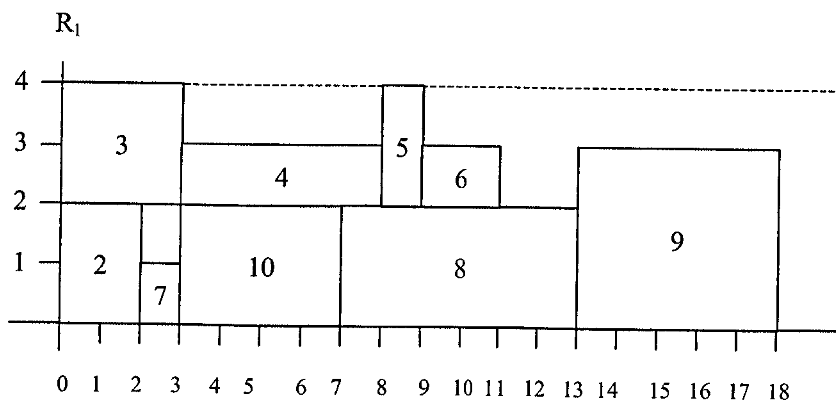


Figura 2.2

El vector λ = (1 2 3 7 4 10 8 5 6 9 11) es una representación AL de S, mientras que el vector λ' = (1 2 3 7 4 10 8 5 9 6 11) que también lleva a S no lo es, puesto que no respeta los tiempos de comienzo ($s_6 < s_9$ y 9 está colocada en λ' antes que 6). De hecho, no es una representación de ninguna secuencia. Sin embargo, si redefinimos λ' obtenemos λ. Las otras representaciones AL son (1 2 3 7 10 4 8 5 6 9 11), (1 3 2 7 4

10 8 5 6 9 11) y (1 3 2 7 10 4 8 5 6 9 11). El vector γ = (1,2,3,6,8,9,4,7,10,5,11) es una representación OT de la secuencia S. Las otras representaciones alternativas son (1,3,2,6,8,9,4,7,10,5,11), (1,2,3,5,8,9,4,7,10,6, 11) y (1,3,2,5,8,9,4,7,10,6,11). El vector γ' = (1,2,3,6,8,10,4,7,9,5,11), que también lleva a S, no es una representación OT de S, dado que no es compatible con los tiempos de inicio ($s_6 < s_9$ y $\gamma_6' > \gamma_9'$). Si redefinimos γ', obtenemos γ; en particular, γ' no es representación de ninguna secuencia. Las representaciones λ y γ (y también las codificaciones λ' y γ') son duales porque $\gamma_i = k$ sii $j_K = i$.

2. El algoritmo CARA

El algoritmo CARA ('Critical Activity Reordering Algorithm') es una implementación no estándar de principios de la búsqueda tabú basada en la representación por órdenes topológicos. Dada γ, una representación OT de una secuencia S, distinguimos dos partes en S: la cabeza, formada por aproximadamente la mitad de las actividades con menor orden y la cola, formada por el resto de actividades. El procedimiento, que puede ser visto como una aproximación por oscilación estratégica ('stategic oscillation', cf. Glover y Laguna, 1997), trata de mejorar alternativamente la cola y la cabeza, manteniendo la otra parte fija.

El algoritmo consta de dos fases, la primera de las cuales comienza con una población inicial de vectores OT. En esta fase se emplean dos movimientos definidos probabilística y fundamentalmente como función de la información ofrecida por la holgura relativa de las actividades. Para mejorar la cola (cabeza) de una representación OT se aplica una búsqueda en forma de abanico ('fan search', Glover y Laguna, 1997) sobre el conjunto que asigna el mismo orden a las actividades de la cabeza (cola).

La segunda fase comienza con la mejor solución obtenida en la primera fase, a partir de la cual se genera una nueva población mediante un método de muestreo sesgado modificado. A continuación se procede a intentar mejorar cada solución, con la misma estrategia de mejorar alternativamente la cabeza y cola. En esta fase se define el movimiento a ejecutar con un muestreo aleatorio de nuevo cuño.

Todos los movimientos de CARA se caracterizan por un uso estratégico de las probabilidades. Estas probabilidades se calculan para favorecer decisiones atractivas a priori, sin excluir completamente en general otras opciones. Esto induce variedad, compensa la imperfección de la decisión más prometedora pero miope ('greedy') y

hace innecesario emplear explícitamente estructuras de memoria como las listas tabú para evitar la repetición de soluciones.

2.1. Definiciones preliminares

Dada una secuencia S, $\mathbf{LF_i}$ denota el tiempo de finalización máximo de la actividad i, determinado por recursión hacia atrás, omitiendo las restricciones de recursos y empleando T(S) como cota superior de la duración del proyecto. Se tiene $LF_i = T(S) - CPM(TSuc_i)$, donde $CPM(S_i)$ simboliza la longitud del CPM (cf. apartado 2 capítulo 1) en la subred formada por (todos) los sucesores de i. Es claro que $T(S) \geq f_i + CPM(TSuc_i)\ \forall i$. Además, $LF_n = f_n = s_n = T(S)$, y como siempre existe al menos una predecesora directa de n, i, que finaliza en T(S), se cumple que siempre existe una actividad i, $i \neq n$ con $LF_i = f_i = T(S)$.

La **holgura total de una actividad i relativa a S** ('total float') se puede definir como $TF(i,S) = LF_i - f_i \geq 0$. Una actividad i, $i \neq 1$, n, se dice **crítica relativa a S** cuando $TF(i,S) = 0$. En ese caso $T(S) = f_i + CPM(TSuc_i)$. Definiendo **Cr**(S) como el conjunto, siempre no vacío, de las actividades críticas relativas a S, cualquier secuencia S' de menor longitud que S debe cumplir la condición (*) $s_i' < s_i\ \forall i \in Cr(S)$. (Demostración: $\exists\, i \in Cr(S) / s_i' \geq s_i \Rightarrow f_i' \geq f_i \Rightarrow T(S') \geq f_i' + CPM(TSuc_i) \geq f_i + CPM(TSuc_i) = T(S)$ porque $i \in Cr(S)$).

Para mejorar una secuencia dada S, el algoritmo CARA trata de adelantar los comienzos de todas las actividades de Cr(S). Dada una representación OT, γ, de S, el método consiste en disminuir el orden γ_i asociado a cada una de esas actividades con el objetivo de obtener un nuevo vector OT γ' cuya $S(\gamma')$ satisfaga la condición (*). Para que esta disminución tenga el efecto deseado de reducir el inicio de las actividades críticas puede ser necesario adelantar una o varias de sus predecesoras no críticas. Veamos un ejemplo de esto último y de algunas de las definiciones dadas.

Ejemplo

La Tabla 1 muestra los valores de LF_j, f_j y TF(j,S) para cada actividad de la secuencia S de la Figura 2.2, codificada por el vector OT $\gamma = (1,2,3,6,8,9,4,7,10,5,11)$. Como se puede observar Cr(S) = {9}. Dado que la actividad 5 es un predecesor directo de 9 y $\gamma_5 = 8$, lo máximo que podemos adelantar la actividad 9 sin modificar la prioridad de 5 es asignando el nuevo orden $\gamma_9' = 9$, obteniendo el vector $\gamma' = (1,2,3,6,8,10,4,7,9,5,11)$. Pero esto no acorta la duración, puesto que $S(\gamma') = S$. Sin embargo, si además

adelantamos 5 lo más posible y obtenemos $\gamma'' = (1,2,4,8,3,10,5,9,6,7,11)$, el resultado es $S(\gamma'')$ (Figura 2.3), 3 unidades menor que S.

Actividad j	1	2	3	4	5	6	7	8	9	10	11
LF_j	8	12	11	16	13	18	13	18	18	18	18
f_j	0	2	3	8	9	11	3	13	18	7	18
TF(j,S)	8	10	8	8	4	7	10	5	0	11	0

Tabla 2.1

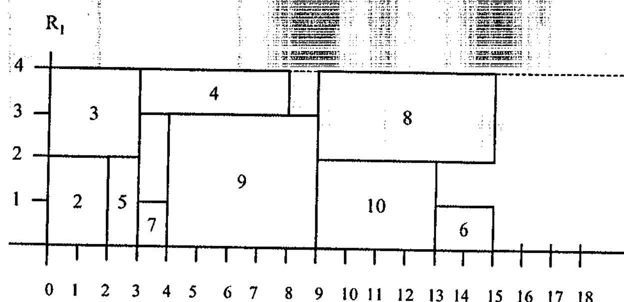


Figura 2.3

Existen otros motivos por los que adelantar actividades no críticas predecesoras de otras críticas. Si el LF fuera calculado con una cota inferior más potente que la del CPM – en vez de $CPM(TSuc_i)$ calculáramos $LB(TSuc_i)$ con otra cota inferior mejor – (cf. Klein, 2000), entonces posiblemente algunas actividades con holgura total positiva se volverían críticas. Supongamos que j es una actividad crítica para una cierta cota inferior LB y denominamos $T(TSuc_j)$ a la longitud mínima del proyecto formado por $TSuc_j$. Ahora bien, $0 = LF_j^{LB} - f_j = T(S) - LB(TSuc_j) - f_j \geq T(S) - T(TSuc_j) - f_j \geq T(S) - (\max\{f_h / h \in TSuc_j \backslash \{n\}\} - \min\{s_h / h \in TSuc_j\}) - f_j \geq T(S) - (\max\{f_h / h \in TSuc_j \backslash \{n\}\} - f_j) - f_j = T(S) - \max\{f_h / h \in TSuc_j \backslash \{n\}\}$, lo cual implica que $\max\{f_h / h \in TSuc_j \backslash \{n\}\} = T(S)$, por lo que j tiene una sucesora distinta de n que acaba en T(S).

Así pues, cada actividad crítica para cualquier cota inferior LB igual o mejor que el CPM es una actividad crítica para el CPM o posee una sucesora crítica para el CPM (i.e., un elemento de Cr(S)), por lo que si adelantamos el suficiente número de actividades predecesoras de las actividades del conjunto Cr(S) podemos asegurar que adelantamos todas las actividades críticas para cualquier LB.

Con respecto a las actividades no relacionadas con las críticas, su holgura, definida respecto a la secuencia S, es mucho más útil si los movimientos que transforman S no cambian su estructura en profundidad. Una estrategia para limitar el alcance de estos cambios consiste en mantener fijo el orden de aproximadamente la primera mitad de

las actividades (la cabeza) y únicamente reordenar la otra mitad (la cola). Esta estrategia aumenta el efecto intensificador al limitar la región de búsqueda. Todas estas consideraciones conducen a la siguiente definición.

Definición: El conjunto Advance(S)

> **Advance**(S) = $\{j \in Cr(S,10) \ / \ \gamma_j > n/2\} \cup \{j \in Cr(S,3) \ / \ \gamma_j \leq n/2\}$, donde **Cr**(S,δ) = $Cr(S) \cup \{j \ / \ 0 < TF(j,S) \leq \delta \text{ y } \exists \ i \in TSuc_j \cap Cr(S)\}$, $\delta \geq 0$.

Para mejorar una secuencia, CARA intenta adelantar el inicio de todas las actividades de Advance(S). Para conseguirlo, se establece una frontera cercana a n/2, **mino**, sobre el orden de las actividades en γ. Esto divide las actividades en la cabeza y la cola de tal modo que la parte de la secuencia que cambia (la cola) siempre contiene el conjunto Advance(S). La cabeza (cola) está formada por las actividades con menor (mayor o igual) orden que mino, las que se secuencian antes (después) cuando se aplica Serie:

Definición: Cabeza y cola de una secuencia

> **Head**(S) = Head(γ) = $\{j \ / \ \gamma_j < mino\}$, **Tail**(S) = Tail(γ) = $\{j \ / \ \gamma_j \geq mino\}$, donde **mino** = $\min\{\min\{\gamma_j \ / \ j \in Advance(S)\} - 3\}, n/2\}$.

Notar que los dos valores de δ (3 y 10) empleados en la definición de Advance(S) reflejan la intención de tratar de forma diferente la primera y segunda parte de la secuencia.

Volviendo a la secuencia de la Figura 2.2 y para la OT γ = (1,2,3,6,8,9,4,7,10,5,11), Cr(S) = {9}, Cr(S,10) = {9} ∪ {2,5,7} = {2,5,7,9}, Cr(S,3) = {9} ∪ ∅ = {9}, Advance(S) = {5,9}, mino = min{min{8,10} - 3, 11/2} = 5, Head(S) = Head(γ) = {1,2,3,7} y Tail(S) = Tail(γ) = {4,5,6,8,9,10,11}.

2.2. Fase 1 de CARA

Definición de los movimientos

Sea γ una representación OT de una secuencia activa S. El objetivo de los dos tipos de movimientos descritos a continuación es generar un orden topológico tal que:

(1) $\gamma_j' < \gamma_j$, $\forall j \in$ Advance(S).

(2) $\gamma_j' = \gamma_j$, $\forall j \in$ Head(S).

(3) $\forall i, j \in$ Tail(S)\Advance(S): $\gamma_i < \gamma_j \rightarrow \gamma_i' < \gamma_j'$.

Las condiciones (1) y (2) se han comentado ya, mientras que la condición (3) exige que se mantenga el orden relativo de aquellas actividades que no se desean adelantar específicamente. Esto ayuda a mantener la estructura de S y a que la información dada por la holgura no pierda su significado, y, además, refuerza el efecto intensificador de la búsqueda. La condición (1) es un objetivo general que no siempre va a ser posible satisfacer.

El primer tipo de movimiento, LF_BACKWARD_MOVE(γ), ordena iterativamente las actividades de la cola desde el final al principio, asignando primero el orden n, después el n-1 y así sucesivamente. En cada iteración se escoge una actividad j para asignarle el orden k. La elección de la actividad, dividida en tres etapas, está guiada principalmente por la regla de prioridad LF (cf. apartado 8.1 capítulo 1), pero no exclusivamente. En la primera, el conjunto *Candidate* – las actividades de Tail(γ) que no han sido reordenadas pero cuyas sucesoras sí lo han sido – se divide en tres subconjuntos, dependiendo de si pertenecen al conjunto Advance(S) o no y, dentro del primero, si nos interesa asignarle orden o no en esta iteración. En la segunda etapa se escoge uno de esos subconjutos probabilísticamente, teniendo en cuenta los valores que la regla LF asigna a sus elementos. En la última etapa, j se selecciona dentro del conjunto elegido. LF se emplea de nuevo para elegir esa actividad si ésta pertenece a Advance(S) y nos interesa asignarle orden en esa iteración.

Este proceso, y la forma en que se construyen las probabilidades, facilita el doble objetivo de utilizar la información ofrecida por LF y de satisfacer las condiciones (1), (2) y (3). Es interesante remarcar que la condición (1) se cumplirá sii al finalizar el procedimiento no se ha escogido en ninguna iteración C_NE.

Figura 2.4. LF_BACKWARD_MOVE(γ)

Paso 0. Inicialización.

Calcular Advance(S), mino, Head(S) y Tail(S).

$\gamma_j' = \gamma_j \ \forall j \in$ Head(γ); $\gamma_j' = \infty \ \forall j \in$ Tail(γ); $\gamma_n' = n$; $k = n - 1$.

Paso 1. Partición del conjunto de candidatas.

Definir

Candidate = $\{j / j \in$ Tail(γ); $\gamma_j' = \infty$ y $\gamma_i' < \infty \ \forall i \in S_j\}$,

Eligible = $\{j / \gamma_j > k\}$.

Dividir Candidate en

C_NA = Candidate \ Advance(S).

C_E = Candidate $\cap$ Advance(S) $\cap$ Eligible.

C_NE = Candidate\ (C_NA $\cup$ C_E).

Paso 2. Selección probabilística del subconjunto.

Si C_NA $\cup$ C_E = $\varnothing$, entonces SELECT = C_NE. Ir al Paso 3.

Si no, calcular:

$\alpha = \max\{LF_i / i \in C_E\}$ (Si C_E = $\varnothing$, entonces $\alpha = 0$).

$\beta = \max\{LF_i / i \in C_NA\}$ (Si C_NA = $\varnothing$, entonces $\alpha = 0$).

Seleccionar r de (0,1) aleatoriamente.

Si $r \leq \alpha/(\alpha+\beta)$, SELECT = C_E. En otro caso, SELECT = C_NA.

Paso 3. Selección probabilística de la actividad.

Si SELECT = C_NE, j es la actividad con mayor orden en C_NE.

Si SELECT = C_NA, j es la actividad con mayor orden en C_NA.

Si SELECT = C_E, j es escogida probabilísticamente de la siguiente manera: un 75% de probabilidad de que sea la actividad de mayor LF de C_E y un 25% de probabilidad de ser una de las demás actividades en C_E, escogida aleatoriamente.

Paso 4.

Realizar $\gamma_j' = k$; $k = k-1$.

Si $k <$ mino, devolver γ'', la redefinición del orden topológico γ'. En otro caso ir al Paso 1.

El segundo tipo de movimiento, DSLACK_FORWARD_MOVE(γ), primero construye una secuencia S' para después redefinir el orden topológico γ' que ha llevado a S' y devolver la representación OT resultante. S' se construye completando iterativamente la secuencia parcial formada por las actividades de la cabeza. Para seleccionar la siguiente actividad j a secuenciar se implementa un procedimiento de tres pasos similar al del anterior movimiento, y que comparte los mismos objetivos. En esta ocasión, sin embargo, la holgura dinámica ('DSLACK'), que por definición se actualiza en cada iteración de acuerdo con el estado actual de la secuencia parcial, establece parcialmente la prioridad por la que se seleccionan los conjuntos en la etapa 2.

DSLACK es empleado además para escoger la actividad en la etapa 3 si ésta debe ser del conjunto Advance(S). Notar que si en algún momento se cumple $\gamma(i_0) < k+1$ no se podrá cumplir la condición (1).

Figura 2.5. DSLACK_FORWARD_MOVE(γ)

Paso 0. Inicialización.

Calcular Advance(S), mino, Head(S) y Tail(S).

Secuenciar en serie las actividades Head(S) en orden creciente de γ.

Label(j) = 1, $\gamma'(j) = \gamma(j)$ $\forall j \in$ Head(S); Label(j) = 0 $\forall j \in$ Tail(S); k = mino.

Paso 1. Partición del conjunto de candidatas.

Definir

Candidate = {j / j $\in$ Tail(γ); Label(j) = 0 y Label(j) = 1 $\forall i \in P_j$},

Dividir Candidate en

C_A = Candidate $\cap$ Advance(S).

C_NA = Candidate\ Advance(S).

Paso 2. Selección probabilística del subconjunto.

Si C_A = $\varnothing$ (C_NA = $\varnothing$), entonces SELECT = C_NA (C_A). Ir al Paso 3.

Sea $i_0 \in$ C_A tal que $\gamma_{i_0} = \min\{\gamma_i / i \in C_A\}$.

Si $\gamma_{i_0} \le k+1$, entonces j = i_0. Ir al Paso 4.

Si no, calcular:

$\alpha = \min\{DSLACK_i / i \in C_A\}$.

$\beta = \min\{DSLACK_i / i \in C_NA\}$.

2.1. Si $\alpha < 0$ y $\beta \ge 0$, SELECT = C_A.

2.2. Si $\alpha \ge 0$ y $\beta < 0$, SELECT = C_NA.

2.3. En otro caso, calcular r de (0,1) aleatoriamente.

2.4. Si $\alpha < 0$ y $\beta < 0$:

2.4.1 Si $r \ge \alpha/(\alpha+\beta)$, SELECT = C_A. En otro caso, SELECT = C_NA.

2.5. Si $\alpha < 0$ y $\beta < 0$:

2.5.1 Si $r \le \alpha/(\alpha+\beta)$, SELECT = C_A. En otro caso, SELECT = C_NA.

2.6. Si $\alpha = 0$ y $\beta = 0$: Seleccionar aleatoriamente entre C_A y C_NA.

Paso 3. Selección probabilística de la actividad.

Si SELECT = C_A, j es la actividad con menor holgura dinámica en C_A.

Si SELECT = C_NA, j es la actividad con menor orden en C_NA.

Paso 4. Secuenciación de la actividad.

Secuenciar j tan pronto como sea posible teniendo en cuenta las relaciones de precedencia y los recursos empleados. Label(j) = 1, $\gamma'(j)$ = k y k = k+1.

Paso 5.

Si j = n, S' es la secuencia construida. Devolver γ'' la redefinición de γ'. Stop.

En otro caso, ir al Paso 1.

Procedimiento de mejora de la cola

La Figura 2.6 describe el procedimiento para mejorar la cola de un vector OT γ. Comienza generando (PopSize + nbs) vecinos de γ y seleccionando los mejores PopSize para formar la población POP. Los vecinos de γ son generados aplicando X_MOVE(γ), donde esta función representa LF_BACKWARD_MOVE la mitad de las veces y DSLACK_FORWARD_MOVE la otra mitad. Después de esto, se aplica una búsqueda en forma de abanico (FAN_SEARCH) a cada vector OT de POP. Si este conjunto de búsquedas encuentra una solución mejorada el procedimiento total comienza de nuevo a partir de ella.

Figura 2.6. Esquema de TAIL_IMPROVING(γ)

1. POP = $\varnothing$.
2. Desde i = 1 hasta PopSize + nbs, hacer
   2.1. σ_i = X_MOVE(γ).
   2.2. POP = POP $\cup$ $\{\sigma_i\}$.
3. Seleccionar los PopSize mejores vectores OT de POP: $\sigma_1, ..., \sigma_{PopSize}$.
4. Desde i = 1 hasta PopSize, hacer
   4.1. σ = FAN_SEARCH(σ_i).
   4.2. Si (T(σ) < T(γ)), $\gamma = \sigma$. Ir a 1.
5. Devolver γ.

FAN_SEARCH(γ) es un procedimiento iterativo que en la primera iteración genera *width* vecinos de γ a partir de la aplicación de X_MOVE(γ). En las iteraciones siguientes genera width vecinos de la mejor solución obtenida en la iteración anterior. FAN_SEARCH(γ) acaba después de *depth* iteraciones sin mejoras.

Los parámetros se tomaron PopSize = 4, nbs = 20, width = 4 y depth = 4.

Procedimiento de mejora de la cabeza

TAIL_IMPROVING(γ) es un procedimiento que realiza una búsqueda en el subespacio formado por los órdenes topológicos que comparten la cabeza con γ y busca la 'mejor' cola que puede ser añadida a esa cabeza. Una vez esta búsqueda se agota, parece natural cambiar la estrategia de búsqueda en el sentido de fijar la cola para intentar encontrar la 'mejor' cabeza.

Antes de explicar el procedimiento HEAD_IMPROVING(γ) es necesario recordar unas definiciones comentadas en el apartado 5 del capítulo 1.

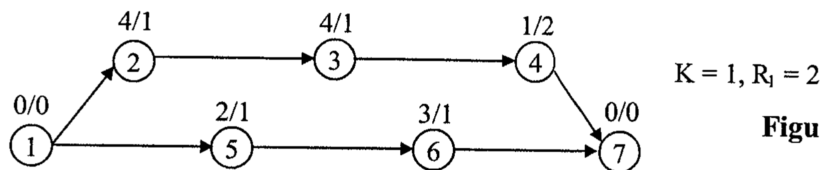
Definición: Justificar una actividad y una secuencia

Dada una secuencia S, **justificar una actividad i ≠ n (1) a la derecha (izquierda)** consiste en obtener la única secuencia posible S' donde $s_j' = s_j \ \forall j \neq i$ y s_i es lo más grande (pequeño) posible.

Justificar una secuencia S a la derecha (izquierda) consiste en obtener una secuencia S activa a la derecha (izquierda) S^R (S^L) aplicando sucesivas justificaciones a las actividades en orden decreciente (creciente) de f_i (s_i).

Estas secuencias no son únicas, dependen de los desempates entre las actividades que finalizan (comienzan) a la vez. Si $s^R_1 > 0$ ($s^L_n < T$), entonces S^R (S^L) es de menor longitud que S. Es fácil demostrar que $T(S^R) \leq T(S)$ ($T(S^L) \leq T(S)$).

Ejemplo



K = 1, R_1 = 2

Figura 2.7

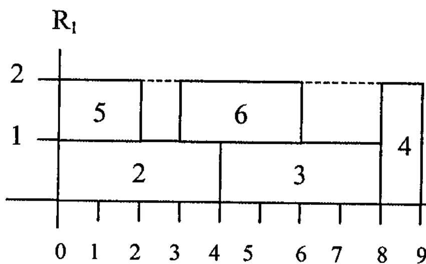


Figura 2.8

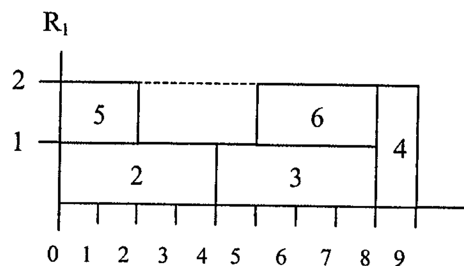


Figura 2.9

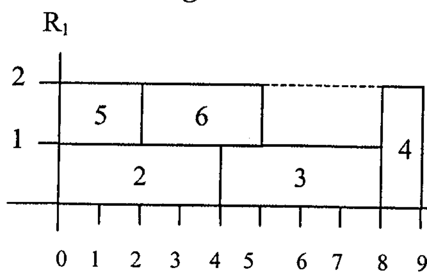


Figura 2.10

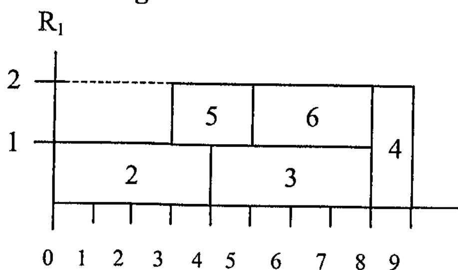


Figura 2.11

La secuencia S de la Figura 2.8 es posible en el problema de la Figura 2.7. Si justificamos a la derecha la actividad 6 obtenemos la secuencia de la Figura 2.9. La actividad 6 no puede secuenciarse más tarde debido a las restricciones de recursos. Si la justificamos a la izquierda llegamos a la solución de la Figura 2.10; no se puede ejecutar antes por recursos ni por relaciones de precedencia (5 → 6). Si justificamos S a la derecha obtenemos la secuencia S' de la Figura 2.11. Las actividades se han

justificado en orden decreciente según sus finales, primero 6 y luego 5 (2, 3 y 4 están fijas).

Se ha comentado anteriormente que el proyecto obtenido al invertir las relaciones de precedencia se denomina inverso del proyecto original. Dada una secuencia S, la transformación s^r_j = T(S) – f_j proporciona una secuencia S^r del proyecto inverso. Obviamente, si S es una secuencia de un proyecto (inverso), entonces $(S^R)^r$ ($(S^L)^r$) es una secuencia activa para el proyecto inverso (original). La cola (cabeza) de S se transforma aproximadamente en la cabeza (cola) de $(S^R)^r$. Esto permite construir una nueva función HEAD_IMPROVING (Figura 2.12), haciendo uso de TAIL_IMPROVING*, que no es más que TAIL_IMPROVING trabajando con la red inversa. RIGHT_JUSTIFICATION (LEFT_JUSTIFICATION) justifica las actividades a la derecha (izquierda) en orden decreciente de f_j (creciente de sj); los desempates se rompen aleatoriamente.

Figura 2.12. Esquema de HEAD_IMPROVING(γ)

1. $\gamma_{best} = \gamma$.
2. σ = RIGHT_JUSTIFICATION(γ).
3. φ = TAIL_IMPROVING*(σ^r).
4. θ = LEFT_JUSTIFICATION(φ).
5. Si $(T(\theta) < T(\gamma_{best}))$ $\gamma_{best} = \theta^r$.
6. Devolver γ_{best}.

Oscilación Estratégica

El mecanismo oscilatorio descrito en la Figura 2.13 produce una interacción efectiva entre la intensificación y la diversificación. La búsqueda se lleva alternativamente a dos regiones diferentes, la región con una cabeza fija y aquella con la cola fija, mientras que la búsqueda en abanico es un medio de intensificar la búsqueda en las dos regiones.

Figura 2.13. Esquema de HEAD_TAIL_IMPROVING(γ)

1. σ = TAIL_IMPROVING(γ).
2. φ = HEAD_IMPROVING(σ).
3. Si $(T(\varphi) < T(\sigma))$ σ = TAIL_IMPROVING(φ). En otro caso, devolver σ.
4. Si $(T(\sigma) < T(\varphi))$ ir a 2. En otro caso, devolver φ.

Población inicial

La población inicial de soluciones se genera con el objetivo de introducir calidad y diversidad. El procedimiento INITIAL_SET_1(N1) genera 10 órdenes topológicos aleatorios. Dos secuencias se obtienen a partir de cada uno, aplicando Serie y Paralelo. También se emplean nueve conocidas reglas de prioridad, MTS, LST, LFT, CRPW, WRUP (w_1 = 0.3 y w_2 = 0.7), MSLK, WCS, RSM e IRSM; cada una de ellas genera una o dos secuencias dependiendo de si se puede emplear con un SGS o con ambos (cf. capítulo 1). Para cada secuencia se obtiene una representación OT con evaluación T(S), y la población inicial se crea con las mejores N1 de estas soluciones – representaciones. En la Figura 2.14 se puede observar el esquema de la Fase 1.

Figura 2.14. Esquema de la Fase 1(N1)

1. POP = INITIAL_SET_1(N1).
2. HEAD_TAIL_IMPROVING(POP).
3. Devolver la mejor solución obtenida.

2.3. Fase 2 de CARA

La Fase 2 es similar a la Fase 1, diferenciándose en la generación de la población inicial, el tipo de movimiento y el procedimiento de mejora de la cola.

La Fase 1 trabaja con una población inicial de soluciones de calidad relativamente baja y lleva la búsqueda a una especie de óptimo local, γ, presumiblemente un vector OT de calidad alta. Como la Fase 2 comienza la búsqueda dentro de una región de alta calidad, parece apropiado realizar movimientos controlados para evitar acabar en regiones de mucha menor calidad. Los métodos de muestreo multipaso pueden ser adaptados adecuadamente para definir estos movimientos.

El método de muestreo aleatorio sesgado **β-Biased** ($0 \leq \beta \leq 1$) consiste en, dado un vector OT γ, obtener un orden topológico γ' empleando Serie y seleccionando en cada iteración una actividad elegible j de la siguiente manera. Se genera un número p aleatoriamente en (0,1). Si $p < \beta$, j es la actividad con menor prioridad. En otro caso, j es una de las otras actividades elegibles, escogida por muestreo aleatorio sesgado. Por actividad elegible entendemos que sus predecesoras han sido secuenciadas. Al finalizar el proceso se redefine γ' para que la salida sea una representación OT.

El parámetro β tiene la siguiente interpretación. En media, $n\beta$ es el número de iteraciones en las que γ y γ' escogerían la misma actividad. Por lo tanto, si definimos $\beta = 1 - k/n$, con k entero, entonces k es, en media, el número de iteraciones en las que la decisión según γ y γ' sería diferente. Por tanto, el parámetro β (o k) controla lo diferente que un orden topológico generado se diferencia del vector OT original. El procedimiento INITIAL_SET_2(γ_0,N2) genera 400 vectores OT con $\beta = 1\text{-}20/n$ y 200 con $\beta = 1\text{-}10/n$ aplicando β-Biased a γ_0 y, después, selecciona los mejores N2 para la población inicial POP de la Fase 2.

En β-Biased, la selección de las actividades para secuenciar – una vez $p \geq \beta$ – se realiza conforme a uno de los mejores métodos de muestreo de la literatura. Esto quiere decir que los movimientos que se realizan sobre la solución original están avalados por la efectividad de esos métodos. Sin embargo, β-Biased se puede aplicar sobre cualquier solución, algo imposible en los muestreos usuales, que se utilizan siempre con la(s) misma(s) regla(s) de prioridad. Esto limita casi exclusivamente su uso en heurísticos 'avanzados' a la obtención de soluciones iniciales. Esta restricción no existe en β-Biased, ya que es capaz de obtener soluciones diferentes de una dada, pero de una calidad similar. Esto nos permitirá emplearlo en algoritmos muy diversos a lo largo de la tesis.

Definición de movimiento

El procedimiento WINDOW_SAMPLING_MOVE(γ) construye una secuencia S' para después redefinir el orden topológico γ' que ha dado lugar a S' y devolver la representación OT correspondiente. S' se construye completando la secuencia parcial formada por las actividades de la cabeza de S empleando Serie y un nuevo método de muestreo aleatorio para seleccionar la siguiente actividad a ser secuenciada. En cada iteración se consideran elegibles aquellas actividades con predecesores secuenciados y cuyos órdenes difieran del mínimo $w_0 = \min(\gamma_i \,/\, i \in \text{Candidate})$ como máximo una cierta cantidad prefijada *window*. El procedimiento escoge la actividad a ser secuenciada aleatoriamente de entre las elegibles.

La cercanía de γ' a γ puede ser controlada seleccionando apropiadamente este parámetro, aunque su efecto en la proximidad de S' a S depende de la secuencia en un primer término, y en general de la instancia en la que se esté trabajando. Un valor de window producirá menos cambios en general en una secuencia donde muchas actividades comiencen a la vez que en una donde este fenómeno sea menor. Una estrategia robusta es emplear dos valores, uno pequeño y otro mayor.

El parámetro window restringe por medio de γ las actividades que se pueden escoger en cada iteración. Es interesante resaltar que en β-Biased ($0 < \beta < 1$) se pueden

obtener todos los órdenes topológicos, dado que en cada iteración todas las actividades con predecesores escogidos tienen una probabilidad mayor de 0 de ser elegidas. Sin embargo, en WINDOW_SAMPLING_MOVE en general esto no es así, porque se descartan las actividades cuyos órdenes difieran de w_0 en más del parámetro window. Como en general window será pequeño, el subconjunto de soluciones accesibles mediante la aplicación de este procedimiento es mucho menor que en β-Biased. Para contrarrestar este hecho, la elección de la actividad a secuenciar se lleva a cabo aleatoriamente, en lugar de la forma que emplea β-Biased, muy guiada por γ. Es interesante remarcar que WINDOW_SAMPLING_MOVE(γ) podría escoger la actividad de otra manera, por ejemplo, asignando una prioridad a cada actividad elegible (mediante LF, holgura dinámica, etc.) y seleccionando la actividad probabilísticamente de acuerdo con esas prioridades.

La función WINDOW_SAMPLING_MOVE(γ) tiene algunas similitudes con una técnica de Möhring et. al, 2000, diseñada de forma independiente y comentada en el apartado 8.3 del capítulo 1.

Figura 2.15. WINDOW_SAMPLING_MOVE(γ)

Paso 0. Inicialización.

Aplicar Serie a las actividades de $Head(S) = \{j \,/\, \gamma_j < \lfloor n/2 \rfloor\}$ en orden creciente de γ. $Label(j) = 1, \gamma'_j = \gamma_j \; \forall j \in Head(S); Label(j) = 0 \; \forall j \in Tail(S); k = \lfloor n/2 \rfloor$.

Paso 1. Selección probabilística de la actividad

Definir

$Candidate = \{j \,/\, j \in Tail(\gamma); Label(j) = 0 \text{ y } Label(j) = 1 \; \forall i \in P_j\}$,

$w_0 = \{\gamma_j \,/\, j \in Candidate\}$.

$Eligible = \{j \,/\, j \in Candidate \text{ y } \gamma_j - w_0 \leq window\}$.

Escoger aleatoriamente j en Eligible.

Paso 2. Selección probabilística del sunconjunto.

Si $C_NA \cup C_E = \emptyset$, entonces SELECT = C_NE. Ir al Paso 3.

Si no, calcular:

$\alpha = \max\{LF_i \,/\, i \in C_E\}$ (Si $C_E = \emptyset$, entonces $\alpha = 0$).

$\beta = \max\{LF_i \,/\, i \in C_NA\}$ (Si $C_NA = \emptyset$, entonces $\alpha = 0$).

Seleccionar r de (0,1) aleatoriamente.

Si $r \leq \alpha/(\alpha+\beta)$, SELECT = C_E. En otro caso, SELECT = C_NA.

Paso 3. Secuenciación de la actividad.

Secuenciar j tan pronto como sea posible teniendo en cuenta las relaciones de precedencia y los recursos empleados. $Label(j) = 1, \gamma_j' = k, k = k - 1$.

Paso 4.

Si $j = n$, S' es la secuencia construida. Devolver γ'', la redefinición de γ'. Stop.

En otro caso, ir al Paso 1.

Procedimiento de mejora de la cola

En esta fase, TAIL_IMPROVING(γ) es un método de muestreo aleatorio. Aplica WINDOW_SAMPLING_MOVE(γ) cien veces con window = 2 y otras 100 veces con window = 5.

Esquema de la Fase 2 y de CARA

La Fase 2 es muy similar a la Fase 1, sólo hay que tener en cuenta la nueva forma de construir la población y de mejorar las secuencias. Existe otra diferencia: en la primera fase nos interesaba aplicar la función intensificadora a todos los individuos de la población inicial dado que, en otro caso, podíamos simplemente escoger un N1 más pequeño. En este caso, sin embargo, nos queremos mover en el entorno de una solución que sea lo mejor posible. Por eso, cada vez que se encuentra un vector OT que mejora la mejor solución global, la Fase 2 comienza de nuevo desde esa solución. Las soluciones de la población anterior no son eliminadas, sino que se unen a las nuevas secuencias construidas y de entre todas se escogen las mejores.

Figura 2.16. Esquema de la Fase 2(γ,N2)

1. POP = INITIAL_SET_2(γ,N2).
2. HEAD_TAIL_IMPROVING_2(POP), donde POP está ordenada en orden creciente de duración.
3. Devolver la mejor solución obtenida.

Se puede establecer una conexión entre la Fase 2 y GRASP (cf. apartado 7.5 capítulo 1). Cada aplicación de esta fase a un γ dado puede ser interpretada como la ejecución de varias iteraciones GRASP. El procedimiento β-Biased se ajusta bastante al proceso constructivo de una solución de un GRASP: se va construyendo una solución añadiendo en cada iteración una componente de la solución; las decisiones se toman sesgando las probabilidades de los posible elementos (las actividades) de acuerdo a una función 'greedy' que consiste en seleccionar la actividad con menor prioridad en γ; las probabilidades varían en cada iteración, adaptándose a las decisiones tomadas en previas iteraciones. El otro elemento que comparten GRASP y la Fase 2 es la aplicación de un procedimiento de mejora, en este caso HEAD_TAIL_IMPROVING_2, con una pequeña diferencia, en este caso sólo se emplea con las secuencias más prometedoras, las N2 mejores de las calculadas. Existe una diferencia más importante con GRASP, porque la Fase 2 va más allá. Cada vez que se encuentra con una mejor

solución γ', el proceso entero vuelve a comenzar a partir de γ'. Es una ventaja sobre los métodos GRASP habituales, dado que se puede cambiar de función greedy. Visto desde otra perspectiva, se puede decir que esta fase recorre un camino en el espacio de los óptimos locales para HEAD_TAIL_IMPROVING_2, de manera análoga a como lo hacen otros procedimientos conocidos como métodos de perturbación ('perturbation approaches', cf. Martin et al., 1992).

El esquema del algoritmo completo CARA aparece en la Figura 2.17.

Figura 2.17. Esquema de CARA(N1,N2)

1. γ = Fase 1(N1).
2. γ_1 = Fase 2(γ,N2).
3. Devolver S(γ_1).

3. El algoritmo HIAC

Mientras que CARA se basa en las relaciones de precedencia y en la situación global de las actividades (en qué lugar de la solución se están secuenciando), HIAC se centra en los recursos y en las actividades que se secuencian en paralelo (con qué otras actividades se están secuenciando).

HIAC es un heurístico basado en poblaciones que incorpora diferentes estrategias para generar y hacer evolucionar una población de secuencias. El método consta de 3 fases. En la Fase 1 se aplica una función de mejora, HIA, dirigida a mejorar localmente la utilización de recursos, a todos los individuos de una población inicial. HIA incorpora un mecanismo oscilatorio que alternativamente dirige la búsqueda a dos regiones diferentes del espacio de soluciones. En la Fase 2 se trata de hacer evolucionar la población resultante de la primera fase mediante la aplicación alternativa de un mecanismo de combinación de secuencias – que integra características de Búsqueda dispersa y de Reencadenamiento de Trayectorias – y de HIA. La combinación de soluciones está dirigida a obtener nuevas secuencias que compartan rasgos con las mejores soluciones de la población. Para ello, las secuencias se calculan en la envoltura convexa de éstas. La Fase 3 tiene como punto de partida la mejor solución obtenida hasta el momento y su finalidad es explorar con detenimiento entornos cercanos a secuencias de gran calidad. Primero se genera una población por medio de uno de los muestreos definidos anteriormente, para después aplicar HIA a la población resultante. Cada vez que la actuación de HIA sobre un elemento produzca

una mejora global se interrumpe la exploración actual y se comienza una nueva exploración a partir de la nueva mejor secuencia.

HIAC combina las codificaciones por listas de actividades y por órdenes topológicos. Salvo cuando se redefina se empleará en ambas la regla de desempate de la menor actividad. Dada S una secuencia activa denotaremos por $\gamma(S)$ y $\lambda(S)$ a sus únicas representaciones OT y AL respectivamente. Emplearemos S, $\gamma(S)$ y $\lambda(S)$ indistintamente.

3.1. El algoritmo de intervalos homogéneos: un procedimiento de mejora

Todas las secuencias posibles de un proyecto utilizan la misma cantidad total de unidades de recursos $TR = \sum_{i=1}^{n} \sum_{k=1}^{K} r_{i,k}$, pero no la utilizan necesariamente durante el mismo número de unidades de tiempo. Dada una secuencia S, la utilización media de recursos se puede definir como TR/T(S), y obviamente aumenta conforme disminuye la duración de la secuencia. De hecho, es equivalente minimizar la duración de un proyecto que maximizar la utilización media de recursos. Además, el número de periodos de tiempo donde existe una baja utilización de recursos (con respecto a la media que alcanzan las soluciones óptimas) decrece, en general, al disminuir a todos los individuos de una población inicial la duración de la secuencia. Por ello, el Algoritmo de Intervalos Homogéneos (the Homogeneous Interval Algorithm) **HIA**, intenta disminuir la duración de una secuencia activa mejorando localmente la utilización de recursos. Para ello, recorre la secuencia deteniéndose en cada intervalo homogéneo e intentando aumentar la utilización de recursos en dicho intervalo. La secuencia puede ser recorrida de izquierda a derecha o de derecha a izquierda según se aplique Forward_HIA o Backward_HIA, respectivamente. HIA es el resultado de aplicar alternativamente Forward_HIA y Backward_HIA hasta que no se obtiene ninguna mejora.

Estrictamente hablando, HIA opera en el espacio de las representaciones AL. A partir de una representación AL inicia un proceso iterativo de mejora en el que, para pasar de una representación AL a la siguiente, se adelanta la posición de un subconjunto de actividades y, consecuentemente, se retrasa la posición de otras actividades.

Ejemplo

Consideremos el proyecto de la Figura 2.18, y la secuencia posible S de la Figura 2.19. La media de utilización de recursos en S es TR/17 = 69/17 = 4.058. En el

intervalo [3,6] se utilizan 3 unidades de recurso por lo que se puede considerar que hay una baja utilización de recursos. Adelantando convenientemente en λ = (1 3 4 2 7 8 5 6 9) la posición de la actividad 6 se obtiene la lista de actividades λ' = (1 3 4 2 6 7 8 5 9) que al ser decodificada proporciona la secuencia S' de la Figura 2.20 cuya longitud es tres unidades más corta que la de S. La media de utilización de recursos de S' es TR/14 = 69/14 = 4.928.

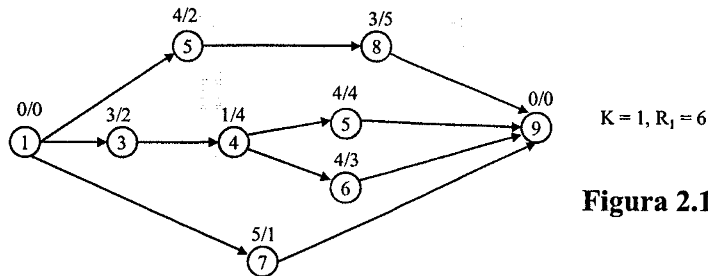


Figura 2.18

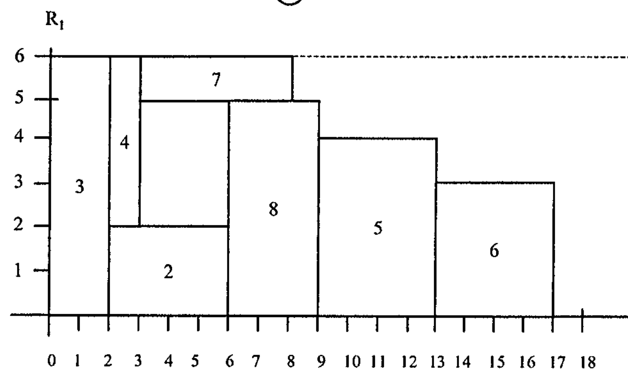


Figura 2.19

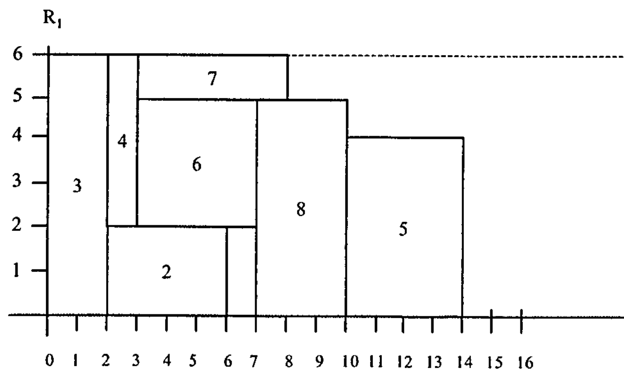


Figura 2.20

Intervalos Homogéneos

Definición: Intervalo Homogéneo

Dada una secuencia activa S, un **intervalo homogéneo** de S es un intervalo de tiempo $I = [I_s, I_f]$, con $I_s < I_f$, que cumple dos propiedades:

1) Toda actividad i que se secuencie en al menos una unidad de I se secuencia en todo I, es decir, $\{i \in \mathbf{V} ; [I_s,I_f] \cap]s_i,f_i[\neq \emptyset\} = \{i \in \mathbf{V}; [I_s,I_f] \subseteq [s_i,f_i]\}$.

2) No existe ningún intervalo de tiempo J que cumpla la propiedad 1) y que contenga estrictamente a I, es decir, I es maximal con respecto a la inclusión.

La propiedad 1) es equivalente a que no comienza ni finaliza ninguna actividad en $]I_s,I_f[$. La propiedad 2) implica que en I_s y en I_f comienza o finaliza alguna actividad; es más, la condición de secuencia activa de S conlleva que no puede comenzar ninguna actividad sin que finalice otra, por lo que 2) implica que en I_s y en I_f finaliza alguna actividad. En el primer intervalo homogéneo, que comienza en 0, la actividad que finaliza es la 0.

Estas consideraciones proporcionan una definición equivalente de intervalo homogéneo y una manera eficaz para su localización: Si $0 = t_0 < t_1 < ... < t_q = T(S)$ son los instantes en los que finaliza alguna actividad en S, el conjunto de los intervalos homogéneos viene dado por $\{[t_{j-1},t_j], j = 1, ..., q\}$.

Una consecuencia directa de 1) es que la utilización de cada tipo de recurso es la misma en todas las unidades del intervalo I, por lo que cada intervalo homogéneo es un intervalo de carga constante (según la terminología empleada en otros problemas de secuenciación, Valls et al., 1998). El recíproco no es cierto, porque dos combinaciones diferentes de actividades pueden consumir los mismos recursos.

Volviendo a la secuencia S en la Figura 2.19, cuya única representación AL es λ = (1 3 2 4 7 8 5 6 9), los intervalos homogéneos son aquellos cuyos extremos son dos instantes consecutivos de la siguiente lista: {0,2,3,6,8,9,13,17}.

Mejora de la utilización de recursos en un intervalo homogéneo

Para intentar obtener a partir de S una secuencia que utilice mejor los recursos y, por lo tanto, sea más corta, seguimos la estrategia de recorrer los intervalos homogéneos en orden creciente de I_s, intentando mejorar la utilización de los recursos en cada uno de ellos. Cuando el procedimiento llega a un intervalo homogéneo I, el procedimiento busca la mejor manera de utilizar los recursos cambiando las actividades que se secuencian en él. Como los intervalos anteriores ya han sido procesados, se considera

que las actividades que han comenzado antes de I_s ya están fijadas. Sin embargo, el conjunto de las actividades que comienzan en I_s puede ser sustituido por otro conjunto B de actividades que puedan comenzar en I_s y que mejore la utilización de recursos. Esta sustitución se realiza construyendo, a partir de $\lambda = \lambda(S)$, una nueva lista de actividades de manera que la posición de las actividades consideradas fijas no cambie y que se adelante la posición de las actividades de B. Antes de continuar la descripción del algoritmo necesitamos algunas definiciones.

Sea S una secuencia activa, λ una representación AL de S e I un intervalo homogéneo de S. Definimos **Fixed(I)** como el conjunto de actividades que se consideran ya fijadas, i.e., Fixed(I) = $\{i \in \mathbf{V} / s_i < I_s\}$; **FirstPosition(I)** = max$\{k / \lambda(k) \in$ Fixed(I)$\}$ + 1 la primera posición que puede ser cambiada; **Started(I)** = $\{i \in \mathbf{V} / s_i = I_s\}$ el conjunto de actividades que empiezan en I; **Candidate(I)** el conjunto de actividades no fijas que pueden secuenciarse en I_s sin violar las relaciones de precedencia, i.e., Candidate(I) = $\{i \in \mathbf{V} / s_i \geq I_s$ y $f_j \leq I_s \; \forall j \in Pred_i\}$.

AvRes(I,k) = $R_k - \sum_{i \in Fixed(I)/f_i > I_s} r_{ik}$ es la cantidad del recurso k que está disponible en I_s después de haber secuenciado las actividades en Fixed(I).

Una medida agrupada de la proporción de recursos que utiliza una actividad j es la **razón de utilización de recursos** ('Resource Utilisation Ratio'): RUR(i) $= \frac{1}{K}\sum_{k=1}^{K}\frac{r_{i,k}}{R_k}$.

Esta medida se puede generalizar de modo natural a un conjunto cualquiera B de actividades: RUR(B) = $\sum_{i \in B} \mathrm{RUR}(i)$.

Diremos que B $\subseteq$ Candidate(I) es un conjunto de actividades **elegible** sii:

(1) $\sum_{i \in B} r_{ik} \leq$ AvRes(I,k) $\forall$k.

(2) RUR(B) > RUR(Started(I)).

La condición (1) es equivalente a que las actividades de B se pueden procesar en I_s junto a las actividades de Fixed(I).

Volviendo a la secuencia S de la Figura 2.19, consideremos el intervalo homogéneo I = [3,6]. Entonces: Fixed(I) = {1,2,3,4}, FirstPosition = 5, Started(I) = {7}, Candidate(I) = {5,6,7}, AvRes(I,1) = 6 – 2 = 4 y RUR(Started(I)) = 1/6. Los conjuntos elegibles y sus razones de utilización de recursos son: {5}, {6}, {6,7} y 2/3, 1/2, 2/3, respectivamente.

Dados un intervalo homogéneo I de la secuencia λ y un conjunto elegible B, el operador **SET_SHIFT(B,I,λ)** devuelve la lista de actividades λ^B que se obtiene, a partir de λ, adelantando la posición de las actividades de B de manera que la nueva posición de la primera actividad de B sea FirstPosition(I) y todas las actividades de B ocupen posiciones consecutivas en orden creciente de su posición en λ.

En la secuencia S de la Figura 2.19, si escogemos el conjunto elegible B = {6,7}, el resultado de aplicar SET_SHIFT(B,I,λ) con λ'' = (1 3 2 4 7 8 5 6 9) es λ''' = (1 3 2 4 7 6 8 5 9) que conduce a la secuencia S' de la Figura 2.20.

Veamos que en la secuencia $S^B = S(\lambda^B)$ todas las actividades de B comienzan en, o antes de, I_s y que terminan después de I_s.

Al ser B un subconjunto de Candidate(I) se cumple $f_j \leq I_s$ $\forall j \in Pred_i$ $\forall i \in B$. Por lo tanto, las actividades de B pueden comenzar en I_s por relaciones de precedencia. Además, las actividades de B se sitúan en λ^B inmediatamente después de Fixed(I). Esto implica que, cuando van a secuenciarse las actividades de B, los recursos disponibles en cada unidad de tiempo de $[I_s, \infty[$ son al menos los disponibles en I_s, AvRes(I,k). Si a esto le añadimos que B cumple (1) por ser elegible, se puede asegurar que cada actividad i de B se puede procesar en $[I_s, I_s + d_i]$. Veamos que la condición de activa de S impide a las actividades del conjunto B finalizar antes de, o en, I_s. Supongamos que una actividad i de B pudiera procesarse en [a,b] con $b \leq I_s$ en S^B. Al secuenciarse i en las representaciones AL correspondientes, existen más recursos disponibles en $[0,I_s]$ en S que en S^B, por definición de Fixed(I). Esto implica que i se puede procesar en [a,b] con $b \leq I_s$ en S, lo que es una contradicción con que S sea activa.

Así pues, todas las actividades de B se secuencian (parcialmente) en el intervalo $[I_s, I_s + 1]$. Como por (2) RUR(B) > RUR(Started(I)), hemos conseguido nuestro objetivo de mejorar la utilización de recursos en un intervalo que comienza en I_s. La longitud de este intervalo dependerá de B y, en general, no será igual a la de I.

Fijémonos también en que: (i) λ^B y λ tienen en común las primeras FirstPosition–1 componentes; (ii) la posición relativa de las actividades de B entre sí es la misma en λ^B que en λ, y (iii) la posición relativa del resto de actividades entre sí también coincide en λ^B y en λ. Así pues, la modificación que el operador SET_SHIFT(B,I,λ) realiza sobre S es menor conforme I_s es mayor. Además, (i) implica que a mayor FirstPosition es necesario secuenciar en la práctica menos actividades, dado que podemos partir de la secuencia parcial de las actividades Fixed(I) en λ para construir λ^B.

Se puede establecer conexiones interesantes entre Serie y este proceso. Cada vez que nos fijamos en un I_s estamos trabajando con la secuencia parcial obtenida en Serie tras secuenciar Fixed(I). En la iteración correspondiente de Serie existe un conjunto de actividades elegibles, de las que Serie en general escoge una. En este caso Candidate(I) es precisamente el conjunto de elegibles, y simplemente calculamos un conjunto B de actividades elegibles que al secuenciarlas se puedan procesar en $[I_s, I_s+1]$.

Dado un intervalo homogéneo I de una secuencia λ, la función **Resource_Utilisation_Improving(I,λ)** genera todos los conjuntos elegibles, selecciona B* como el conjunto que maximiza RUR(B), obtiene $\lambda^{B^*}_{I}$ = SET_SHIFT(B*,I,λ) y devuelve λ^{B^*} una vez redefinida. La manera de desempatar entre conjuntos con la misma razón de utilización de recursos es escoger aquel que contenga la actividad con mayor razón de utilización de recursos. Si persistiera el empate se miraría la segunda actividad de cada conjunto con mayor RUR y así sucesivamente. Si no existen conjuntos elegibles se devuelve una lista de actividades ficticia con una secuencia asociada de longitud infinito.

Un dato importante para el tiempo de cómputo del algoritmo es darse cuenta que Resource_Utilisation_Improving resuelve óptimamente el problema

$$\left.\begin{array}{l}\text{Max } RUR(B)=\sum_{i\in B}\sum_{k=1}^{K}\frac{r_{i,k}}{R_k}\\ \text{s.a. } \sum_{i\in B} r_{i,k}\le AvRes(I,k)\ \forall k\\ B\subseteq Candidate(I)\end{array}\right\}(P)\ \overset{si\,k=1}{=}\ \left.\begin{array}{l}\text{Max } \sum_{i\in B}\frac{r_{i,1}}{R_1}\\ \sum_{i\in B} r_{i,k}\le AvRes(I,1)\\ B\subseteq Candidate(I)\end{array}\right\}\equiv\left.\begin{array}{l}\text{Max } \sum_{i=1}^{nCand} r_i x_i\\ \sum_{i=1}^{nCand} r_i x_i\le R\\ x_1,x_2,\dots,x_{nCand}=0,1\end{array}\right\}(P^*)$$

Si existiera un único recurso, P se transformaría en (P*), el llamado 'Subset-Sum Problem', que es NP-duro (cf. Martello y Toth, 1990). Esto quiere decir que la función Resource_Utilisation_Improving debe resolver un problema NP-duro. Los problemas más grandes con los que hemos probado HIAC contienen 120 actividades no ficticias, por lo que el conjunto Candidate(I) ha sido lo suficientemente pequeño como para poder resolver óptimamente el problema P en cada iteración sin que el tiempo de HIAC sea excesivo (ver resultados computacionales). Para un n mucho mayor habría que aplicar un heurístico en lugar de un exacto, por ejemplo, limitando el cardinal de B.

Forward_HIA

Una vez definida la función que mejora la utilización de recursos en un intervalo homogéneo podemos describir el algoritmo Forward_HIA, un procedimiento de mejora estricta. Dada una secuencia, λ, inicia una búsqueda para obtener una secuencia

mejor. Para ello, obtiene los intervalos homogéneos y los recorre en orden creciente de sus instantes de comienzo. Cada vez que un intervalo I es seleccionado, aplica el procedimiento Resource_Utilisation_Improving(I,λ) obteniendo una lista de actividades λ'. Si la duración de λ' es menor que la de λ, λ' se convierte en la nueva mejor secuencia y el procedimiento comienza de nuevo desde ella, dado que las nuevas condiciones pueden llevar a diferentes elecciones de B* o a que adelantamientos rechazados previamente produzcan mejoras. Si la longitud de λ' es mayor que la de λ, se considera que la dirección de búsqueda actual no es prometedora a pesar de que la utilización de recursos en I ha sido aumentada. Por ello, no se modifica el intervalo actual y se pasa a estudiar el siguiente intervalo homogéneo. En el caso en que T(λ') = T(λ), se continúa la exploración a partir de λ' y del siguiente intervalo homogéneo ya que, hasta el intervalo actual, λ' utiliza mejor los recursos que λ, por lo que si se mejora la utilización de los recursos en los intervalos posteriores al actual se podría llegar a una secuencia mejor. El procedimiento termina cuando ya no quedan más intervalos homogéneos por explorar. Si la secuencia obtenida no es mejor estrictamente que la original se devuelve la secuencia original, ya que no se ha podido mejorarla. En la Figura 2.21 se puede ver el esquema de Forward_HIA(λ), donde s'_i (f'_i) es el instante de comienzo (terminación) de la actividad i en la secuencia S' = S(λ') y t es un contador temporal que marca el intervalo homogéneo que va a ser estudiado.

Figura 2.21. Algoritmo FORWARD_HIA(λ)

1. t = 0; T = T(λ).
2. Mientras (t < T)
   - 2.1. Sea $I = [I_s, I_f]$ el primer intervalo homogéneo de λ que cumpla $I_s \geq t$.
   - 2.2. λ' = Resource_Utilisation_Improving(λ,I).
   - 2.3. Si (T(λ') < T): λ = λ'; t = 0 ; T = T(λ').
   - 2.4. En otro caso
     - 2.4.1. Si (T(λ') = T): λ = λ', $t = \min(f'_i \,/\, s'_i \leq I_s < f'_i)$.
     - 2.4.2. Si (T(λ') > T): $t = I_f$.
3. Devolver λ.

Backward_HIA

Forward_HIA(λ) trata de acortar una secuencia mejorando localmente la utilización de recursos, recorriendo de izquierda a derecha la secuencia, desde 0 hasta T. Esto lo lleva a cabo deteniéndose en cada final de actividad donde se replantea, teniendo en cuenta criterios de utilización de recursos, las actividades a procesar a partir de ese instante de tiempo. Una vez recorrida toda la secuencia, parece natural aplicar las

mismas técnicas pero recorriendo la secuencia de derecha a izquierda. Para ello, emplearemos las definiciones dadas en el apartado 2.2. de este capítulo.

Backward_HIA(λ) (Figura 2.22) comienza con la secuencia S obtenida por Forward_HIA(λ). Primero, justifica S a la derecha desempatando según λ y obtiene S^R; después, procede de manera análoga, aunque simétrica, a como lo haría Forward_HIA(λ). Es decir, el nuevo procedimiento (Forward_HIA*) recorre S^R de derecha a izquierda, desde T hasta 0, se detiene en cada comienzo de actividad, las actividades predecesoras son consideradas sucesoras y viceversa, el decodificador secuencia las actividades en orden inverso al indicado por la lista de actividades, las actividades se secuencian lo más tarde posible, etc. Backward_HIA(λ) devuelve la mejor secuencia obtenida una vez ha sido justificada a la izquierda, desempatando según ν. Forward_HIA* es equivalente a Forward_HIA, simplemente trabaja con la red inversa.

Figura 2.22. Esquema de Backward_HIA(λ)

1. $\lambda_{best} = \lambda$.
2. μ = RIGHT_JUSTIFICATION(λ).
3. ν = Forward_HIA*(μ^r).
4. η = LEFT_JUSTIFICATION(ν).
5. Si $(T(\eta) < T(\lambda_{best}))$ $\lambda_{best} = \eta^r$.
6. Devolver λ_{best}.

Observemos que este esquema es idéntico al de HEAD_IMPROVING (cf. Figura 2.12, apartado 2.2. de este capítulo), sustituyendo TAIL_IMPROVING por Forward_HIA.

HIA

HIA es el resultado de alternar los dos procedimientos Forward_HIA(λ) y Backward_HIA(λ) hasta que no se produzca mejora alguna, tal como hicimos con HEAD_IMPROVING y TAIL_IMPROVING.

Figura 2.23. HIA(λ)

1. μ = Forward_HIA(λ).
2. ν = Backward_HIA(μ).
3. Si $(T(\nu) < T(\mu))$ μ = Forward_HIA(ν). En otro caso, devolver μ.
4. Si $(T(\mu) < T(\nu))$ ir a 2. En otro caso, devolver φ.

El mecanismo oscilatorio descrito en HIA(λ) produce una interacción efectiva entre intensificación y diversificación. La búsqueda se lleva alternativamente a dos regiones diferentes, la región en la que la utilización de los recursos tiende a ser alta al principio de la secuencia y aquella en la que los recursos tienden a estar mejor utilizados al final de la secuencia, mientras el mecanismo intensificador es el mismo en las dos regiones.

3.2. El Algoritmo de Búsqueda Convexo: un método para combinar secuencias

En esta sección presentamos un procedimiento, el algoritmo de búsqueda convexo, para construir secuencias dentro de la región generada por un conjunto dado de secuencias. El algoritmo está basado en las representaciones OT de secuencias (cf. apartado 1 de este capítulo).

La suma como un operador para combinar vectores de prioridad

Se ha comentado que se puede emplear Serie como decodificador para obtener una secuencia $S(\gamma)$ a partir de un vector de prioridades γ, seleccionando en cada etapa la actividad j elegible de menor γ_j. Así pues, el orden en que se secuencian las actividades está determinado por las magnitudes relativas de las prioridades, no por sus valores absolutos. Dicho de otro modo, la calidad de $S(\gamma)$ está determinada por la importancia relativa de las actividades, indicada por las prioridades asignadas.

Si γ y γ' son vectores de prioridad compatibles con las relaciones de precedencia entonces $\frac{1}{2}\gamma + \frac{1}{2}\gamma'$ también lo es, pues si i es una actividad predecesora de j se cumple que $\gamma_i < \gamma_j$ y $\gamma'_i < \gamma'_j$ por lo que $\frac{1}{2}(\gamma_i + \gamma'_i) < \frac{1}{2}(\gamma_j + \gamma'_j)$. El vector $\gamma^{1,1} = \frac{1}{2}\gamma + \frac{1}{2}\gamma'$ es el punto medio del segmento que une γ y γ'. De la misma manera, el vector $\gamma^{2,1} = \frac{1}{2}\gamma + \frac{1}{2}\gamma^{1,1}$ ($\gamma^{2,2} = \frac{1}{2}\gamma^{1} + \frac{1}{2}\gamma'$) es el punto medio del segmento que une γ y $\gamma^{1,1}$ ($\gamma^{1,1}$ y γ'). Siguiendo iterativamente este proceso, en la iteración k-ésima, $k \geq 1$, habremos obtenido 2^k-1 vectores de prioridad que están en el segmento que une γ y γ'.

Por otra parte, si la actividad i es más importante que la actividad j según γ y también según γ' (es decir, $\gamma_i < \gamma_j$ y $\gamma'_i < \gamma'_j$), entonces también se cumple que la actividad i es más importante que la actividad j según $\gamma^{1,1}$ ($\gamma_i^{1,1} < \gamma_j^{1,1}$). Así pues, si los dos sumandos γ y γ' coinciden en la importancia relativa que asignan a dos actividades, entonces $\gamma^{1,1}$ mantiene la importancia relativa común. Esto quiere decir que la suma de dos secuencias $S(\gamma)$ y $S(\gamma')$ conserva de algún modo la parte común de ambas, una propiedad muy interesante para un operador que combina soluciones.

Ahora bien, si la actividad i es más importante que la actividad j según γ ($\gamma_i < \gamma_j$), pero menos importante según γ' ($\gamma'_i > \gamma'_j$), y $|\gamma_j - \gamma_i| < |\gamma'_j - \gamma'_i|$ o, equivalentemente, $D_{ij} = \gamma_j - \gamma_i < -D'_{ij} = \gamma'_i - \gamma'_j$, entonces $\gamma^{1,1}$ mantendrá la importancia relativa de γ' [$\gamma_j - \gamma_i < \gamma'_i - \gamma'_j \Rightarrow$ ½ $(\gamma_j + \gamma'_j) <$ ½ $(\gamma_i + \gamma'_i) \Rightarrow \gamma_j^{1,1} < \gamma_i^{1,1}$]. Si, por el contrario, $|D_{ij}| > |D'_{ij}|$ (y $D_{ij} > 0$, $D'_{ij} < 0$) $\Leftrightarrow D_{ij} > -D'_{ij}$, entonces $\gamma^{1,1}$ mantendrá la importancia relativa de γ.

Si γ^p, p=1,..., 2^k-1, denota los 2^k-1 vectores de prioridad generados en el segmento que une γ y γ' numerados en orden creciente de su distancia euclídea a γ ($\gamma^p = (1 - p/2^k)\,\gamma + p/2^k\,\gamma'$), entonces se cumple $D^p_{ij} = \gamma^p_j - \gamma^p_i = (1 - p/2^k)\,\gamma_j + p/2^k\,\gamma'_j - [(1 - p/2^k)\,\gamma_i + p/2^k\,\gamma'_i] = (1 - p/2^k)(\gamma_j - \gamma_i) + p/2^k\,(\gamma'_j - \gamma'_i) = (1 - p/2^k)\,D_{ij} + p/2^k\,D_{ij}'$. Esto quiere decir que para cada k y cada par i, j existe un número p', $0 \leq p' \leq 2^k$-1 de manera que los vectores de prioridad $\gamma^1, \ldots, \gamma^{p'}$ asignan la misma importancia relativa a i, j que γ y $\gamma^{p'+1}, \ldots, \gamma^{2^k-1}$ asignan la misma importancia relativa a i, j que γ'. En el caso p' = 0 (2^k-1), todos comparten la importancia de γ' (γ). Sin embargo, se puede asegurar que existe un k a partir del cual se tiene 0 < p' < 2^k-1, es decir, se generan vectores de prioridad que asignan la misma importancia relativa a i, j que γ y otros que asignan la misma importancia relativa a i, j que γ'.

Como veremos después, estas propiedades se trasladan de un cierto modo a la sucesión de secuencias $S(\gamma^p)$ respecto de $S(\gamma)$ y $S(\gamma')$.

Combinando secuencias

El procedimiento descrito en el apartado anterior podría ser utilizado para, dadas dos secuencias S y S', generar nuevas secuencias que combinen los órdenes relativos en que se secuencian las actividades según S y según S'. Para ello, primero habría que hallar dos vectores de prioridades γ y γ' tales que $S = S(\gamma)$ y $S' = S(\gamma')$; segundo, se calcularían los vectores γ^p, p = 1,..., 2^k-1, y, finalmente, se decodificarían dichos vectores obteniéndose los secuencias $S(\gamma^p)$, p = 1,..., 2^k-1.

Ahora bien, este procedimiento no estaría unívocamente determinado pues existen infinitos vectores de prioridad que representan a una misma secuencia S; las componentes de dichos vectores pueden ser arbitrariamente grandes o arbitrariamente pequeñas y la diferencia entre una prioridad y la siguiente puede ser arbitrariamente grande o arbitrariamente pequeña. Por otra parte, aunque se cumpla $\gamma_j < \gamma_i$, siempre se secuenciará i antes de j si la actividad i es predecesora de la actividad j. Estos inconvenientes desaparecen si en vez de utilizar vectores de prioridad trabajamos con representaciones OT de secuencias.

El algoritmo de búsqueda convexa

Dadas dos representaciones OT γ, γ' y un entero $k \geq 1$, el vector $\gamma^p = (1 - p/2^k)\,\gamma + p/2^k\,\gamma'$, $p = 1,\ldots, 2^k-1$, es un vector de prioridades compatible con las relaciones de precedencias que está en el segmento que une γ con γ'. Aunque γ^p no es un vector OT, pues no es una permutación de {1, ..., n}, puede ser fácilmente transformado en un vector OT que proporcione la misma secuencia que γ^p al ser decodificado. Sea **SEGMENT**(k,γ,γ') una función que devuelve los 2^k-1 vectores de prioridades γ^p después de haber sido transformados en vectores OT.

El algoritmo de búsqueda convexo ('Convex Search Algorithm', **CSA**) utiliza estas ideas para explorar la "envoltura convexa" de un conjunto de secuencias SET = {S^1, S^2,..., S^q}. Para ello, primero obtiene las representaciones OT de las secuencias en SET; después, aplica la función SEGMENT a cada par de vectores OT y, finalmente, devuelve los vectores OT generados después de ser decodificados.

El CSA puede ser descrito así:

Figura 2.24. CSA(k,SET)

1. Sean $\gamma^1, \gamma^2, \ldots, \gamma^q$ las representaciones OT de las secuencias en SET.
2. POP = $\varnothing$.
3. Desde i = 0 hasta q –1, hacer
   3.1. Desde j = i + 1 hasta q, hacer POP = POP $\cup$ SEGMENT(k,γ^i,γ^j).
4. Devolver NEWSET = {$S(\gamma)$, $\gamma \in$ POP}.

Análisis de SEGMENT

Dados dos representaciones OT γ y γ' y dado $k \geq 1$ entero, la función SEGMENT(k,γ,γ') genera 2^k-1 vectores de prioridades γ^p, $p = 1,\ldots, 2^k-1$, que están en el segmento geométrico que une los vectores γ y γ' de R^n. Además, por construcción, cualquier par de puntos consecutivos de la sucesión {γ, {γ^p, $p = 1, \ldots, 2^k-1$}, γ'} están siempre a la misma distancia euclídea. La pregunta que surge es ¿ocurre algo similar con los $2^k + 1$ secuencias activas {$S(\gamma)$, {$S(\gamma^p)$, $p = 1,\ldots, 2^k-1$}, $S(\gamma')$}?.

Para analizar la relación de las secuencias $S(\gamma^p)$, $p = 1,\ldots, 2^k-1$, entre sí y con $S(\gamma)$ y $S(\gamma')$ es necesario especificar una distancia entre secuencias.

Dada un secuencia activa S, sea **detrás(i,S)** el número de actividades que comienzan después que i en S, o sea, detrás(i,S) = $|\{j \,/\, s_j > s_i\}|$. Dadas dos secuencias S y S' definimos la distancia entre S y S' como $d(S,S') = (1/n)\sum_{i=1}^{n} |\,\text{detrás}(i,S) - \text{detrás}(i,S')\,|$.

Es fácil demostrar que d(S,S') cumple las propiedades de una función distancia (cf. apartado 6.2) en el espacio de las secuencias activas. En particular se cumple d(S,S') = 0 sii S = S'.

Hemos realizado el siguiente experimento: hemos seleccionado la primera instancia de cada uno de los 60 grupos de 10 instancias generados con la misma combinación de valores de los parámetros dentro del conjunto j120 de PSPLIB (cf. apartado 3 capítulo 1). Para cada instancia hemos generado aleatoriamente 20 parejas de secuencias activas, lo que hace un total de 1200 parejas. Para cada pareja de secuencias S, S' hemos obtenido sus representaciones OT γ, γ' y hemos aplicado SEGMENT(4,γ,γ') (k = 4 es el valor que se empleará en HIAC), obteniendo así 15 secuencias activas S^1, S^2, ..., S^{15}. Después hemos calculado el coeficiente de correlación r entre X = (1, 2, ..., 15) e Y = (d(S^1,S), ..., d(S^{15},S)), así como el coeficiente de correlación r' entre X e Y' = (d(S^1,S'), ..., d(S^{15},S')).

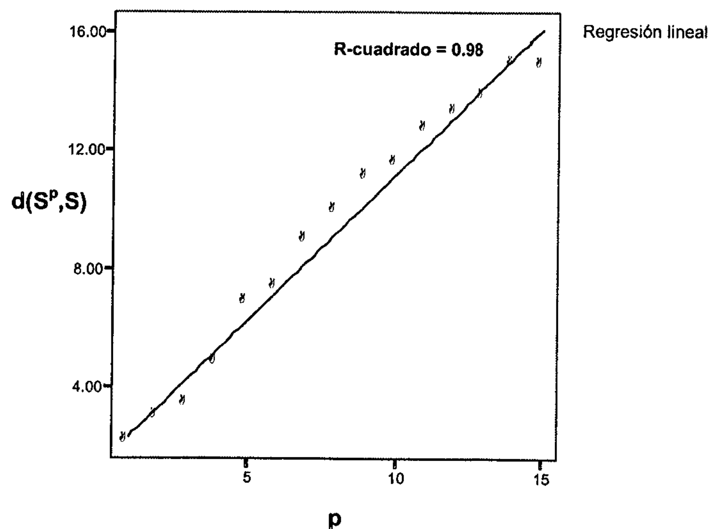


Figura 2.25. Nube de puntos y recta de mínimos cuadrados para un ejemplo de j1201_1

A modo de ejemplo, la Figura 2.25 muestra la nube de puntos {(p,d(S^p,S)), p = 1, ..., 15} obtenida en un ejemplo de la instancia j1201_1, junto con el coeficiente de determinación R^2 y la recta de mínimos cuadrados asociada.

La Figura 2.26 (Figura 2.27) muestra la nube de puntos {(d(S,S'),r), ∀S, S'} ({(d(S,S'),r'), ∀S, S'}). Podemos observar que, en general, r (r') toma valores cercanos

a 1 (-1), de hecho el 94'92 (95.58) % de los coeficientes de correlación son superiores (inferiores) a 0'95 (-0'95) y el 99'17 (99.00) % de los coeficientes de correlación son superiores (inferiores) a 0'9 (-0'9). Además, podemos observar que el número de veces que r (r') está por debajo (encima) de un cierto valor, digamos 0'95 (-0'95), disminuye conforme aumenta la distancia entre S y S'. Así pues, en general, el vector (1, 2, ..., p) explica gran parte de la variación de $d(S^p,S)$ y de $d(S^p,S')$ y podemos afirmar que la distancia $d(S^p,S)$ ($d(S^p,S')$) aumenta (disminuye) de forma aproximadamente lineal conforme p aumenta. Así pues, a medida que p aumenta, la sucesión de secuencias $S(\gamma^p)$ se aleja de $S(\gamma)$ al tiempo que se acerca a $S(\gamma')$. (Uno de estos hechos, en general, no implica el otro).

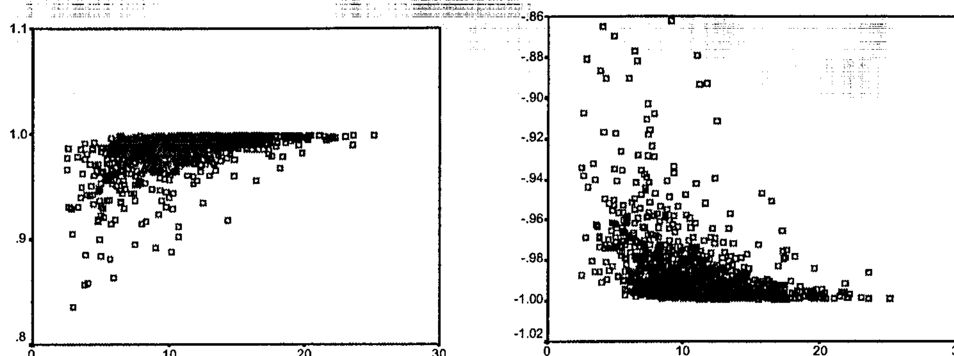


Figura 2.26 y 2.27. Nubes de puntos {(d(S,S'),r), ∀S, S'} y {(d(S,S'),r'), ∀S, S'}.

Veamos ahora cuán cerca están S^1 de S y S^{15} de S'. El gráfico de la Figura 2.28 muestra la nube de puntos $\{(d(S,S'),d^*(S^1,S)), \forall S, S'\}$, donde S^1 es la primera secuencia generada por SEGMENT(4,S,S') y la distancia de S^1 a S ha sido estandarizada de manera que, para toda pareja S, S', la distancia estandarizada de S a S' sea siempre 16, es decir, $d^*(S^1,S) = d(S^1,S)*16/ d(S, S')$. Los datos indican que en el 91'83 % de los casos la distancia estandarizada de S^1 a S está entre 0 y 3, es decir, que en el 91'83 % de los casos la distancia de S^1 a S es menor de 3/16 veces la distancia que separa S de S'. El gráfico de la Figura 2.29 muestra la nube de puntos $\{(d(S,S'),d^*(S^{15},S')), \forall S, S'\}$. En este caso, en el 91.17 % de los casos la distancia de S^{15} a S' es menor de 3/16 veces la distancia que separa S de S'. Así pues, en general, la sucesión S^p, p = 1,...,15, parte de un secuencia relativamente cercana a S y termina en un secuencia relativamente cercana a S'.

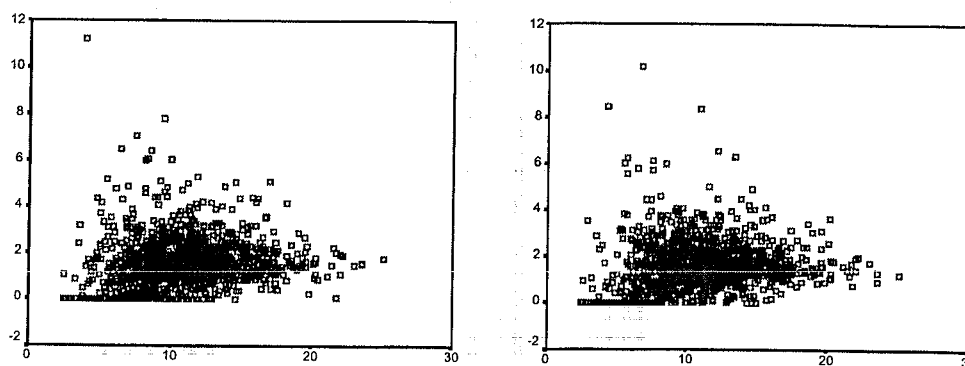


Figura 2.28 y 2.29. Nubes de puntos {(d(S,S'),d*(S^1,S)), ∀S,S'} y {(d(S,S'),d*(S^{15},S')), ∀S,S'}.

3.3. Algoritmo HIAC estándar

Fase 1 del algoritmo

La Fase 1 es equivalente a la Fase 1 en CARA, aplicando HIA en vez de HEAD_TAIL_IMPROVING. En este caso, sin embargo, la salida es la población mejorada de individuos en lugar de sólo la mejor solución.

Figura 2.30. Esquema de la Fase 1(N1)

4. POP = INITIAL_SET_1(N1).
5. POP' = HIA(POP).
6. Devolver POP'.

Fase 2 del algoritmo

CSA e HIA pueden integrarse en un procedimiento iterativo que primero explore la "envoltura convexa" de un conjunto de soluciones y después aplique HIA a las mejores secuencias para mejorarlas. Las mejores de las soluciones mejoradas constituyen el conjunto a explorar en la siguiente iteración.

Figura 2.31. Fase 2(POP,N3)

1. Desde i = 1 hasta 2, hacer
   1.1. POP5 = {las mejores 5 secuencias de POP}.
   1.2. POP' = CSA(4,POP5).
   1.3. POPN3 = {las mejores N3 secuencias de POP'}.
   1.4. POP = HIA(POPN3).
2. Devolver γ(S) con S la mejor secuencia obtenida.

Las dos primeras fases pueden ser interpretadas en términos de la metodología de la búsqueda dispersa (cf. apartado 7.6 capítulo 1). El método de generación diversificada consiste en la construcción de la población inicial y el de mejora es HIA. La manera de actualizar el conjunto de referencia es, simplemente, eliminar todas las soluciones de la iteración anterior y considerar exclusivamente las mejores secuencias obtenidas en el punto 1.4. La forma de generación de subconjuntos es generar todos los pares de soluciones, mientras que el método de combinación de soluciones difiere un poco de lo que es habitual en la búsqueda dispersa. En HIAC se sigue la estrategia del reencadenamiento de trayectorias (cf. apartado 7.3 capítulo 1) de ir incorporando cada vez más atributos de la solución guía en la inicial para obtener varias soluciones entre las dos iniciales, mediante el operador suma.

La Fase 3 de HIAC

Cuando este procedimiento se agota (final de la Fase 2), la mejor solución obtenida es presumiblemente de una calidad alta. Una vez agotadas las combinaciones de soluciones y que éstas han convergido a una única solución o a soluciones muy parecidas, tiene sentido trabajar exclusivamente con esa mejor solución y aplicar la fase 2 del algoritmo CARA, empleando HIA en lugar de HEAD_TAIL_IMPROVING_2.

Figura 2.32. Esquema de la Fase 3(γ,N1)

4. POP = INITIAL_SET_2(γ,N1).
5. HIA(POP), donde POP está ordenada en orden creciente de makespan.
6. Devolver la mejor solución obtenida.

Recordemos que esto es un esquema aproximado, puesto que una mejora global provoca una interrupción de la exploración actual para comenzar de nuevo la fase 3 a partir de la nueva mejor secuencia. Las soluciones no empleadas de la antigua población se incorporan a la nueva si son de suficiente calidad.

Algoritmo completo

Denominaremos HIAC(N1,N3) al algoritmo completo, que resumido en fases presentaría el siguiente aspecto:

Figura 2.33. Las diversas fases de HIAC(N1,N3)

1. POP = Fase 1(N1).
2. γ_1 = Fase 2(POP,N3).
3. γ_2 = Fase 3(γ_1,N1).
4. Devolver S(γ_2).

Denotaremos HIAC a HIAC(20,40).

3.4. Variaciones sobre el algoritmo estándar

HIAC_look_ahead

La función Resource_Utilisation_Improving(I,λ) es un procedimiento agresivo pero miope ('greedy'): escoge en cada intervalo el conjunto elegible B* que maximiza localmente la utilización de recursos. Sin embargo, pudiera ocurrir que dicha elección imposibilitara una buena utilización de recursos en los siguientes intervalos homogéneos. Por ello, parece adecuado que la función que selecciona el mejor conjunto elegible B* sea un procedimiento que tenga en cuenta la utilización de recursos en I_s y en periodos posteriores aunque, eso sí, ponderados con pesos decrecientes conforme se avanza en el tiempo.

El algoritmo HIAC_look_ahead se obtiene del algoritmo estándar con sólo definir la función Resource_Utilisation_Improving(I,λ) del siguiente modo:

Figura 2.34. Resource_Utisation_Improving(I,λ)

1. Si no existen conjuntos elegibles, devolver λ. En caso contrario,
2. Sea $M^* = \max\{15, \max\{|\{i \in B / s_i > I_s\}|, B \text{ elegible}\} + |Started(I)|\}$
3. Calcular $\mu(B)$ para todo B elegible de la siguiente forma:
   3.1. $\mu(B) = 0$.
   3.2. $\lambda^B = SET_SHIFT(B,I,\lambda)$.
   3.3. $S^B = S(\lambda^B)$.
   3.4. Desde p = FirstPosition hasta FirstPosition + M* –1, hacer
      3.4.1. Sea $i = \lambda^B_p$.
      3.4.2. Desde j = 1 hasta d_i, hacer $\mu(B)=\mu(B)+RUR(i)/(\max\{s^B_i - I_s,0\}+ j + 1)$.
4. Escoger B* elegible / $\mu(B)$ sea máximo.
5. Devolver λ^{B^*} una vez redefinida.

Fijémonos en que $\mu(B)$ estima la calidad potencial de seleccionar B como conjunto a ser avanzado. $\mu(B)$ es la suma ponderada, con pesos decrecientes, de la utilización de recursos de las primeras M* actividades que se secuencian según S^B en I_s o después. Desde el punto de vista computacional es interesante notar que para calcular $\mu(B)$, B elegible, no es necesario secuenciar todas las actividades cada vez. Como las primeras FirstPosition -1 actividades de todos los λ^B son las mismas, basta procesarlas una vez. Después, para calcular cada $\mu(B)$ es suficiente con secuenciar M* actividades cada vez.

Notar que secuenciar totalmente S^B, para todo B, sería demasiado costoso desde el punto de vista computacional. Como incluso el proceso descrito para calcular $\mu(B)$ puede alargar en exceso el tiempo de cómputo empleado si existen demasiados conjuntos elegibles, se acota el número a los que se aplica a 50. Si existen más de 50 conjuntos elegibles, se escogen únicamente los 50 con mayor razón de utilización de recursos.

HIAC_look_ahead_Paralelo

Para aumentar la potencia de la fase 2 del algoritmo HIAC_look_ahead se puede utilizar la siguiente estrategia. La población POP obtenida al final de la Fase 1 se ordena en orden no decreciente de longitud. Sean POP_impar = $\{S^1, S^3, \ldots, S^{N1-1}\}$ y POP_par = $\{S^2, S^4, \ldots, S^{N1}\}$. A cada una de estas poblaciones se le aplica la Fase 2 obteniendo POP_impar* y POP_par*, respectivamente (nos quedamos con la población resultante de esta fase en lugar de únicamente con la mejor solución). Finalmente se aplica otra vez la Fase 2 a la reunión de las dos poblaciones pero, en la primera aplicación de CSA, no se emplea SEGMENT con soluciones que provengan de la misma población, dado que pueden ser muy similares. De este modo, al principio, las dos poblaciones evolucionan en paralelo, combinándose por separado las características existentes en cada una de ellas. Esto se realiza con la esperanza de que las poblaciones POP_impar* y POP_par* contengan soluciones de gran calidad que habrán evolucionado hacia características distintas. Así pues, la aplicación de la Fase 2 a su reunión permite explorar una región no visitada anteriormente, partiendo de individuos no demasiado similares y de una calidad presumiblemente alta.

La estrategia descrita es una adecuación a las características operacionales del algoritmo del llamado modelo de islas ('island model') desarrollado por Kohlmorgen et al. (19) dentro de un esquema genético. En nuestro caso, en lugar de conectar las islas por migración, se reúnen las dos poblaciones resultantes de la evolución y se combinan entre sí.

4. PRUEBAS COMPUTACIONALES

En esta sección se presentan los resultados computacionales concernientes a los dos algoritmos estudiados en los apartados 2 y 3. Todos los experimentos de la tesis se han llevado a cabo en un PC AMD K6 2 a 400 MHZ. Los algoritmos han sido

programados en C y han sido evaluados en los problemas de PSPLIB (cf. apartado 3 capítulo 1). Las cotas superiores se tomaron en el 10 de septiembre del 2001.

Antes de ver los resultados vamos a explicar algunas notaciones que se emplearán en las tablas de este apartado y a lo largo de la tesis. Si A es un algoritmo, A(i) es la duración de la solución proporcionada por A en la instancia i. Del mismo modo, UB(i) (CPM(i)) es el valor de la solución contenida en PSPLIB (el valor del CPM) para esa instancia. Teniendo en cuenta esto, y que cjto es el conjunto de PSPLIB con el que se esté trabajando, definimos:

$\Sigma A = \sum_{i \in cjto} A(i)$, la suma de las duraciones de las secuencias obtenidas por A; $\Sigma PSPLIB = \sum_{i \in cjto} UB(i)$, la suma de las mejores soluciones conocidas; el número desv_UB A = $\frac{1}{|cjto|} \sum_{i \in cjto} \frac{A(i) - UB(i)}{UB(i)}$, la desviación media con respecto a las mejores soluciones de PSPLIB; desv_CPM A = $\frac{1}{|cjto|} \sum_{i \in cjto} \frac{A(i) - CPM(i)}{CPM(i)}$, la desviación media con respecto al CPM; max_desv A = $\max\{\frac{A(i) - UB(i)}{UB(i)}, i \in cjto\}$, la máxima desviación con respecto a las cotas superiores y mejor_sol A = número de instancias i / A(i) = UB(i), el número de soluciones donde A obtiene la misma duración que la mejor solución conocida.

Tanto desv_UB A como desv_CPM A, así como la suma, sirven para comparar la eficacia de los algoritmos. La desviación respecto del CPM es la medida de eficiencia más comúnmente empleada por los investigadores, dado que las cotas superiores cambian con el paso del tiempo. A pesar de ello, hemos incluido la desviación con respecto a la mejor solución porque es una medida mucho más intuitiva y proporciona una mejor estimación de la desviación con respecto del óptimo. Basta ver la fila de j30 para observar que, en general, el CPM se encuentra en media lejano al óptimo.

Con respecto a la información temporal, med_CPU A y max_CPU A representan la media de tiempos del algoritmo A y el máximo para todas las instancias.

Cuando se obvie el algoritmo A, las medidas definidas se referirán al algoritmo que marque la fila correspondiente.

Elección de los parámetros

En los nuevos algoritmos que se describen en esta tesis no se ha intentado buscar (salvo que se especifique lo contrario) la mejor combinación de parámetros posible. En

lugar de eso, hemos escogido una serie de parámetros para los algoritmos estándar y hemos obtenido diferentes algoritmos modificando alguno(s) de esos parámetros. Los parámetros de los algoritmos estándar se han calculado mediante unas pruebas preliminares sobre un subconjunto reducido del conjunto j120 (en CARA también se han realizado experimentos para calibrar mínimamente algún parámetro en j30, y en HGA, capítulo 5, en j30, j60 y j90), en general una o dos semillas de las 10 existentes. De este modo se contrarresta el posible (mínimo teniendo en cuenta el método de trabajo) efecto de trabajar con los mismos problemas con los que posteriormente se realizan las pruebas finales.

Los algoritmos modificados se han seleccionado teniendo en cuenta el significado de cada parámetro y cambiando aquellos que parecía que podían conducir a algoritmos o conclusiones interesantes.

Los algoritmos estándar

Las Tablas 2.2 y 2.3 resumen los resultados de CARA y HIAC. El algoritmo CARA para j30 varía con respecto a los otros conjuntos, puesto que se observó que era útil incrementar el tamaño de la cola, cambiando en la definición de Advance(S) y mino n/2 por n/4.

	ΣPSPLIB	ΣCARA	ΣHIAC	desv_UB CARA	desv_UB HIAC	desv_CPM CARA	desv_CPM HIAC
j120	74013	76356	75009	2.58	1.17	34.53	32.18
j90	45741	46247	45967	0.85	0.40	11.12	10.44
j60	38368	38671	38512	0.61	0.31	11.45	10.98
j30	28316	28335	28361	0.06	0.13	13.46	13.64

Tabla 2.2. Resultados computacionales (I) para j30, j60, j90 y j120.

	max_desv CARA	max_desv HIAC	mejor_sol CARA	mejor_sol HIAC	med_CPU CARA	med_CPU HIAC	max_CPU CARA	max_CPU HIAC
j120	10.05	5.48	198	223	17.00	14.52	43.94	60.80
j90	7.02	5.13	364	372	4.63	2.53	25.46	17.57
j60	6.32	5.26	371	385	2.76	1.14	14.61	7.03
j30	3.45	3.45	463	443	1.61	0.38	6.15	1.54

Tabla 2.3. Resultados computacionales (II) para j30, j60, j90 y j120.

La desviación media de ambos algoritmos respecto de las mejores soluciones es pequeña en j30, j60 y j90, y no demasiado grande en j120. El tiempo empleado para ello no es excesivo, ni siquiera el máximo para todas las instancias. Como era de esperar, tanto el tiempo como la desviación respecto a las mejores soluciones aumentan con el tamaño del problema. En la comparación entre CARA e HIAC hay un claro dominador, HIAC, que en j120 ofrece mucho mejores resultados con menor tiempo medio, aunque mayor tiempo máximo. Estas diferencias existen, aunque son menores, en j60 y j90, mientras que en j30 es el único conjunto en el que CARA obtiene mejor calidad, aunque necesita para ello entre 4 y 5 veces más tiempo.

Otras pruebas computacionales adicionales I: CARA

En la Tabla 2.4 se comprueba cómo distintos valores del número de soluciones en la población inicial (menores de 20) redundan en algoritmos interesantes, porque emplean menos tiempo que CARA y no ofrecen soluciones excesivamente superiores en duración. Aumentar la población inicial a 30 no aumenta la calidad pero sí el tiempo.

	Σ	desv_UB	desv_CPM	med_CPU
N1 = 5	76503	2.76	34.78	8.13
N1 = 12	76384	2.63	34.58	12.10
CARA (N1 = 20)	76356	2.58	34.53	17.00
N1 = 30	76379	2.59	34.57	23.55

Tabla 2.4. Distintos tamaños de población en la Fase 1.

	Σ	desv_UB	desv_CPM	med_CPU
N2 = 10	76546	2.79	34.86	14.23
CARA (N2 = 20)	76356	2.58	34.53	17.00
N2 = 30	76266	2.48	34.37	19.62
N2 = 40	76206	2.42	34.27	22.55

Tabla 2.5. Distintos tamaños de población en la Fase 2.

La Tabla 2.5 es útil para demostrar que una disminución en la población de la Fase 2 no merece la pena, dado que la disminución en calidad no se compensa con la del tiempo (N1 = 5, N2 = 20 es un algoritmo mejor). También confirma que aumentar esta población aumenta la calidad, pero a costa de emplear más tiempo de computación. Si juntamos estos resultados con los de la anterior tabla tenemos que si queremos rebajar el tiempo de CARA debemos rebajar el tamaño de la población inicial, y si

queremos aumentar la calidad, es preferible aumentar el tamaño de la población de la Fase 2. El tiempo que se quiera emplear marcará los tamaños concretos.

	Σ	desv_UB	desv_CPM	med_CPU
Fase 1 sólo, N1 = 20	77026	3.33	35.71	11.30
CARA	76356	2.58	34.53	17.00
Fase 1 sólo, N1 = 30	76965	3.24	35.60	18.17

Tabla 2.6. Efectividad de la segunda fase en CARA.

La Tabla 2.6 muestra la conveniencia de añadir la Fase 2. Primero, porque mejora sensiblemente la calidad obtenida en la Fase 1 y segundo, porque esta Fase por sí misma no consigue alcanzar la calidad de CARA aunque se le permita superarle en tiempo medio. La efectividad de esta fase parece avalar la suposición de que 'cerca' de soluciones de cierta calidad se pueden encontrar soluciones de la misma calidad e incluso mejores. En realidad lo que queda constatado es que la aplicación de una función de mejora F a soluciones de un entorno de una solución optimizada por F puede conducir, en ciertos casos, a mejoras globales.

Uno de los aspectos que no se ven reflejados en estas pruebas computacionales son los experimentos realizados con cotas inferiores mejores que el CPM. Las cotas que utilizamos, de las mejores existentes, sirven en teoría para reducir la ventana temporal donde se puede procesar cada actividad para obtener secuencias de una cierta calidad, lo que debe permitir reducir muy considerablemente el espacio de búsqueda y conducir a soluciones mejores. Lamentablemente, nuestra experiencia parece indicar que las cotas actuales no son suficientemente fuertes para acotar ese espacio lo suficiente, al menos con los procedimientos de búsqueda que hemos empleado. Los resultados que obteníamos no sólo no mejoraban la calidad del algoritmo significativamente, sino que el tiempo de cómputo se veía aumentado muy considerablemente, debido al coste de calcular las cotas. Una conclusión que parece surgir de esto es que para cada actividad (o al menos para un conjunto grande de ellas) existe en principio un intervalo relativamente grande donde puede secuenciarse y que la secuencia resultante sea de una calidad alta. Es posible que cuando se trate de calidades muy cercanas al óptimo esto no sea así, pero sí parece serlo en la calidad en la que se mueve CARA.

Otro aspecto mejorable en CARA es tener en cuenta explícitamente los recursos a la hora de completar las secuencias parciales con actividades del conjunto Advance y el resto. Los movimientos empleados son útiles hasta un cierto nivel de calidad, pero nuestra opinión es que se deben considerar los recursos y cómo se combinan las

actividades si se quiere mejorar ese nivel. HIAC sí trabaja con esas premisas y consigue obtener una calidad claramente mayor.

Otras pruebas computacionales adicionales II: HIAC.

	Σ	desv_UB	desv_CPM	med_CPU
HIAC(10,10)	75384	1.62	32.84	4.96
HIAC(20,20)	75180	1.37	32.48	9.75
HIAC = HIAC(20,40)	75009	1.17	32.18	14.52
Fase 1 HIAC sola	75355	1.57	32.78	6.82
HIAC sin Fase 3	75161	1.33	32.45	12.18
HIAC_look_ahead	74843	1.01	31.89	28.32
HIAC_look_ahead_ Paralelo	74671	0.81	31.58	59.43

Tabla 2.7. Resultados computacionales para diferentes versiones de HIAC.

La Tabla 2.7 muestra que los algoritmos HIAC(10,10) e HIAC(20,20) son dos alternativas interesantes a HIAC, obteniéndose con ellos menos calidad pero con bastante menos tiempo medio. La incorporación de la Fase 3 mejora a las Fases 1 y 2, y la aplicación de la Fase 2 mejora a la Fase 1 por sí misma. Tanto HIAC(10,10) como HIAC(20,20) igualan prácticamente en calidad a la Fase 1 y a HIAC sin la Fase 3, respectivamente, empleando para ello dos segundos menos de media. Esto indica la efectividad de utilizar las tres fases, a pesar de la dificultad de mejorar las soluciones que ya son de muy buena calidad (como se demostrará en la comparación con otros algoritmos), especialmente con la misma función intensificadora. En las últimas dos filas de la tabla se demuestra que las variaciones propuestas son muy interesantes, ya que permiten obtener una desviación menor con respecto a UB y CPM, aunque a costa de un mayor tiempo medio. El hecho de que la evolución en paralelo de las poblaciones mejore al enfoque estándar parece indicar que existen posibilidades de mejora en la segunda fase, en lo concerniente a la combinación de soluciones. Resulta concluyente que HIAC(10,10), el algoritmo que emplea menos tiempo de la Tabla 2.7, mejore sensiblemente la mejor de las versiones de CARA.

Comparación con otros heurísticos.

La comparación con otros heurísticos es complicada porque los datos publicados por los diferentes autores se refieren a diferentes ordenadores, empleando diferentes reglas de parada y expresando la calidad de las soluciones obtenidas de diferentes

modos. A pesar de esto, en algunos casos sí se pueden extraer conclusiones claras por lo que vamos a facilitar los resultados de los mejores heurísticos de los que tenemos noticia y a intentar obtener una medida de la bondad de CARA e HIA. Dado que, en general, no se dispone de la desviación con respecto a las cotas superiores, y que esta medida es mucho más intuitiva que la desviación con respecto al CPM, hemos incluido una columna con una estimación de esa desviación (en cursiva cuando es exacta). Hemos buscado un algoritmo B que cumpla desv_CPM B ≤ desv_CPM A y hemos empleado desv_UB B para estimar desv_UB A. Teniendo en cuenta desv_CPM, B obtiene al menos la misma calidad que A, por lo que hemos tratado de no ofrecer una desviación más grande de la real. Estas estimaciones no son útiles para comparar de forma clara algoritmos con una desviación respecto del CPM similar, pero sí para ofrecer una idea de las desviaciones con respecto a las cotas superiores.

La mayoría de los algoritmos heurísticos con los que vamos a comparar se han comentado en el capítulo 1, exceptuando los pertenecientes a Nonobe e Ibaraki y a Klein. Nonobe e Ibaraki, 1999, desarrollan un TS basado en listas de actividades para una extensión del RCPSP, con un enfoque similar al de Baar et al., 1997. Klein, 2000, describe un algoritmo tabú para otra extensión del RCPSP. En ambos trabajos se aplican los procedimientos al RCPSP, a los conjuntos de PSPLIB, en el primer caso a todos ellos y en el segundo exclusivamente a j30 y j60.

	desv_CPM	med_CPU	ordenador	nº secuencias	estimación desv_UB
Alcaraz y Maroto	36.57	9.37(i)	PC 166	5000	3.86
Dorndorf et al.	37.1	205	PC 200	5000	4.41
Hartmann (1)	36.74	13.15	PC 133	5000	3.88
Hartmann (2)	35.35	14.05	PC 133	5000	2.99
Hartmann (3)	35.55	≥ 17 (=CARA)	PC 400 (ii)	-	*3.32*
Merkle et al. (1)	36.65	25	PC 500	5000	3.86
Merkle et al. (2)	33.68	25 minutos	PC 500	-	1.98
Merkle et al. (3)	32.97	>> 25 min. (iii)	PC 500	-	1.62
Merkle et al. (4)	35.43	25	PC 500	5000	2.99
Möhring	36.2	65	Sun Ultra 2 200	-	3.81
Möhring (2)	35.3	≈ 65 (iv)	Sun Ultra 2 200	-	2.99
Nonobe e Ibaraki	34.99	645	Sun Ultra 2 300	-	*2.95*
Tormos y Lova	35.62	29.85	PC 200	5000 ó 7500 (v)	3.11

Tabla 2.8. Resultados de otros algoritmos en j120.

(i) Los tiempos medios del algoritmo del artículo Alcaraz y Maroto, 2001, son 20, 10 y 4 segundos para j120, j60 y j30, respectivamente. En estas tablas y en el resto de la tesis utilizamos los datos aparecidos en Alcaraz, 2001, correspondientes al mismo algoritmo, pero sensiblemente inferiores.

(ii) El Dr. Hartmann nos facilitó el programa ejecutable del algoritmo Hartmann (1). Al ejecutarse en entorno LINUX, calculamos el tiempo de CARA en un ordenador con LINUX y fuimos aumentando el número de secuencias de Hartmann (1) en cada conjunto hasta que el tiempo superó al de CARA. En consecuencia, se puede considerar que los resultados de Hartmann (3) (= Hartmann (1) con más secuencias) se obtuvieron en un tiempo no inferior al empleado por CARA.

(iii) El tiempo empleado en este caso no está especificado, si bien el máximo de secuencias calculadas es 8 veces mayor que en Merkle et al. (2), sin tener en cuenta la búsqueda local que emplean ambos algoritmos. Por ello es claro que el tiempo empleado será varias veces el de Merkle et al. (2), es decir, bastante más de 25 minutos.

(iv) Estos resultados se obtienen (cf. Möhring et al., 1999) al aplicar heurísticos de mejora simple, con un aumento marginal de tiempo empleado.

(v) En cada iteración de este algoritmo se calcula una secuencia con un muestreo aleatorio y esta secuencia se justifica a derecha e izquierda. Cada iteración se contabiliza en el artículo como dos secuencias, puesto que las dos justificaciones suman una secuencia y, de acuerdo a este cálculo, el algoritmo tiene un límite de 5000 secuencias. Según nuestra opinión deberían contabilizarse en total 3 secuencias, dado que una justificación puede cambiar los inicios de la mayoría de actividades, a priori no se conocen las actividades que van a poder justificarse (salvo en algún caso aislado que se sabe que alguna actividad no se va a justificar) y que justificar es secuenciar en un cierto orden (ver el apartado 3 del capítulo 3). Según nuestro cálculo el límite sería de 7500 secuencias.

La primera conclusión clara es que la calidad de HIAC y CARA es, en general, superior a la de los algoritmos de la Tabla 2.8. Únicamente Merkle et al. (2) y (3) ofrecen una calidad superior a la de CARA, pero con unos tiempos prohibitivos. De hecho, CARA mejora a Merkle et al. (1) y Merkle et al. (4), tanto en calidad como en tiempo computacional. Todas las versiones de HIAC obtienen una desviación menor que cualquier algoritmo de la Tabla 2.8. Cabe destacar que las mejores versiones de HIAC bajan del 32 (0.9)% con respecto al CPM (UB) con un tiempo medio no excesivo, mientras que la mejor desviación conocida hasta ahora era de 32.97 (≈1.62)%.

Si nos fijamos exclusivamente en el tiempo computacional, los tiempos de HIAC y CARA para la mayoría de sus versiones parecen (bastante) menores que los cuatro algoritmos de Merkle et al. y el de Dorndorf et al. Existe una forma bastante conocida de medir la rapidez de los microprocesadores, SPEC ('Standard Performance Evaluation Corporation'). Según esta medida, el microprocesador Sun Ultra a 200 Mhz obtiene un valor de 10.4 (7.44) cuando los cálculos se realizan en coma flotante (con valores enteros), mientras que un Pentium 2 a 400 Mhz obtiene 12.4 (15.8). Estos valores y la diferencia no demasiado grande entre un Pentium 2 a 400 Mhz y un AMD K6 a 400 MHz parecen indicar que el algoritmo de Möhring et al. emplea más tiempo que HIAC y CARA. Teniendo esto en cuenta, el tabú de Nonobe e Ibaraki utiliza bastante más tiempo que todas las versiones de CARA e HIAC. Esto indica que tanto HIAC como CARA mejoran a todos estos algoritmos. También parece claro que al menos la versión más rápida de HIAC no es más lenta que el algoritmo de Tormos y Lova, por lo que HIAC lo mejora. Es menos obvia la relación temporal entre la versión más rápida de CARA y este algoritmo, pero la diferencia en calidad parece señalar que CARA también supera a este algoritmo.

Los únicos algoritmos de la Tabla 2.8 claramente más rápidos que las versiones estándar de HIAC y CARA son Hartmann (1) y (2) y Alcaraz y Maroto. Hartmann (3) nos indica que el genético simple de Hartmann (Hartmann (1)) no es capaz de alcanzar la calidad de CARA aunque le demos el suficiente tiempo. Esto parece indicar (no lo asegura) que tampoco lo sería el genético de Alcaraz y Maroto, un algoritmo similar en muchos aspectos a Hartmann (1) y con una calidad sólo ligeramente superior. Esto implica que el único algoritmo con el que no está clara la comparación es Hartmann (2), el genético autoregulable. Teniendo en cuenta HIAC(10,10) y que la Fase 1 por sí misma, con 5 soluciones, obtiene una desviación con respecto al CPM (UB) de 34.50 (2.36) %, con un tiempo medio de 0.78 segundos, parece claro que HIAC sí mejora a Hartmann (2) (y de forma clara a Alcaraz y Maroto).

No hemos incluido en la tabla los resultados de Artigues et al., porque únicamente calculan 120 secuencias en cada instancia. La desviación con respecto al CPM que obtiene es de 39.34%.

Los conjuntos j90 y j60

Estos conjuntos no han sido tan empleados como el j120 para comprobar los heurísticos. En la Tabla 2.9 se pueden ver los resultados de los que disponemos.

	desv_CPM j90/j60	med_CPU j90/j160	ordenador	nº secuencias
Alcaraz y Maroto	- / 11.86	- / 3.60	PC 166	5000
Hartmann (1)	- / 11.89	- / 2.5	PC 133	5000
Hartmann (2)	- / 11.7	- / 2.52	PC 133	5000
Hartmann (3)	11.25 / 11.43	≥ 4.63 / ≥ 2.76 (=CARA)	PC 400	-
Möhring et al.	11.8 / 12.3	9.6 / 3.8	Sun Ultra 2 200	-
Nonobe e Ibaraki	11.25 / 11.55	181.41 / 26.49	Sun Ultra 2 300	-
Bouleimen y Lecocq	- / 11.90	-	-	5000
Klein	-/ 12.03	-	-	5000
Tormos y Lova	- / 11.82	- / 14.71	PC 200	5000 ó 7500

Tabla 2.9. Resultados computacionales de otros autores en j90 y j60.

A pesar de no poder realizar comparaciones apropiadas, tanto CARA como HIAC obtienen soluciones de mejor calidad (menor desviación respecto al CPM) tanto en j90 como en j60 (salvo CARA y Hartmann (3) en j60). Además, HIAC es el único de todos capaz de bajar de 11% de desviación respecto al CPM en ambos conjuntos. Respecto al tiempo, tanto Hartmann (3) como Nonobe e Ibaraki, Tormos y Lova y posiblemente Möhring emplean más tiempo (o igual en el caso de Hartmann (3) y CARA) que CARA e HIAC.

El conjunto j30

Este conjunto se diferencia de los demás en que se conocen todos los óptimos. Esto permite comparar desviaciones respecto de los óptimos (**desv_opt**) en lugar de respecto del CPM. La Tabla 2.10 muestra los resultados de los mejores heurísticos para este conjunto que conocemos.

	desv_CPM	med_CPU	ordenador	nº secuencias	desv_opt
Alcaraz y Maroto	-	1.44	PC 166	5000	0.12
Hartmann (1)	-	1.4	PC 133	5000	0.25
Hartmann (2)	-	1.4	PC 133	5000	0.21
Hartmann (3)	13.51	≥1.61(=CARA)	PC 400	-	0.09
Nonobe e Ibaraki	13.46	9.07	Sun Ultra 2 300	-	0.06
Bouleimen yLecocq	-	-	-	5000	0.23
Klein	-	-	-	5000	0.17
Tormos y Lova	-	4.72	PC 200	5000 ó 7500	0.15

Tabla 2.10. Resultados computacionales de otros autores en j30.

Todos los algoritmos están cercanos a las soluciones óptimas. Los mejores algoritmos en cuanto a calidad son CARA y el de Nonobe e Ibaraki, si bien emplean bastante más tiempo que el resto, en particular que HIAC. Un dato significativo de la tabla es la diferencia entre la desviación respecto del CPM y respecto de los óptimos, constatándose lo pobre que es, en general, esta cota.

Es obvio que tanto j30 como j60 y j90 son conjuntos de problemas mucho más sencillos que j120 (ver apartado 3 del capítulo 1). En ellos no se puede discriminar tan bien los algoritmos como con j120, debido a que los heurísticos no se diferencian tanto a la hora de resolver problemas sencillos como cuando tratan de afrontar los difíciles. A pesar de ello, los datos disponibles en j60 y j90 sugieren una superioridad de CARA y sobre todo de HIAC con respecto a la mayoría del resto de heurísticos. En j30 las diferencias entre los algoritmos son todavía más pequeñas, pero es destacable la calidad de CARA y la rapidez de HIAC.

5. CONCLUSIONES

Hemos desarrollado dos algoritmos basados en nuevos conceptos relacionados con las secuencias, los más importantes de los cuales son las actividades críticas para una secuencia y los intervalos homogéneos. Estos conceptos forman parte de las características de las secuencias y, por tanto, son importantes por sí mismos y pueden formar parte de algoritmos diferentes de los empleados aquí.

La implementación particular escogida para su explotación ha dado lugar a CARA, un algoritmo que parece superior en el conjunto j120 al resto de algoritmos exceptuando a Hartmann (2), y a HIAC, claramente por encima del resto. HIAC ha sido capaz de obtener en un tiempo razonable una calidad media no observada hasta ahora en la literatura, a pesar de la larga duración de ejecución de algunos de los algoritmos existentes.

Los buenos resultados obtenidos son fruto de la inclusión de técnicas específicas de la secuenciación de proyectos, que trabajan con características esenciales de las soluciones del RCPSP, como las funciones de mejora y los nuevos muestreos, y al uso de esas técnicas junto con ideas metaheurísticas que las potencian.

Uno de los elementos clave en ambos algoritmos es el mecanismo oscilatorio que se emplea en las funciones de mejora. En CARA es útil para alternar la búsqueda en la región con la cabeza fija y aquella con la cola fija; en HIAC consigue que se explore tanto la región donde la utilización de recursos tiende a ser alta al principio de las

secuencias como donde los recursos tienden a estar mejor empleados en la parte final de las soluciones. Este mecanismo no es más que combinar la búsqueda en el subconjunto de soluciones activas para la red original (donde se mueven la práctica totalidad de los algoritmos) y la región de soluciones activas para la red inversa. Ambos tipos de soluciones se podrían calificar de 'eficientes', ya que cada actividad está secuenciada lo más pronto posible en el caso de la red original y lo más tarde posible en la inversa. Por eso, tiene sentido emplear la función de mejora F que se tenga definida en ambos subconjuntos de soluciones, como hacen CARA e HIAC. Este mecanismo tiene un enorme potencial dentro de los heurísticos para el RCPSP, dado que se puede emplear junto a otro tipo de técnicas diferentes de las empleadas en nuestros algoritmos.

Los elementos comunes de CARA e HIAC se pueden abstraer hasta formar un esquema algorítmico dividido en módulos con una función determinada cada uno, capaz de producir algoritmos para el RCPSP con sólo especificar cada componente. En el apartado 7 de este capítulo se avanza en el desarrollo de este esquema.

6. Distancias en el RCPSP

En el apartado 3.2 hemos definido una distancia entre secuencias y la hemos empleado para demostrar que la suma era útil para obtener soluciones que enlazaran dos secuencias dadas. En este apartado se definen algunas funciones que pueden servir para medir la diferencia entre secuencias y se demuestra que algunas de ellas son distancias. Según nuestro conocimiento, nada se ha realizado al respecto en el RCPSP. Lo único relacionado es una medida de similitud entre listas de actividades (ver apartado 6.2) en el problema multimodo (Hartmann, 2001).

En el apartado 6.3 se explican varias aplicaciones posibles de las distancias en HIAC o CARA. Esto proporcionará una idea de las posibilidades que tiene el uso de las distancias en los distintos heurísticos para el RCPSP.

6.1. Definición de distancia

Definición: Distancia

Sea X un conjunto. Una **distancia** en X es una función

$d : X \times X \to R$

$(A, B) \mapsto d(A,B)$ que cumple las siguientes propiedades

i) $d(A,B) \geq 0 \; \forall A, B, \; d(A,A) = 0 \; \forall A \in X$.

ii) $d(A,B) = d(B,A) \; \forall A, B \in X$.

iii) $d(A,B) = 0 \rightarrow A = B \; \forall A, B \in X$.

iv) $d(A,C) \leq d(A,B) + d(B,C) \; \forall A, B, C \in X$.

Sea P una instancia del RCPSP y S(P) su espacio de soluciones posibles. Una **distancia (para el RCPSP)** es una función d que es distancia en S(P) para todo P. Si sólo cumple las propiedades i), ii) y iv) diremos que d es una **semidistancia** (para el RCPSP). Diremos que una función d es una **distancia en AS** si es una distancia en AS(P) para todo P, siendo AS(P) el conjunto de las secuencias activas de P.

La propiedad ii) se denomina de simetría, d es simétrica cuando la cumple. Cuando d(A,B) = 0 se dice que A y B son iguales para d o que d las considera iguales. Si se cumple iii) a menudo se interpreta como que d discrimina o distingue las secuencias.

Cualquier distancia lo es también en AS y también es semidistancia, pero los recíprocos no son ciertos. En principio los conceptos distancia en AS y semidistancia no tienen una relación clara, pero la mayoría de distancias en AS que veremos serán semidistancias que sólo son capaces de distinguir entre secuencias activas.

Ejemplos triviales

1) $d(A,A) = 0$, $d(A,B) = 1 \; \forall A \neq B$ es una distancia.

2) $d(A,B) = |T(A)-T(B)|$ es una semidistancia. Esta d considera dos secuencias iguales si tienen la misma longitud. Obviamente no es distancia ni distancia en AS.

3) Dada una actividad i fija, $d(A,B)=|s_i^A - s_i^B|$ es una semidistancia que claramente no cumple iii) ni en S(P) ni en AS(P).

6.2. Ejemplos no triviales

Distancias basadas en los instantes de comienzo

Las distancias absolutas se basan en los inicios de las actividades en sus secuencias.

1) La distancia de los inicios absolutos $dist_1(A,B) = \frac{1}{n}\sum_{i=1}^{n}|s_i^A - s_i^B|$. Esta es sin duda una de las distancias más naturales, la media de las diferencias absolutas entre los inicios. Tanto en esta expresión como en el resto de definiciones, donde se divida

por n también se podría dividir por n-2, el número de actividades no ficticias. Una distancia similar es aquella en las que se suman las diferencias al cuadrado en lugar de los valores absolutos.

2) La distancia de los inicios relativos $dist_1'(A,B) = \frac{1}{n}\sum_{i=1}^{n}\left|\frac{s_i^A}{T(A)} - \frac{s_i^B}{T(B)}\right|$. Lo que mide esta función es la posición relativa de la actividad dentro de la secuencia a través de comparar su inicio con la duración total. Una actividad puede comenzar en el mismo instante en dos secuencias (e.g. $s_i = 50$ en A y en B) y que su colocación dentro de las secuencias sea muy diferente (e.g. $T(A) = 50 + d_i$, $d_i = 1$, $T(B) = 100$, i es la última actividad en A mientras que en B está en la mitad).

3) $dist_1''(A,B) = \max\{|s_i^A - s_i^B|, i \text{ de } \mathbf{V}\}$. Esta distancia tiene su interés porque presenta un comportamiento distinto a las demás: secuencias con representaciones AL muy diferentes pero donde los inicios de las actividades no varíen mucho están más próximas con respecto a esta distancia que secuencias con representaciones AL parecidas pero con alguna actividad cuyo inicio sea muy diferente. Esta distancia podría ser útil en algoritmos donde se modifiquen directamente los inicios de las secuencias, especialmente si los movimientos con lo que se trabaja producen cambios pequeños en éstos.

PROPIEDADES 2.1

1) $dist_1$ y $dist_1''$ son distancias.
2) $dist_1'$ es una distancia siempre que se predetermine como restricción que el inicio de la actividad 1 sea 0 y el inicio de n sea el máximo de los finales de las actividades. En otro caso es una semidistancia.

Demostración

En todos los casos están claras las condiciones i) y ii), por lo que se debe demostrar la iii) y la iv). Esta última se obtiene muy fácilmente en las tres funciones, debido a que el valor absoluto cumple la desigualdad triangular. Veamos que se cumple la propiedad iii).

1) Tanto $dist_1(A,B) = 0$ como $dist_1''(A,B) = 0$ implican $s_i^A = s_i^B$ $\forall i$, por lo que A = B y ambas funciones cumplen la condición iii) y son distancias.

2) Sea A y B dos secuencias con $dist_1'(A,B) = 0$. Esto implica $s_i^A\,T(B) = s_i^B\,T(A)$ $\forall i$, o, lo que es lo mismo, existe k para el que $s_i^B = ks_i^A$ $(s_i^A = s_i^B/k)$ $\forall i$, con $k = T(B)/T(A) > 0$. (T(A) no puede ser 0). Sea i (j) la actividad para la que se cumple $T(A) = f_n^A = s_n^A = f_i^A$ $(T(B) = f_j^B)$. Se cumple $f_j^B = s_j^B + d_j = T(B) = k\,T(A) \geq kf_j^A = ks_j^A + kd_j$. Pero $s_j^B = k\,s_j^A$,

por lo que $ks_j^A + d_j \geq ks_j^A + kd_j$ y, por lo tanto, $k \leq 1$ ($d_j \neq 0$). Pero $s_i^A + d_i = T(A) = T(B)/k \geq s_i^B/k + d_i/k$ y, también, $s_i^A = s_i^B/k$, con lo que $s_i^B/k + d_i \geq s_i^B/k + d_i/k$ y, por lo tanto, $k \geq 1$ ($d_i \neq 0$). Juntando ambas tenemos $k = 1$, lo que implica $s_i^A = s_i^B$ $\forall i$ y A = B. Q. E. D.

Distancias basadas en posiciones relativas

Definición: El concepto detrás

Llamamos **relac** de una actividad i al número de actividades de **V** relacionadas con i, i.e., relac(i) = {j / i es predecesor o sucesor de j}. Este concepto no depende de ninguna secuencia en particular, únicamente del grafo.

Dada una secuencia A, denominamos **detrás**(i,A) al número de actividades que comienzan después de i en A, i.e., detrás(i,A) = $\{j \,/\, s_j^A > s_i^A\}$. Análogamente se define **delante**(i,A) – con menor estricto – e **igual**(i,A) – con un igual –. Se tiene la igualdad n = delante(i,A) + igual(i,A) + detrás(i,A), $\forall$i.

El concepto detrás (y $dist_2$) ya se ha definido y empleado en el apartado 3.2. Definimos $dist_2(A,B)=\frac{1}{n}\sum_{i=1}^{n}\left|\text{detrás}(i,A)-\text{detrás}(i,B)\right|$, $dist_2'(A,B) = \frac{1}{n}\sum_{i=1}^{n}\left|\underline{\text{detrás}}(i,A)-\underline{\text{detrás}}(i,B)\right|$, donde $\underline{\text{detrás}}(i,A)$ = detrás(i) - relac(i). Obviamente $dist_2'(A,B) = dist_2(A,B)$ $\forall$A, B.

Lo que no es equivalente es $\|A\|_1 = \frac{1}{n}\sum_{i=1}^{n}\left|\text{detrás}(i,A)\right|$ y $\|A\|_2 = \frac{1}{n}\sum_{i=1}^{n}\left|\underline{\text{detrás}}(i,A)\right|$.

Análogamente se pueden definir funciones con delante ($dist_2''$) e incluso con igual ($dist_2'''$). Un posible nombre para $dist_2$ sería la distancia de los órdenes relativos.

LEMA 2.2

Dada A secuencia activa, diremos que ρ es un vector de prioridades **compatible con los inicios** de A si cumple $s_i^A < s_j^A \rightarrow \rho(i) < \rho(j)$. Entonces, si se secuencia (por el método Serie, secuenciando primero las actividades con menor prioridad) según ρ se obtiene A.

Demostración

Si construimos una lista de actividades λ a partir de ρ - a menor prioridad, menor orden - obtenemos una representación AL de S. Pero es obvio que aplicar Serie a λ es equivalente a aplicarlo sobre ρ. Como al secuenciar según una representación AL se obtiene la secuencia original, se tiene la demostración. Q. E. D.

Los vectores de prioridades compatibles con los inicios son similares a los vectores OT, pero las prioridades son números reales cualesquiera. Al igual que al aplicarle Serie a una representación OT de una secuencia S se obtiene S, también se cumple lo mismo al aplicarlo a un vector compatible con los inicios de S.

PROPIEDADES 2.2

1) $dist_2$ (y $dist_2$') es una semidistancia que es una distancia en AS.

2) $dist_2$" es una semidistancia que es una distancia en AS.

3) $dist_2$'" no es semidistancia ni distancia en AS.

Demostración

1) Es evidente que se cumplen las propiedades i) y ii) de la definición de distancia. También se cumple la propiedad iv), debido a que el valor absoluto cumple la desigualdad triangular. Veamos la propiedad iii). Sean A, B secuencias con $dist_2(A,B) = 0$, ¿podemos asegurar A = B? La respuesta es claramente negativa si no se exige que A y B sean activas. Basta considerar, por ejemplo, secuencias A y B tales que $s_i^B = s_i^A + 1$ $\forall i \neq 1$.

   Sean A, B secuencias activas tales que $dist_2(A,B) = 0$. Construimos el vector de prioridades ρ_A tal que la componente i toma el valor -detrás(i,A). Claramente ρ_A es un vector de prioridades compatible con los inicios de A: $s_i^A < s_j^A \rightarrow$ detrás(i,A) > detrás(j,A) $\rightarrow \rho_A(i) < \rho_A(j)$. Por el lema, y dado que A es activa, al secuenciar ρ_A se obtiene A. Análogamente podemos construir ρ_B, que es compatible con los inicios de B y al secuenciarlo obtenemos B, ya que B también es activa. Pero $dist_1(A,B) = 0$ $\Rightarrow$ detrás(i,A) = detrás(i,B) $\forall i \Rightarrow \rho_A = \rho_B \Rightarrow A = B$.

2) Análogo a 1) ($\rho_A(i)$ = delante(i,A) es un vector de prioridades compatible con los inicios de A).

3) Veamos con un contraejemplo de que la condición iii) no se cumple ni en S ni en AS. Consideramos el proyecto que tiene 3 actividades no ficticias sin relaciones de precedencia entre ellas, con una unidad de duración cada una y que consumen una unidad del único recurso existente, cuya disponibilidad es también 1. Sean A = {0,0,1,2,3} y B = {0,0,2,1,3}. Estas dos secuencias son activas, cumplen $dist_2$'"(A,B) = 0 y son distintas. La propiedad iv) tampoco se cumple ni en S ni en AS.

Q. E. D.

Si definiéramos delante(i,A) = {j / $s_j^A \leq s_i^A$}, se cumpliría delante(i,A) + detrás(i,A) = n $\forall$i, y entonces |delante(i,A) - delante(i,B)| = |n - detrás(i,A)- (n-detrás(i,B))| = |detrás(i,B) - detrás(i,A)|, con lo que en este caso $dist_2$''(A,B) = $dist_2$(A,B).

Existe una distancia homóloga a $dist_1$'', definida por $dist_2^{(iv)}$(A,B) = max{|detrás(i,A)-detrás(i,B)|, i de **V**}. Esta función es una distancia en AS, ya que $dist_2^{(iv)}$ (A,B) = 0 $\Rightarrow$ detrás(i,A) = detrás(i,B) $\forall$i y el resto de propiedades son triviales.

Distancias basadas en codificaciones y distancias basadas en movimientos

Generalmente los heurísticos trabajan con codificaciones de soluciones, no con las soluciones en sí; los cambios se realizan sobre las codificaciones y luego éstas se transforman en secuencias. Esto conduce a que, a menudo, la representación concreta de la solución tiene influencia en la solución que se obtiene tras aplicar las modificaciones. Si esto es así, y dado que se trabaja con codificaciones y no con soluciones, no es descartable medir las distancias entre soluciones a partir de distancias entre las codificaciones concretas con las que se trabaja. Tengamos en cuenta que, dadas las secuencias A y B y dos pares de codificaciones suyas a y a', y b y b', respectivamente, puede ser bastante más difícil moverse de a a b con los movimientos que se están manejando que de a' a b', por lo que *para el algoritmo utilizado* A y B estarían más alejadas en el primer caso que en el segundo.

Dadas dos secuencias A y B, codificadas respectivamente por listas de actividades a y b, se define la **distancia basada en la codificación** $dist_3$(A,a;B,b) = $\frac{1}{n}\sum_{i=1}^{n}|\text{orden}(i,a)-\text{orden}(i,b)|$. Esta función no es una distancia porque depende de a y b. La expresión equivalente si la codificación es por vectores de prioridades es $\frac{1}{n}\sum_{i=1}^{n}|\gamma_i-\eta_i|$, con γ y η los vectores de prioridades que codifican respectivamente a A y B. Nótese que estamos ante la misma función si se trabaja con vectores OT. Estas medidas de diferencias entre codificaciones son distancias en AS (no son semidistancias porque sólo tienen sentido en AS) siempre que las codificaciones sean únicas (e.g. representaciones AL u OT con una regla de desempate predeterminada) y el decodificador devuelva la misma solución al ser aplicado sobre la misma codificación. Supondremos a partir de ahora que esto último es cierto.

Definición: Distancia basada en movimientos dependiente de la codificación

Dadas dos secuencias A, B, codificadas por a y b respectivamente y un movimiento m, llamaremos **distancia por movimientos dependiente de la**

> **codificación** entre A y B y denotaremos dist(A,a;B,b) al número mínimo de movimientos m's que se han de realizar sobre a para obtener b.
>
> En el caso en que no exista un k tal que b se pueda obtener de a después de aplicar k movimientos consecutivos, diremos que dist(A,a;B,b) es infinito.

Existen movimientos para los que dist(A,a;B,b) es un entero (no negativo) para todas A, B, a y b.

Ejemplo

Si a = (1 2 3 4) y b = (1 4 3 2) representan a A y B respectivamente, y m es insertar una actividad en otra posición, dist(A,a;B,b) = 2, porque con un único movimiento no se puede obtener b a partir de a, pero si primero adelantamos 4 a la 2ª posición (y obtenemos (1 4 2 3)) y después adelantamos 3 por delante de 2, obtenemos b. Si m consiste en adelantar o atrasar la actividad 4, la distancia entre A y B con esas codificaciones es infinita.

Esta 'distancia' sólo lo es en determinados casos porque, en general, no tiene por qué cumplirse dist(A,a;B,b) = dist(B,b;A,a), dependerá del m. Si m es adelantar una actividad dentro de una lista de actividades, la distancia de (1 2 3 4) a (1 4 2 3) es 1, pero la distancia de (1 4 2 3) a (1 2 3 4) es 2. El resto de condiciones sí se cumple, la iii) porque si no necesitamos realizar movimientos para ir de a a b es que a = b y, por tanto, A = B. La iv) porque, dadas tres codificaciones a, b y c, para ir de a a b se puede ir primero de a a c y luego de c a b.

A partir de esta 'distancia' puede construirse otra donde no exista dependencia de las codificaciones particulares.

Definición: Distancia basada en movimientos

> Dadas dos secuencias A, B y un movimiento m, llamaremos **distancia basada en movimientos** a $dist_4(A;B)$ = min{dist(A,a;B,b), a codificación de A, b codificación de B}.

En principio, esta 'distancia' (de nuevo que se cumpla la simetría dependerá del m) es difícil de calcular. Pero no lo es si se trabaja exclusivamente con una única codificación de cada solución. Así, por ejemplo, una vez establecido un criterio de desempate, la representación AL y la representación OT son únicas, por lo que tiene sentido la siguiente definición. Dadas dos secuencias A, B y un movimiento m, llamaremos (por abuso de notación) **distancia basada en movimientos** a $dist_4(A;B)$ = dist(A,a;B,b), con a y b las únicas representaciones AL u OT de A y B,

respectivamente. También sería posible calcular el mínimo para todas las representaciones AL u OT de A y B, ya que al conocerlas todas podemos calcular la máxima y mínima posición de una actividad i en una representación AL de una secuencia S o su máximo y mínimo peso en una representación OT.

Un ejemplo: las distancias de Hartmann

Hartmann, 2001, ha definido una medida de similitud entre dos listas de actividades λ y μ para el problema mutimodo que se puede restringir al RCPSP. Sea $Q = \{\{i,j\} / i, j = 1, \ldots, n; i \neq j, i \notin P_j, j \notin P_i\}$. Sea $\alpha_{\{i,j\}}^{\lambda,\mu} = 1$ si i está antes de j en λ y μ o j está antes que i en λ y μ; 0, en otro caso. Se define la medida de similitud de Hartmann como $m_H(A,B) = \frac{1}{|Q|}\sum_{\{i,j\}\in Q}\alpha_{\{i,j\}}^{\lambda,\mu}$.

Esta medida de similitud se puede convertir fácilmente en distancias para AS de la siguiente manera :

- $d_H^1(A,B) = 1 - \frac{1}{|Q|}\sum_{\{i,j\}\in Q}\alpha_{\{i,j\}}^{\lambda,\mu}$, con λ y μ las únicas representaciones AL de A y B respectivamente. Previamente se tiene que haber definido una manera de escoger entre las diversas representaciones AL.

- $d_H^2(A,B) = 1 - \frac{1}{|Q|}\sum_{\{i,j\}\in Q}\overline{\alpha}_{\{i,j\}}^{A,B}$, donde $\overline{\alpha}_{\{i,j\}}^{A,B} = 1$ si $s_i < s_j$ en A y en B, o $s_i > s_j$ en A y en B o $s_i = s_j$ en A y en B; 0, en otro caso.

<u>PROPOSICIÓN 2.3</u>

d_H^1 y d_H^2 son distancias en AS.

<u>Demostración</u>

Veamos que d_H^1 es una distancia en AS. La demostración con d_H^2 es análoga. La propiedad i) se cumple porque $\sum_{\{i,j\}\in Q}\alpha_{\{i,j\}}^{\lambda,\mu} \leq |Q|$, la propiedad ii) es obvia. Veamos la iii) y la iv).

iii) Sean A y B tales que d_H^1(A,B) = 0. Entonces $\alpha_{\{i,j\}}^{\lambda,\mu} = 1 \ \forall \{i,j\} \in Q$. Esto quiere decir que todas las parejas tienen un orden relativo igual en λ y μ, por lo que $\lambda = \mu$ y A = B.

iv) Sean A, B y C tres secuencias activas y λ, μ y ν sus únicas representaciones AL, respectivamente. Queremos demostrar que d_H^1(A,B) $\leq$ d_H^1(A,C) + d_H^1(C,B). Esto

es equivalente a $|Q| + \sum_{\{i,j\}\in Q} \alpha_{\{i,j\}}^{\lambda,\mu} \geq \sum_{\{i,j\}\in Q} \alpha_{\{i,j\}}^{\lambda,\nu} + \sum_{\{i,j\}\in Q} \alpha_{\{i,j\}}^{\nu,\mu} \Leftrightarrow |Q| + |D| \geq 2^*|E| + |F|$, donde $D = \{\{i,j\} \in Q \,/\, \alpha_{\{i,j\}}^{\lambda,\mu} = 1\}$, $E = \{\{i,j\} \in Q \,/\, \alpha_{\{i,j\}}^{\lambda,\nu} = 1$ y $\alpha_{\{i,j\}}^{\nu,\mu} = 1\}$ y $F = \{\{i,j\} \in Q \,/\, \alpha_{\{i,j\}}^{\lambda,\nu} + \alpha_{\{i,j\}}^{\nu,\mu} = 1\}$. Está clara la relación $|Q| \geq |E| + |F|$. Supongamos, además, que se cumple $|D| \geq |E|$. Entonces tendríamos $|Q| + |D| \geq |E| + |F| + |D| \geq |E| + |F| + |E| = 2^*|E| + |F|$, que es lo que queremos demostrar. Falta pues ver la desigualdad $|\{\{i,j\} \in Q \,/\, \alpha_{\{i,j\}}^{\lambda,\mu} = 1\}| \geq |\{\{i,j\} \in Q \,/\, \alpha_{\{i,j\}}^{\lambda,\nu} = 1$ y $\alpha_{\{i,j\}}^{\nu,\mu} = 1\}|$. Sea $\{i,j\} \in Q \,/\, \alpha_{\{i,j\}}^{\lambda,\nu} = 1$ y $\alpha_{\{i,j\}}^{\nu,\mu} = 1$, veamos que $\alpha_{\{i,j\}}^{\lambda,\mu} = 1$. Cuando la posición de i en λ (μ, ν) sea menor (mayor) que la de j lo denotaremos $i \prec j$ ($i \succ j$) en λ (μ, ν). $\alpha_{\{i,j\}}^{\lambda,\nu} = 1$ implica $i \prec j$ en λ y ν o $i \succ j$ en λ y ν. Supondremos sin pérdida de generalidad que $i \prec j$ en λ y ν. $\alpha_{\{i,j\}}^{\nu,\mu} = 1$ implica $i \prec j$ en ν y μ o $i \succ j$ en ν y μ. Como $i \prec j$ en ν, se cumple $i \prec j$ en μ, por lo que $i \prec j$ en λ y μ y $\alpha_{\{i,j\}}^{\lambda,\mu} = 1$.

Q. E. D.

Fijémonos en que d_H^1 es un ejemplo de distancia basada en movimientos con una única representación, y d_H^2 uno de distancia por movimientos teniendo en cuenta todas las representaciones AL (u OT), dado que las parejas de actividades que incrementan la distancia entre A y B son exclusivamente aquellas que en cualquier representación AL (u OT) λ y μ de A y B presentan diferente comportamiento.

Veamos por qué d_H^1 se puede considerar como una distancia por movimientos con una única representación. Supongamos que tenemos dos secuencias A y B con λ y μ sus únicas representaciones AL. Consideramos el movimiento m de intercambiar, dentro de la lista de actividades, las posiciones de dos actividades adyacentes que no estén relacionadas (porque en otro caso el vector resultante no es una lista de actividades). Para cada pareja de actividades no relacionadas que no tienen el mismo orden relativo en λ y μ habrá que realizar un movimiento para ir de A a B (o equivalentemente, de λ a μ), por lo que el número de movimientos para ir de A a B es |Q| menos el número de pares de actividades no relacionadas que ya tienen el mismo orden relativo en λ y μ. La distancia de Hartmann para listas de actividades (d_H^1) calcula ese número pero lo normaliza para que esté entre 0 y 1.

Ejemplo

En un projecto con 3 actividades no ficticias sin relaciones entre ellas, si λ = (1 2 4 3 5) y μ = (1 4 3 2 5), necesitamos dos movimientos para ir de λ a μ, pasando primero por (1 4 2 3 5). El número de pares de actividades (no ficticias) que tienen el mismo orden relativo es 1 (el par {4,3}), por lo que efectivamente |Q| - 1 = 3 –1 = 2.

Análogamente, d_H^2 se puede considerar como una distancia basada en movimientos teniendo en cuenta todas las representaciones AL (u OT), ya que es una interpretación de d_H^1 desde el punto de vista de las secuencias. Estas dos distancias son muy parecidas, pero no son iguales. Cuando dos actividades i y j cumplen $s_i < s_j$ en A y $s_i = s_j$ en B, d_H^2 considera que se debe realizar un movimiento más para ir de A a B, puesto que al α correspondiente le asigna el valor 0. Sin embargo, d_H^1 puede no considerarlo necesario, porque i puede estar antes que j en ambas listas de actividades. En ese caso, i y j no tienen la misma relación dentro de las secuencias A y B pero sí en las listas de actividades.

Este ejemplo demuestra que no se cumple $d_H^1 \geq d_H^2$. Sin embargo, sí es cierto que $d_H^1 \leq d_H^2$ si la regla de desempate para calcular las representaciones cumple la propiedad (*): no existen A, B, i, j tales que $i \prec j$ en λ, $s_i^A = s_j^A$, $j \prec i$ en μ y $s_i^B = s_j^B$. Es decir, siempre que existe empate entre dos actividades se desempata a favor de la misma actividad. Veamos que $d_H^1 \geq d_H^2$. Sea $\{i,j\} \in Q$ / $\overline{\alpha}_{\{i,j\}}^{A,B} = 1$. Entonces (a) $s_i^A < s_j^A$ y $s_i^B < s_j^B$, (b) $s_i^A > s_j^A$ y $s_i^B > s_j^B$ o (c) $s_i^A = s_j^A$ y $s_i^B = s_j^B$. Si se cumple (a) ((b)), se tiene $i \prec j$ ($j \prec i$) en λ y μ por lo que $\alpha_{\{i,j\}}^{\lambda,\mu} = 1$. La condición (c) implica $i \prec j$ en λ y μ o $j \prec i$ en λ y μ (lo que lleva a $\alpha_{\{i,j\}}^{\lambda,\mu} = 1$), porque no se puede dar $i \prec j$ en λ y $j \prec i$ en μ (ni $j \prec i$ en λ e $i \prec j$ en μ), porque esto contradeciría (*).

6.3. Usos de las distancias

Las distancias (y, en general, las funciones para medir las diferencias o similitudes entre secuencias) pueden ser muy útiles para perfeccionar los algoritmos heurísticos para el RCPSP. La gran mayoría de heurísticos – entre ellos HIAC y CARA – exploran la región de soluciones posibles de una manera ciega, sin saber en ningún momento en qué parte del espacio de soluciones están ni si las soluciones que se construyen son muy similares o diferentes.

Mediante las distancias se pueden controlar relativamente los movimientos que realiza el algoritmo, si éstos llevan a secuencias muy similares o no. También se puede analizar si las mejores soluciones calculadas a lo largo del algoritmo están cercanas unas de otras o si las intensificaciones y diversificaciones están efectivamente explorando regiones cercanas y lejanas respectivamente. Estas informaciones pueden ayudar a mejorar las técnicas de manera que se corrijan los sesgos de los diversos procedimientos empleados, incluso durante la ejecución de los propios algoritmos.

A continuación vamos a indicar posibles aplicaciones de las distancias para mejorar HIAC y CARA. Las distancias podrían servir:

- Para controlar que las soluciones del conjunto SET en CSA no sean demasiado parecidas, para que las repeticiones de características no influyan en la evolución y el subespacio que se explore sea más grande.

- Para medir la diversidad después de cada CSA. De este modo se podría decidir el número de veces que se aplica esta función, ahora fijado a 2. Algunas mediciones parecen indicar que en algunas instancias las soluciones calculadas después del primer CSA se parecen excesivamente para que otro CSA tenga éxito, mientras que en otras, tras el segundo CSA, la población no ha convergido todavía y es susceptible de mejora.

- Para escoger el parámetro k de SEGMENT. Dependiendo de d(S,S') tiene sentido calcular más o menos soluciones mediante la combinación de las dos soluciones. No sólo eso, sino que se puede calcular la distancia entre las secuencias originales y las nuevas secuencias que se van generando, para decidir entre qué parejas es adecuado aplicar la suma de nuevo y cuándo dos secuencias son lo suficientemente parecidas como para que no merezca la pena hacerlo.

- Para calcular el parámetro β a aplicar en la última fase de CARA e HIAC. Dependiendo de la instancia y de la secuencia de la que se parte, el mismo β puede conducir a secuencias muy similares o muy diferentes. Con un pequeño muestreo preliminar con diferentes β's y calculando la distancia de las soluciones construidas a la original, se puede fijar el valor más adecuado del parámetro.

- Para calcular, del mismo modo, el valor más adecuado del parámetro window en la Fase 2 de CARA. Incluso se puede (también en β-Biased) variar dentro mismo del procedimiento si las soluciones que se calculan siguen estando demasiado alejadas o cercanas a la original.

Parece claro que las distancias pueden ayudar a realizar una búsqueda más controlada y sistemática y, en consecuencia, más eficaz. Pero esta información es relativamente costosa, por lo que parece especialmente adecuada para algoritmos que no estén muy limitados por el tiempo y sean capaces de alcanzar niveles altos de calidad.

Una posible línea de investigación futura es incorporar las distancias a un algoritmo que podría ser HIAC, o similar, en el que la Fase 2 de combinación de secuencias se fortaleciera (que HIAC_look_ahead_Paralelo funcione mejor que HIAC_look_ahead

parece indicar que existe posibilidad de mejora). Se trataría de particionar el espacio de soluciones en conjuntos de soluciones cercanas o clusters, empleando para ello las soluciones calculadas en la primera fase. A partir de esta división, la combinación de soluciones se llevaría a cabo dentro de los clusters (intensificación) o entre ellos (diversificación). De este modo, se podría controlar la evolución de la población para que ésta no fuera excesivamente rápida.

Conocimiento del problema

Los resultados de CARA e HIAC parecen indicar la posibilidad de que cerca de soluciones de calidad existen soluciones de calidad y de que combinando soluciones de calidad se pueden obtener soluciones de calidad. Las distancias pueden emplearse para investigar hasta qué punto estas hipótesis y otras semejantes son ciertas. De este modo se intentaría comprender mejor los espacios de soluciones del RCPSP que, además de ser un tema interesante por sí mismo, puede ayudar a diseñar mejores heurísticos.

7. UN ESQUEMA ALGORÍTMICO VÁLIDO PARA EL RCPSP

HIAC y CARA comparten una serie de componentes y estructuras que se pueden emplear para diseñar otros algoritmos. Estos elementos comunes definen un esquema algorítmico capaz de producir algoritmos para el RCPSP al especificar cada uno de los módulos o fases, en particular la función de mejora F.

El esquema sería el siguiente:

Figura 2.35. MetaRCPSP(F)

Fase 0. POB = Construcción Población Inicial.
Fase 1. POB' = F(POB).
Fase 2. S = Combinación de soluciones(POB').
Fase 3. Exploración del entorno(S).
Devolver la mejor solución encontrada en el algoritmo.

En los siguientes apartados se detallan someramente cada uno de estos puntos, incidiendo en los elementos que se dejan por concretar y algunas formas de hacerlo.

Fase 0. Construcción de la población inicial

Esta fase está dedicada a la formación de una población de secuencias iniciales. Para ello, se emplean algoritmos constructivos rápidos para obtener soluciones con estructuras diversas de relativa buena calidad escogiéndose, a continuación, las mejores. Una posibilidad obvia es la de las reglas de prioridad, de las que existe una amplia variedad y ofrecen una calidad alta para su coste computacional. También se puede realizar un muestreo con una regla de prioridad (cf. apartado 8.1 capítulo 1), así como añadir soluciones aleatorias, algo que puede resultar útil especialmente en instancias pequeñas. La función que realiza esta fase en CARA y HIAC es INITIAL_SET_1.

Fase 1. Aplicación de F

En esta fase se aplica la función F a cada individuo de POB. F es un procedimiento intensificador, o de mejora, que parte de una solución inicial activa S para llegar a otra solución activa de duración como máximo la misma que S. En principio, esa es la única restricción sobre F, que puede pasar durante la búsqueda por soluciones de mayor longitud o incluso no activas o imposibles. Esta función es la parte más importante del esquema, y de su eficacia en mejorar una solución dada, aprovechando sus características sin ofrecer siempre los mismos óptimos locales, dependerá el resultado final del algoritmo. Para corroborar esto último basta observar la diferencia entre CARA y las diferentes versiones de HIAC.

Cada una de las aplicaciones de F puede servirse de la información generada en esa iteración pero, en principio, no es conveniente que emplee frecuencias o características de otros individuos de POB ni de las soluciones obtenidas al actuar F sobre ellos, debido a que la combinación de soluciones se reserva para la Fase 2. La Fase 1 intenta aprovechar al máximo la diversidad de la población inicial, sin injerencias de óptimos locales muy atrayentes. Esto no excluye que se decida trabajar con |POB| = 1 si es conveniente.

Cabe destacar una forma de mejorar sustancialmente F, y es la de enlazar iterativamente, mientras se produzca mejora, las aplicaciones de F sobre el grafo original y sobre el inverso, tal como se ha hecho en CARA (Figura 2.13) y en HIAC (Figura 2.23). Este cambio provoca un aumento considerable del tiempo de cómputo debido a que, como mínimo, se dobla el número de veces que se emplea F pero, según nuestra experiencia, en general aumenta muy considerablemente la calidad del algoritmo.

Fase 2. Combinación de soluciones

En esta fase se construyen soluciones combinando las características de individuos de calidad (presumiblemente) alta. Esta fase se podría ver como una intensificación puesto que la búsqueda se restringe al subespacio de soluciones entorno a unas cuantas secuencias élite. A partir de ellas se evoluciona hacia soluciones cada vez más similares entre sí. Pero también se puede argumentar que esta fase diversifica la búsqueda, al descubrir y explorar nuevas regiones comprendidas entre subespacios ya explorados, que pueden estar muy alejados entre sí.

Existen muchas y diferentes maneras de combinar soluciones; una de las más naturales sería emplear un algoritmo genético. La que hemos implementado nosotros trata de construir soluciones para después aplicarle la función F. Un posible esquema podría ser el siguiente.

Figura 2.36. Combinación de soluciones

1. Sea S_1, S_2, ..., S_k las mejores k soluciones obtenidas en la Fase 1.
2. Para cada pareja de soluciones S_i y S_j construir k_2 soluciones que combinen las características de ambas.
3. Sea POB la población formada por las mejores k_3 soluciones del paso 2. Aplicar F a cada una de ellas.
4. Repetir k_4 veces 1-3, reemplazando en 1 'la Fase 1' por 'el punto 3'.
5. Devolver la mejor solución calculada.

La decisión principal a tomar en este esquema es la forma de realizar el reencadenamiento de trayectorias entre dos soluciones dadas A y B. Una de las posibilidades es emplear el operador suma tal como lo utiliza HIAC, pero no es la única. En el apartado 2.2.1 del capítulo 3 se verán más operadores con los que realizarlo. Un aspecto bastante positivo del esquema es que no es necesario que las soluciones obtenidas mediante el reencadenamiento sean mejores que los puntos de partida. Este hecho es clave en un algoritmo genético usual, donde la combinación es la principal y casi exclusiva vía de mejora, pero aquí el reencadenamiento se emplea para construir soluciones de partida para F, por lo que no es imprescindible mejorar antes de aplicar F. Esto aumenta considerablemente las formas de realizar el paso 2. Sí es exigible al operador que la calidad de los individuos fruto de la combinación no esté excesivamente alejada de la de las soluciones originales, para que existan posibilidades reales de que la aplicación de F produzca una mejora global.

Precisamente para intentar asegurar esto empleamos el reencadenamiento en HIAC, capaz de encontrar soluciones unas más cercanas y otras más alejadas de las originales. Esto fortalece tanto el efecto intensificador como el diversificador de la fase.

Obviamente el esquema empleado es receptivo a muchos cambios. Por citar algunos posibles, no todas las parejas tienen por qué combinarse, ni tenemos que escoger las mejores soluciones en los pasos 1 y 3; se puede intentar que POB sea lo más diversa posible dentro de unos límites de calidad. Además, la soluciones combinadas desaparecen y no se vuelven a combinar, por lo que se pierden. Combinar soluciones de varias iteraciones aumentaría las posibilidades del esquema. Otra idea que ya ha demostrado su efectividad en HIAC es la de hacer evolucionar 2 (o más) poblaciones en paralelo para después combinar las características de los mejores individuos obtenidos.

Fase 3. Exploración de regiones de alta calidad

Esta fase se basa exclusivamente en la mejor solución S encontrada en el algoritmo, suponiendo que las mejoras posibles por combinar las soluciones generadas hasta ese momento han sido debidamente explotadas previamente. El procedimiento consiste en obtener un cierto número de soluciones en un entorno de S y en aplicar F a las mejores de ellas. Cada uno de los individuos calculados comparte un número importante de características con S, y el resto es diferente; la distancia con S no es grande, está bastante limitada. Es decir, realizamos un muestreo en las inmediaciones de la mejor solución, intentando que las soluciones resultantes no sean ni demasiado similares a S ni tan diferentes que sea muy difícil que la calidad sea parecida. Si al aplicar F sobre una solución se obtiene una mejora global, se reitera la exploración comenzando con la nueva mejor solución, bien agotando previamente la población actual o incorporando los individuos no visitados a la nueva población (tal como hacemos en HIAC y CARA) si son de suficiente calidad.

Esta forma de actuación presenta similitudes con los procedimientos conocidos como métodos de perturbación ('perturbation approaches', cf. Martin et al., 1992), donde se perturban los óptimos locales para después aplicar la búsqueda local a las soluciones obtenidas.

La manera concreta de realizar este muestreo no es necesariamente a partir de la aplicación de β-Biased, el proceso que hemos empleado en HIAC y CARA; existen muchas formas de realizarlo, una de las cuales es emplear un operador binario y combinar S con soluciones aleatorias, de manera que se generen soluciones similares a S pero con 'pequeños' cambios aleatorios. También se puede explotar la información

histórica, algo que no contemplan ni CARA ni HIAC. Mediante esa información se puede intentar introducir características que no hayan sido visitadas o que lo hayan sido bastante veces pero no estén presentes en S.

Es interesante remarcar que en esta fase se puede emplear otra función de mejora que no sea F, para intentar explorar caminos no recorridos previamente, tal como realiza CARA. También se pueden combinar (con un esquema similar al de la Fase 2 o con un algoritmo genético) las soluciones generadas mediante el muestreo, para intensificar aún más la búsqueda.

Ejemplo

CARA e HIAC son dos ejemplos de utilización del esquema; en el primero se prescinde de la segunda fase y se utilizan dos funciones diferentes en la Fase 1 y 3 (llamada fase 2 en CARA), mientras que en el segundo se aplican todas las fases, algo que se puede hacer debido a la mayor rapidez de HIA.

Para mostrar que este esquema no es válido solamente para estas dos funciones vamos a estudiar el caso de una función F simple y conocida, el 2-opt, cuyo esquema está representado en la Figura 2.37.

Figura 2.37. Función 2-opt(S)

1. Sea λ una representación AL de S. Sea $S^{best} = S$.
2. Desde i = 2 hasta n-2, hacer
   2.1. Sea λ' el vector de actividades tal que $\lambda'(j) = \lambda(j)\ \forall j \neq i, i+1$ y $\lambda'(i) = \lambda(i+1)$, $\lambda'(i+1) = \lambda(i)$.
   2.2. Si λ' es una lista de actividades, hacer $S' = S(\lambda')$. Si $T(S') < T(S^{best})$, hacer $S^{best} = S'$ y $\lambda^{best} = \lambda'$.
   2.3. En otro caso, eliminar λ'.
3. Si $T(S^{best}) < T(S)$, $S = S^{best}$ e ir a 2 con $\lambda = \lambda^{best}$. En otro caso, devolver S.

El algoritmos MetaRPCSP(F) utiliza las mismas funciones y parámetros que HIAC(20,20) salvo la función de intensificación que es reemplazada por 2-opt. También hemos construido 4 algoritmos más para poder analizar la aportación de cada componente de MetaRCPSP(F) al resultado final. El primero de ellos aplica F a un número suficiente de secuencias aleatorias (90) para que el algoritmo resultante emplee más tiempo que MetaRCPSP(2-opt). El segundo, aplica la Fase 0 y la Fase 1. El tercero aplica F a la población inicial de MetaRCPSP(2-opt) y además la aplica a un

cierto número de soluciones aleatorias (70) de manera que, de nuevo, el tiempo empleado supere a MetaRCPSP(2-opt). El cuarto, se compone de las Fase 0, 1 y 2.

La Tabla 2.11 presenta los resultados obtenidos por estos 5 algoritmos sobre el conjunto j120.

	Σ	desv_UB	desv_CPM
90 aleatorias Fase 1 F	82799	11.22	45.85
Fase 0 + Fase 1 F	79384	6.34	39.88
Fase 0 + Fase 1 F + Fase 1 F 70 aleatorias	79340	6.29	39.80
Fase 0 + Fase 1 F + Fase 2 F	78947	5.76	39.11
MetaRCPSP(F)	78488	5.19	38.31

Tabla 2.11. Resultados de los diferentes algoritmos relacionados con F.

En la Tabla 2.12 se muestran los resultados obtenidos por estos mismos algoritmos pero cambiando F por F', la función obtenida al enlazar iterativamente las aplicaciones de F sobre el grafo original y el inverso. En este caso se necesitan 70 y 50 secuencias aleatorias para sobrepasar el tiempo empleado por MetaRCPSP(F').

	Σ	desv_UB	desv_CPM
Fase 1 F' 70 aleatorias	77945	4.48	37.33
Fase 0 + Fase 1 F'	77528	3.97	36.59
Fase 0 + Fase 1 F' + Fase 1 F' 50 aleatorias	77387	3.80	36.35
Fase 0 + Fase 1 F' + Fase 2 F'	76900	3.23	35.49
MetaRCPSP(F')	76524	2.82	34.83

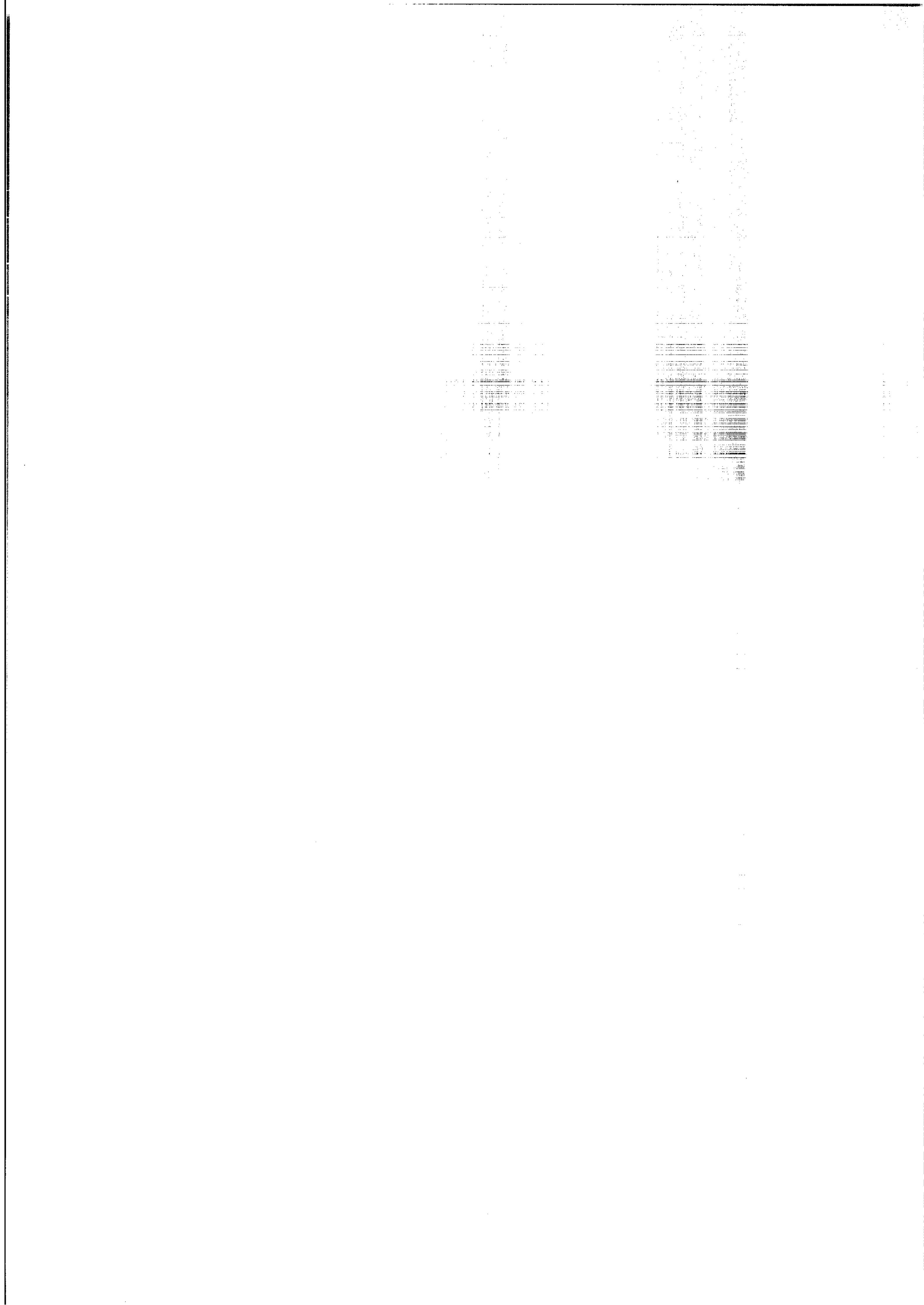
Tabla 2.12. Resultados de los diferentes algoritmos relacionados con F'.

Tanto MetaRCPSP(F) como MetaRCPSP(F') mejoran a los algoritmos sólo con soluciones aleatorias como a los que contienen la Fase 0 y soluciones aleatorias, y también a los algoritmos formados por las Fase 0, 1 y 2. Además la Fase 1 más la 2 mejora a la Fase 1. Obviamente los algoritmos completos emplean más tiempo que si se suprime alguna fase y, dependiendo de la función F con la que se trabaje y el tiempo que se quiera emplear, habrá que escoger un algoritmo u otro. Es importante

remarcar que seguramente se podrían conseguir mejores resultados al añadir la 2ª y 3ª Fase si se calibraran los parámetros. Un aspecto corroborado en estas pruebas es la mejora experimentada al cambiar de F a F' en todos los algoritmos. La desviación con respecto al CPM (UB) es casi 3.5 (2.5) % menor si se emplea F'. Esta mejora viene acompañada de un aumento considerable del tiempo, tardando MetaRCPSP(F') algo menos del doble que su homólogo con F. Sería sin embargo muy difícil que algún cambio de parámetros que equiparara los tiempos (aumentando el de MetaRCPSP(F)) pudiera equilibrar esos resultados. Esto concuerda con el resto de pruebas que hemos llevado a cabo con nuestras propias funciones. Un dato interesante es que la mejora de MetaRCPSP con respecto a la Fase 0 + Fase 1 con F' (ca. 1.76% en desv_CPM y 1.15% en desv_UB) es un poco mayor que la de con F (ca. 1.57% en desv_CPM y 1.15% en desv_UB), y eso que se mueve más de un 2% más cerca de las mejores soluciones.

Capítulo 3

La justificación



1. INTRODUCCIÓN

Tanto en CARA como en HIAC la justificación se empleaba fundamentalmente para dirigir alternativamente la búsqueda dentro de los conjuntos de secuencias activas para el grafo original y para el inverso. Sin embargo, ésta es solamente una de las maneras de utilizar la justificación. La justificación también puede ser empleada para reducir por sí misma la duración de una secuencia.

En la actualidad, la gran mayoría de heurísticos – entre ellos algunos de los mejores que nosotros conocemos – no emplean la justificación (cf. apartado 8 capítulo 1 para ver los algoritmos que emplean técnicas relacionadas). Según nuestro conocimiento, la justificación es una técnica muy antigua, casi tanto como los problemas de secuenciación de proyectos con recursos limitados (cf. Wiest, 1964). Hasta hace pocos años, el uso tanto de la justificación como de las técnicas hacia atrás se limitaba (según varios artículos de revisión bibliográfica) a los artículos de Li y Willis, 1992, y Özdamar y Ulusoy, 1996b (cf. apartado 8.1 capítulo 1). Recientemente, y paralelamente a la confección de la tesis, han surgido algoritmos heurísticos que de una manera u otra emplean estas técnicas.

Este capítulo quiere demostrar que la justificación es una técnica que es conveniente tener en cuenta a la hora de realizar cualquier heurístico para el RCPSP. Para ello mostraremos que se puede incorporar fácilmente a muy diversos algoritmos produciendo incrementos sensibles de la calidad de las secuencias con una variación pequeña – muchas veces negativa – en el tiempo de cómputo. La calidad media obtenida por algunos de estos procedimientos, muy simples y mediocres sin la justificación, llegará a superar la de los mejores heurísticos conocidos (con un límite de 5000 secuencias), poniendo de manifiesto el potencial de la técnica.

2. EL EXPERIMENTO

Como no es posible analizar los beneficios de incorporar la técnica de la justificación a todo tipo de algoritmos, hemos diseñado un conjunto de experimentos computacionales sobre cuatro conjuntos de algoritmos muy diversos. En el primero hemos querido incluir algoritmos sencillos conocidos y muy rápidos. Hemos escogido por su importancia histórica en el RCPSP dos reglas de prioridad, en particular la LFT y la WCS, y un método multipaso, el llamado método de muestreo aleatorio sesgado

basado en la peor elección mediante la regla LFT con $\alpha = 1$ (cf. apartado 8.1 capítulo 1). Además, estudiaremos el impacto que causa la justificación sobre las soluciones activas, que forman el espacio donde se mueven los heurísticos habituales, mediante su aplicación a soluciones activas aleatorias.

En el segundo conjunto hemos querido incluir algoritmos más evolucionados que los anteriores, pero que no queden cerca de los mejores heurísticos. Por ello, el segundo conjunto de algoritmos contiene algoritmos sencillos basados en poblaciones, específicamente diseñados por nosotros para estas pruebas. Todos los algoritmos comparten el mismo procedimiento para generar la población inicial (INITIAL_SET_1, empleado en HIAC y CARA), y, básicamente, el mismo esquema evolutivo que está reducido a la mínima expresión. En cada iteración, una cierta función actúa sobre las secuencias de la población actual para generar una nueva población. Estas funciones se construyen a partir de operadores binarios que combinan las características de dos secuencias para generar una nueva. Tomando como base estas funciones se construyen cuatro esquemas algorítmicos. Los 16 algoritmos de este segundo conjunto se obtienen de combinar de todas las maneras posibles los esquemas algorítmicos (4) y los operadores (4) definidos. La diversidad de operadores y de esquemas algorítmicos induce la variedad de los algoritmos.

Además de estos dos conjuntos, se añadirá la justificación a una función de mejora sencilla que es en sí misma un algoritmo de temple simulado. De este modo se desligará la mejora producida por la justificación de los algoritmos basados en poblaciones.

Por último, incorporaremos la justificación a un algoritmo no ideado por nosotros que proporciona soluciones de calidad (de acuerdo con los estándares actuales) por sí mismo mostrando, de este modo, que la justificación es capaz de aumentar sensiblemente su eficacia a la hora de encontrar mejores soluciones. El algoritmo escogido es el genético de Hartmann (cf. apartado 8.2 capítulo 1).

La doble justificación

La técnica de la justificación puede ser implementada de muy diversas maneras. Puesto que todos los algoritmos que vamos a describir trabajan con secuencias activas, es decir, con secuencias justificadas a la izquierda, nos ha parecido que la manera más sencilla de incorporar la justificación es justificar dos veces consecutivas cada secuencia activa S generada por el algoritmo, primero a la derecha y después a la izquierda, hasta obtener $(S^R)^L$. Sabemos que $(S^R)^L$ es activa y que se cumple $T((S^R)^L) \leq T(S^R) \leq T(S)$. A este proceso lo denominaremos **doble justificación**, y

hablaremos de algoritmos y de algoritmos con doble justificación (algoritmo + DJ). A pesar de esto, hay que tener presente que el concepto importante es el de justificar y que no es imprescindible aplicar la doble justificación para obtener mejoras. Más adelante intentaremos comprobar este extremo. La doble justificación ha sido empleada de forma independiente en el artículo Tormos y Lova, 2001, bajo el nombre de método BF ('Backward-Forward').

El experimento propuesto consiste en comparar la calidad de las secuencias obtenidas por cada uno de los 22 algoritmos antes y después de incorporar la doble justificación. Para poder comparar con otros heurísticos hemos decidido limitar el número de secuencias a 5000, y aplicar los algoritmos al conjunto j120 de PSPLIB (cf. apartado 3 capítulo 1).

2.1. Primer conjunto de experimentos

En este primer conjunto de experimentos hemos incluido cuatro algoritmos sin y con doble justificación. A continuación analizaremos, con un poco más de profundidad, la aplicación de la justificación a soluciones activas aleatorias.

El primer (segundo) algoritmo aplica Serie (Paralelo) y la regla LFT (WCS) en cada instancia. Si posteriormente se justifica dos veces la solución generada se obtiene el algoritmo con doble justificación.

El tercer algoritmo es el método de muestreo aleatorio sesgado basado en la peor elección con la regla LFT y con α = 1. Se aplica 5000 veces en cada instancia. La versión con doble justificación genera 1666 secuencias de la misma manera y después las justifica doblemente.

El cuarto algoritmo genera aleatoriamente 5000 secuencias activas y se queda con la mejor. El mismo algoritmo pero con doble justificación genera aleatoriamente 1666 secuencias que después justifica doblemente. Así pues, tanto el tercero como el cuarto algoritmo con DJ generan 1666*3 = 4998 secuencias en total.

La Tabla 3.1 resume los resultados en el primer conjunto de experimentos. La primera columna indica el algoritmo utilizado. La segunda columna muestra desv_CPM y desv_UB (entre paréntesis) respecto del algoritmo correspondiente. La tercera presenta la misma información pero referida al algoritmo con doble justificación.

Algoritmo	Sin justificación	+ DJ
LFT_Serie	48.11% (11.92%)	43.34% (8.63%)
WCS_Paralelo	43.58% (9.24%)	39.42% (5.91%)
Muestreo sesgado LFT	42.02% (7.74%)	37.47% (4.48%)
Algoritmo aleatorio	47.51% (12%)	38.36% (5.09%)

Tabla 3.1. Resultados para el primer conjunto de algoritmos en j120.

Se puede añadir información reveladora. La doble justificación ha mejorado la secuencia obtenida por la regla LFT en el 77.17 % de las instancias y la generada por WCS, en el 90 %. El 95.36 % de las secuencias generadas por el algoritmo muestreo sesgado LFT fueron mejoradas por la doble justificación. De las 1666*600 = 999600 secuencias aleatorias generadas sólo 170 no se mejoraron con la doble justificación. Dicho de otro modo, la doble justificación ha mejorado el 99.38 % de las secuencias generadas aleatoriamente. Este porcentaje es una estimación de la probabilidad de que la doble justificación mejore una secuencia activa. Obviamente, este porcentaje puede disminuir (y seguramente disminuirá) si reducimos el espacio muestral a regiones de secuencias activas de alta calidad. A lo largo de los experimentos quedará de manifiesto que la doble justificación también es capaz de mejorar soluciones de calidad.

Los datos de la Tabla 3.1 indican que la doble justificación mejora sensiblemente la calidad de las soluciones generadas por los diversos algoritmos. La mejora en la desviación respecto al CPM (UB) es de 9.15 (6.91)% en el algoritmo aleatorio, mientras que para los otros tres algoritmos está entre 4.16 (3.26)% y 4.77 (3.33)%. Esta diferencia puede ser explicada por el hecho de que la calidad inicial de las secuencias generadas aleatoriamente es muy pobre en muchas instancias y, en consecuencia, algunas de éstas pueden ser mejoradas mucho más que secuencias con una calidad más alta. Fijémonos que los algoritmos LFT_Serie y WCS_Paralelo generan una secuencia por instancia mientras que los otros dos generan 5000.

Es conveniente hacer notar que la doble justificación no aumentó el tiempo de cómputo de los algoritmos aleatorio y muestreo sesgado LFT.

Los resultados obtenidos concuerdan con los que se muestran en Tormos y Lova, 2001, que realizan experimentos similares en el conjunto j30.

La justificación en soluciones aleatorias

En este apartado queremos investigar algunos aspectos interesantes de la aplicación de la justificación. Para ello vamos a trabajar exclusivamente con soluciones (activas) aleatorias, 1666 por instancia, para tener aproximadamente 1 millón de secuencias (999600) en el total de 600 instancias. Las secuencias activas aleatorias forman una muestra no sesgada del espacio de soluciones activas, donde se mueven la gran mayoría de los heurísticos para el RCPSP, y por ello – y por su fácil y rápida obtención – las hemos escogido para estudiar algunas características de la justificación. A pesar de sus ventajas, hay que tener en cuenta sus claros inconvenientes, especialmente su baja calidad con respecto a las soluciones con las que habitualmente trabajan los heurísticos.

En todos los demás experimentos de este capítulo hemos aplicado la doble justificación, al ser la manera más sencilla y natural de aplicar la justificación, dado que se trabaja con secuencias activas. En el siguiente experimento hemos querido comprobar, entre otras cosas, si las mejoras observadas provienen de la justificación a la derecha de las soluciones o es necesaria la doble justificación para obtenerlas. También pretendemos estudiar el potencial de aplicar varias veces consecutivas la justificación, lo que denominamos justificación reiterada. Con estos fines hemos realizado la siguiente cadena para cada secuencia activa aleatoria: $S \rightarrow S^R \rightarrow (S^R)^L \rightarrow ((S^R)^L)^R \rightarrow (((S^R)^L)^R)^L \rightarrow ((((S^R)^L)^R)^L)^R \rightarrow (((((S^R)^L)^R)^L)^R)^L$. Es decir, justificamos a la derecha, después a la izquierda y esto lo repetimos 3 veces, hasta un total de 6 justificaciones. En cada etapa calculamos la mejora relativa (T(A) – T(B))/T(A), donde A es la solución de la que se parte y B a la que se llega y también contabilizamos si B mejora a A, es decir, si esa justificación ha logrado mejorar la solución. Estos cálculos los realizamos para todas las instancias, para 1666 secuencias en cada una de ellas, y realizamos las medias sobre el total de secuencias (cerca de 1 millón de soluciones). En la Tabla 3.2 se pueden ver los resultados obtenidos, en la 2ª fila aparece la media de las de mejoras relativas tras la justificación que indica la cabecera de la columna y en la 3ª, la media del número de secuencias mejoradas tras esa justificación (JDk = k-ésima justificación a la derecha, JIk = k-ésima justificación a la izquierda).

	JD1	JI1	JD2	JI2	JD3	JI3
media	12.43 %	1.14 %	0.47 %	0.25 %	0.16 %	0.11 %
%mejoradas	99.22 %	58.25 %	33.81 %	21.32 %	15.18 %	11.07 %

Tabla 3.2. Desviaciones y número de mejoras medias en la justificación reiterada.

El aspecto más importante de los resultados es que no es necesaria la doble justificación para mejorar las soluciones; de hecho, en este conjunto las mejoras más pronunciadas se producen en la primera justificación a la derecha, más que en la justificación a la izquierda posterior. Esto confirma que la técnica clave es la justificación y no la doble justificación, aunque sea esta última la que empleemos en los algoritmos por simplicidad. En cualquier caso, la justificación a la izquierda también mejora la solución obtenida por la justificación a la derecha (en un 58% de los casos), por lo que la combinación de ambas potencia la capacidad de mejora. La enorme diferencia entre las medias de las desviaciones (casi 11 veces más grande una media que otra, mientras que el porcentaje de mejoras no es ni el doble del otro) es sobre todo achacable a que las soluciones aleatorias son de baja calidad en general, por lo que su duración puede rebajarse considerablemente. Otra conclusión interesante que se puede extraer de la Tabla 3.2 es que las mejoras no acaban tras la doble justificación; si se aplican más justificaciones se puede seguir aumentando la calidad. Claramente esta reducción de la duración (y del número de soluciones en que se produce) va decreciendo a medida que justificamos más veces y, quizás, no compense realizar más de una doble justificación en la práctica (aunque éste es un campo interesante para estudiar). Un comentario en este mismo sentido fue realizado en Tormos y Lova, 2001. A pesar de esto, es destacable que un 11% de las soluciones (más de 100000 del millón) que ya han pasado por tres justificaciones a la derecha y dos a la izquierda se hayan mejorado al justificarlas de nuevo a la izquierda (%mejoradas tras JI3). Por supuesto, esto no implica que esas secuencias hayan sido mejoradas en cada una de las 6 justificaciones. La media de mejora relativa considerando únicamente las secuencias que se han mejorado en la 3ª justificación a la derecha (izquierda) es de aproximadamente un 1% (la media que aparece en la tabla también tiene en cuenta las secuencias no mejoradas).

Cada secuencia se puede justificar de diferentes formas a la derecha (o a la izquierda si no es activa), dependiendo de cómo se desempate entre las actividades que finalizan (comienzan) al mismo tiempo. El siguiente experimento persigue analizar la diversidad existente en la justificación de una misma secuencia, es decir, si justificando aleatoriamente de varias formas se pueden obtener resultados (duraciones) distintos. Para ello hemos justificado a la derecha aleatoriamente 10 veces cada una de las 1666 soluciones de cada instancia y hemos calculado la mejora relativa tras 1 justificación a la derecha (1ª JD) y la media de las mejoras relativas, la máxima y la desviación típica de esas mejoras tras 10 justificaciones a la derecha (10 JD); después hemos calculado las medias de esos números para todas las secuencias de todas las instancias. Hemos calculado los mismos valores pero sobre la

justificación a la izquierda (1ª JI y 10 JI), aplicada sobre la justificación a la derecha de las soluciones iniciales.

	media	máximo	desviación típica
1ª JD	12.4359 %	-	-
10 JD	12.4357 %	12.8790%	0.3348 %
1ª JI	1.1462 %	-	-
10 JI	1.1463 %	1.3820 %	0.1826 %

Tabla 3.3. Desviación media tras 1 justificación y media, máxima y desviación típica tras 10 justificaciones.

Según estos datos existe una ligera diversidad en la justificación, pero es pequeña. Esto se concluye a partir de que la desviación típica tras 10 justificaciones es pequeña pero no nula. A pesar de esto, justificar varias veces sí produce mejores resultados, aunque las mejoras que se pueden extraer de ello son muy pequeñas, al menos en el caso de la justificación a la derecha. Es interesante remarcar que la justificación a la izquierda es bastante más diversa que la justificación a la derecha, dado que la desviación típica tras 10 justificaciones a la izquierda es unas 6 veces más pequeña que la media, mientras que en la justificación a la derecha el factor es mayor de 35. Además, el máximo tras 10 justificaciones a la izquierda mejora en un 20 % lo obtenido en una única justificación, mientras que en el caso de la justificación a la derecha es un 3.5%, a pesar de que las soluciones de partida son de calidad netamente superior (un 12 % menores en media) en el caso de la justificación a la izquierda. Esto parece indicar que los desempates son más cruciales en las soluciones de calidad, al menos en las de cierta calidad. Los datos sugieren que puede ser beneficioso buscar una manera de desempatar diferente de la aleatoria, dado que quizás sea posible obtener mejores soluciones.

2.2. Segundo conjunto de experimentos

En esta sección queremos ampliar el experimento a algoritmos no tan sencillos como los de la sección anterior pero que no se acercan a la calidad de los mejores heurísticos del RCPSP. Para ello hemos diseñado específicamente para este experimento 16 algoritmos basados en poblaciones. Todos ellos comparten el mismo procedimiento para generar la población inicial y, básicamente, el mismo esquema evolutivo que está reducido a la mínima expresión. En cada iteración, una cierta

función actúa sobre secuencias de la población actual para generar una nueva población. Estas funciones (2) se construyen a partir de operadores binarios que combinan las características de dos secuencias para generar una nueva. La primera función, función relinking, genera una serie de secuencias C_1, C_2, ..., C_q entre dos secuencias dadas. La segunda función, función first_neighbour, se define a partir de la primera: al aplicarla a un par de secuencias genera solamente la primera secuencia C_1 que generaría la función relinking. A partir de estas dos funciones hemos definido cuatro esquemas algorítmicos. Los tres primeros son prácticamente el mismo esquema. Se diferencian básicamente en que el primero utiliza la función relinking, el segundo aplica la función first_neighbour a parejas de la población actual y el tercero aplica la función first_neighbour sobre una pareja de secuencias en la que la primera secuencia pertenece a la población actual mientras que la segunda es generada aleatoriamente. Para ofrecer una mayor diversidad, hemos diseñado un cuarto esquema algorítmico que se obtiene del tercero cambiando la manera de renovar la población en cada iteración.

Los 16 algoritmos de este segundo conjunto se obtienen al combinar de todas las maneras posibles los 4 esquemas algorítmicos con los 4 operadores definidos. La diversidad de operadores y de funciones induce la variedad de los algoritmos.

2.2.1. Operadores binarios

Un operador binario es una función que a cada par de secuencias A y B asigna un secuencia C = FB(A,B). Definiremos 4 operadores binarios, o más correctamente, cuatro familias de operadores binarios, cada una de las cuales dependiente de un parámetro α. Variando adecuadamente el valor del parámetro α se pueden obtener series de secuencias que comiencen pareciéndose a A y que, progresivamente, se parezcan cada vez menos a A y más a B. Para definir los operadores emplearemos los vectores de prioridad. Supongamos que γ_A y γ_B son vectores de prioridad que codifican A y B respectivamente.

Operador suma

El operador suma de parámetro real α, $0 \leq \alpha \leq 1$, es un operador binario que dadas dos secuencias A y B construye la secuencia $C = S[\alpha\, \gamma_A + (1-\alpha)\, \gamma_B]$. La secuencia C está en el "segmento" que une las secuencias A y B siendo más parecida a A cuanto mayor sea α. De hecho con $\alpha = 0$ se obtiene B y con $\alpha = 1$, A. Fijémonos que este operador es la generalización del empleado en HIAC.

Operador de cambios

El operador de cambios de parámetro entero α, $0 \leq \alpha \leq |\mathbf{V}|$, es un operador binario que dadas dos secuencias A y B construye una secuencia C aplicando el siguiente procedimiento:

Figura 3.1. Operador de cambios

1. $\gamma_C(i) = \gamma_A(i)\ \forall i \in \mathbf{V}$.
2. Cand = **V**.
3. Desde i = 1 hasta α, hacer:
   3.1. Escoger aleatoriamente una actividad $i \in$ Cand.
   3.2. Hacer $\gamma_C(i) = \gamma_B(i)$.
   3.3. Cand = Cand \ {i}, Escogidas = Escogidas $\cup$ {i}.
4. Devolver $C = S(\gamma_C)$.

Fijémonos que se cumple $\gamma_C(i) = \gamma_B(i)\ \forall i \in$ Escogidas y $\gamma_C(i) = \gamma_A(i)\ \forall i \in \mathbf{V}\backslash$Escogidas. El parámetro α indica el número de componentes del vector γ_A que se cambian por componentes de γ_B. Dependiendo de que α sea más grande o más pequeño, γ_C se asemejará más a γ_B o a γ_A y, en general, ello implicará que C se parezca más a B o a A. En particular, con $\alpha = 0$ se tiene C = A y con $\alpha = |\mathbf{V}|$ se tiene C = B. La secuencia C obtenida depende de la aleatoriedad.

Operador porcentaje

El operador porcentaje de parámetro real α, $0 \leq \alpha \leq 1$, es un operador binario que dadas dos secuencias A y B construye una secuencia C aplicando Serie del siguiente modo. En cada etapa, se calcula el conjunto de actividades elegibles y se genera un número aleatorio p entre 0 y 1. Si p es menor o igual que α, se secuencia la actividad elegible con menor orden según γ_A; en caso contrario, se secuencia la actividad elegible con menor orden según γ_B. Así pues, en media, el 100α % de las veces escogemos una actividad según γ_A y el resto de las veces según γ_B. Obviamente $\alpha = 0$ conduce a B y $\alpha = 1$ a A. Este operador guarda muchas similitudes con el procedimiento β-Biased.

Operador ventana

El operador ventana de parámetro α, donde α es un número entero positivo, es un operador binario que dadas dos secuencias A y B construye una secuencia C

aplicando Serie del siguiente modo. En cada etapa, se ordenan las actividades elegibles según γ_A. Si el número de actividades elegibles es mayor que α se seleccionan las α primeras; si no, se seleccionan todas. Se secuencia la actividad del conjunto seleccionado con menor orden según γ_B. Así pues, la secuencia A determina las α actividades más prometedoras (veta las que le parecen inadmisibles) para ser secuenciadas en la etapa actual mientras que la secuencia B escoge de entre ellas la actividad que considera más adecuada. Un valor de α = 1 lleva a A, mientras que α = n, a B.

En el resto del capítulo, FB denotará de forma genérica a un operador binario y FB(A,B,α) simbolizará el resultado de aplicar uno de los anteriores operadores con parámetro α a las secuencias A y B.

2.2.2. Funciones empleadas

A partir de cada uno de los operadores descritos, se pueden definir funciones que dadas dos secuencias A y B construyan q secuencias C_1, C_2, ..., C_q.

Función relinking

La función relinking construye q secuencias de manera que se vayan pareciendo cada vez más a B y menos a A. El nombre proviene del reencadenamiento de trayectorias ('path relinking'). La definición de la función relinking depende del operador binario que se utilice.

Con el operador suma

Función_relinking(FB,A,B,q) = $\{C_j = FB(A,B,j/(q+1)), j = q,...,1\}$ donde FB es el operador suma.

Con el operador de cambio

Función_relinking(FB,A,B,q) = $\{C_j = FB(A,B, \lfloor n/(q+1) \rfloor + j), j = 1,...,q\}$ donde FB es el operador de cambio y donde las secuencias C_j se obtienen de la siguiente manera:

Figura 3.2. Obtención de las secuencias en el operador de cambios

1. $\gamma_C(i) = \gamma_A(i)\ \forall i \in V$. Cand = **V**.
2. Desde j = 1 hasta q, hacer:
   - 2.1. Desde h = 1 hasta $\lfloor n/(q+1) \rfloor$, hacer:
     - 2.1.1. Escoger aleatoriamente una actividad $i \in$ Cand.
     - 2.1.2. Hacer $\gamma_C(i) = \gamma_B(i)$.
     - 2.1.3. Cand = Cand \ {i}, Escogidas = Escogidas $\cup$ {i}.
   - 2.2. Obtener $C_j = S(\gamma_C)$.

Démonos cuenta que cuando calculamos C_j hemos obligado a que el vector γ_C tenga $\lfloor n/(q+1) \rfloor * j$ prioridades en común con γ_B. En cada iteración, se mantienen las prioridades comunes con γ_B de la iteración anterior y se añaden $\lfloor n/(q+1) \rfloor$ nuevos valores de γ_B. Notar que se puede fijar el número de cambios entre una secuencia C_j y la siguiente C_{j+1}, y calcular el q a partir de ese número.

Con el operador porcentaje

Función_relinking(FB,A,B,q) = {C_j = FB(A,B,j/(q+1)), j = q,...,1} donde FB es el operador porcentaje.

Con el operador ventana

Función_relinking(FB,A,B,q) = {C_1, C_2, ..., C_q} donde FB es el operador ventana, q es un número entero par y donde las secuencias C_j se obtienen de la siguiente manera:

C_j = FB(A,B, j +1), j = 1,...,q/2 and C_j = FB(B,A, q-j +2), j = q/2 + 1,...,q

Función first_neighbour

La salida de la función first_neighbour consiste en la secuencia generada por la función relinking que está más cercana a A. Es decir:

Función_first_neighbour(FB,A,B,q) = {$C_1 \in$ Función_relinking(FB,A,B,q)}.

2.2.3. Población inicial

La población inicial va a ser la empleada en CARA e HIAC, i.e., la dada por INITIAL_SET_1.

2.2.4. Esquemas algorítmicos

Esquema algorítmico 1

Figura 3.3. Esquema algorítmico 1(POPsize,niter,F,q)

1. POP = INITIAL_SET_1(POPsize) = $\{C_1, ..., C_{POPsize}\}$.
2. Desde i = 1 hasta niter, hacer:
   2.1. POP' = ∅.
   2.2. Desde j = 1 hasta POPsize, hacer:
      2.2.1. Desde h = j+1 hasta POPsize, hacer
         POP' = POP' ∪ Función_relinking(F,C_j,C_h,q).
   2.3. POP = {mejores POPsize soluciones de POP'}.

Esquema algorítmico 2

Figura 3.4. Esquema algorítmico 2(POPsize,niter,F,q)

1. POP = INITIAL_SET_1(POPsize) = $\{C_1, ..., C_{POPsize}\}$.
2. Desde i = 1 hasta niter, hacer:
   2.1. POP' = ∅.
   2.2. Desde j = 1 hasta POPsize, hacer:
      2.2.1. Desde h = 1 hasta POPsize, $h \neq j$, hacer
         POP' = POP' ∪ Función_first_neighbour(F,C_j,C_h,q).
   2.3. POP = {mejores POPsize soluciones de POP'}.

Esquema algorítmico 3

Se obtiene del esquema algorítmico 2 cambiando la etapa 2.2.1 por la siguiente:

2.2.1 Desde h = 1 hasta POPsize, $h \neq j$, hacer
   2.2.1.1. Generar aleatoriamente una secuencia S_j.
   2.2.1.2. POP' = POP' ∪ Función_first_neighbour(F,C_j,S_j,q).

Figura 3.5. Esquema algorítmico 4(POPsize,nsol,niter,q)

1. POP = INITIAL_SET_1(POPsize) = {C_1, ..., $C_{POPsize}$}.
2. Desde i = 1 hasta niter, hacer:
   2.1. Desde j = 1 hasta POPsize, hacer:
      2.1.1. $POP_j = \varnothing$.
      2.1.2. Generar aleatoriamente una secuencia S_j.
      2.1.3. Desde h = 1 hasta nsol, hacer
      $POP_j = POP_j \cup$ Función_first_neighbour(F,C_j,S_j,q).
   2.2. Reemplazar C_j de POP por la mejor solución de POP_j si esta última es de menor longitud.

2.2.5. Resultados computacionales

Puesto que el objetivo de esta sección es analizar el efecto de incluir la doble justificación en algoritmos relativamente sencillos que produzcan soluciones de relativa calidad pero que no se acerquen demasiado a los mejores algoritmos, no hemos dedicado mucho esfuerzo en optimizar la velocidad de los algoritmos ni en seleccionar los valores de los parámetros. El valor o rango de valores posibles de los parámetros lo(s) hemos fijado atendiendo a diversas motivaciones.

El valor de q es el mismo para todos los algoritmos que utilizan el mismo operador: q = 15 para los que utilizan el operador suma, por similitud con HIAC; q = 8 para los que emplean el operador ventana, por similitud con CARA (de este modo se calcula window = 2, 3, 4 y 5 para A y para B); q = 11 para los que utilizan el operador cambios, de esta manera, cada una de las secuencias generadas por la función relinking se diferencia de la anterior en 10 cambios. Por último, hemos fijado q = 9 para todos los algoritmos que utilizan el operador porcentaje. De esta manera, el "porcentaje de similitud" de cada secuencia generada por la función reencadenamiento con la secuencia A disminuye un 10 % cada vez.

El conjunto de valores posibles del parámetro POPsize se ha limitado a {5, 10, 15, 20, 25, 30}. El parámetro nsol sólo se utiliza en el esquema 4 y puede variar dentro del conjunto {5, 10, 15, 20} en los algoritmos que utilizan el operador porcentaje y el de cambio. En los operadores suma y ventana FB(A,B,α) produce una única secuencia (salvo empates en el caso de la suma) para los mismos A, B y α, por lo que nsol se ha fijado a 1.

Combinando los 4 esquemas con los 4 operadores obtenemos 16 algoritmos distintos que pueden considerarse con o sin DJ, obteniendo finalmente 32 algoritmos distintos.

Mediante unas pruebas preliminares hemos fijado para cada uno de esos 32 algoritmos la "mejor" combinación de los valores de los parámetros POPsize y nsol dentro de los límites. Una vez fijados los parámetros anteriores, niter ha sido calculado para que el total de secuencias generadas sea menor o igual que 5000.

La Tabla 3.4 resume la calidad de las soluciones obtenidas para el conjunto j120. La primera columna indica el operador usado por el esquema algorítmico mencionado en la primera fila. Cada casilla de la tabla se refiere al algoritmo formado por la combinación del esquema indicado por su columna con el operador asociado a su fila. Cada casilla contiene cuatro números, dos en estilo de fuente normal y dos en negrita. Los dos en estilo normal se refieren al algoritmo sin doble justificación, los otros al algoritmo con doble justificación. De los dos números del mismo estilo, el primero indica desv_CPM mientras que el segundo, entre paréntesis, informa sobre desv_UB.

	Esquema 1	Esquema 2	Esquema 3	Esquema 4
suma	38.96(5.55)/ **35.18(2.95)**	39.78(6.19)/ **35.46(3.16)**	38.78(5.46)/ **34.35(2.44)**	39.30(5.81)/ **35.37(3.11)**
cambios	38.05(4.93)/ **34.50(2.52)**	37.80(4.80)/ **34.02(2.22)**	38.71(5.36)/ **34.92(2.76)**	38.44(5.22)/ **34.59(2.55)**
porcentaje	38.12(5.00)/ **34.60(2.61)**	38.16(5.04)/ **34.27(2.40)**	38.60(5.30)/ **34.78(2.72)**	38.04(4.91)/ **34.13(2.29)**
ventana	38.87(5.47)/ **34.46(2.47)**	39.11(5.68)/ **34.08(2.25)**	39.27(5.79)/ **35.49(3.11)**	39.30(5.82)/ **35.31(2.99)**

Tabla 3.4. Resultados computacionales para el segundo conjunto de experimentos en j120.

Podemos observar que todos los algoritmos han mejorado al aplicarles la doble justificación. Sin la justificación, la desviación respecto del CPM (UB) varía desde un valor máximo de 39.78 (6.19) % a un valor mínimo de 37.80 (4.80) %. Mediante la incorporación de la doble justificación, este rango se reduce a [34.02,35.59] ([2.22,3.16]) %. La mejora de la desviación en porcentaje respecto del CPM (UB) al incorporar DJ varía desde un mínimo de 3.52 (2.39)% a un máximo de 5.03 (3.43)%.

Es interesante comparar estos resultados con los obtenidos por los mejores heurísticos conocidos con limitación de 5000 secuencias (Möhring et al. no tiene esa limitación, pero genera 3675 secuencias de media. Tormos y Lova generan 5000 secuencias contabilizando una secuencia por cada DJ; de acuerdo con nuestra manera de contar secuencias, su límite es de 7500). La Tabla 3.5 muestra la desviación media respecto del CPM obtenida por diversos autores en j120 con una cota superior de 5000 secuencias. Podemos observar que todos los algoritmos sin

justificar de la tabla 3.4 generan soluciones de peor calidad que todos los algoritmos de la tabla 3.5. Sin embargo, cuando se introduce la doble justificación, 13 de los 16 algoritmos producen mejor calidad de solución que cualquier algoritmo de la tabla 3.5. Uno de los otros 3 tiene una calidad casi idéntica a Hartmann (2) (el mejor de la tabla) y los otros dos al algoritmo de las hormigas de Merkle et al. con colonias adelante y atrás (el segundo mejor de la tabla). Además la diferencia entre el mejor algoritmo de la Tabla 3.4 y Hartmann (2) es de un 1.33 (≈0.77) % respecto del CPM (UB).

Tipo algoritmo	Autor(es)	desv_CPM
Algoritmo genético	Hartmann (2)	35.35
Optimización 'ant colony' con colonias adelante y atrás	Merkle et al.	35.43
Muestreo aleatorio + adelante-atrás	Tormos y Lova	35.62
Heurístico Lagrangiano	Möhring et al.	36.2
Algoritmo genético	Alcaraz y Maroto	36.57
Optimización 'ant colony'	Merkle et al.	36.65
Algoritmo genético	Hartmann (1)	36.7
Propagación restricciones	Dorndorf et al.	37.1

Tabla 3.5. Resultados computacionales de los mejores algoritmos con 5000 secuencias en j120.

Comentario aparte merece el resultado del algoritmo de Tormos y Lova, dado que emplea la doble justificación y no es considerablemente mejor que el resto. La razón de esto es que el algoritmo original, antes de aplicar DJ, es un muestreo aleatorio; generalmente, los muestreos aleatorios ofrecen una calidad sensiblemente inferior a los metaheurísticos. La doble justificación sí mejora el algoritmo sensiblemente, pero esa mejora sirve únicamente para equipararlo con los mejores metaheurísticos. Otro factor en contra de este tipo de muestreos no autoregulables es que las mejoras obtenidas por DJ en las primeras iteraciones no se aprovechan, dado que en cada iteración se construyen nuevas secuencias sin tener en cuenta las anteriores.

Recordemos que algunos de estos algoritmos emplean un tiempo considerable: Möhring et al. (65 seg. en un Sun Ultra 2, 200 MHz), Merkle et al. (25 seg. en un PC 500 MHz) y Dorndorf et al. (205 seg. en un PC 200). El resto de algoritmos de la literatura comentados en el capítulo 2 (generando más de 5000 secuencias) tampoco son capaces de acercarse al 34.02% de desv_CPM del mejor de los algoritmos de la

Tabla 3.4, exceptuando Merkle 2 (33.68%) y 3 (32.97%), el primero de los cuales emplea 25 minutos y el segundo utiliza un tiempo mucho mayor pero sin determinar.

Es conveniente hacer notar que en ningún caso el tiempo de cómputo de un algoritmo con doble justificación fue superior al tiempo de cómputo del mismo algoritmo sin justificación. A pesar de no haber optimizado los tiempos de los procedimientos, dado que el objetivo no eran los heurísticos por sí mismos sino la diferencia de calidad entre ellos, podemos aportar como dato que varios de los algoritmos con doble justificación emplean entre 2 y 3 segundos de media.

2.3. Aplicación de la doble justificación a una función de mejora

Todos los algoritmos del apartado 2.2. tienen en común que el mecanismo para obtener soluciones es la combinación de secuencias. En este apartado construiremos una función simple F de mejora que no emplee la combinación de soluciones ni la evolución de poblaciones y veremos que, al aplicarle la doble justificación, se mejora sustancialmente la calidad obtenida.

2.3.1. La función de mejora

Definimos la siguiente función SA.

Figura 3.6. Función SA(Sinicial,niter,nsol,μ,ϕ,r)

1. S = Sinicial. Temp = $-(\mu * T(S))/\log(\phi)$.
2. Desde i = 1 hasta niter, hacer:
   2.1. Desde j = 1 hasta nsol, hacer
      2.1.1. S' = Biased(S,β).
      2.1.2. Delta = T(S') – T(S).
      2.1.3. Sea t un número aleatorio en (0,1)
      2.1.4. Si ($t < e^{-Delta/Temp}$) S = S'.
   2.2. Temp = Temp * r.
3. Devolver la mejor solución encontrada.

SA es un algoritmo de temple simulado (cf. apartado 7.2 capítulo 1) bastante sencillo, la temperatura disminuye de forma lineal y el número de soluciones que se visita en cada temperatura es constante. El entorno de una solución S está formado por las secuencias accesibles mediante Biased(S,β). Es decir, en principio se pueden obtener todas las secuencias activas, pero la mayor probabilidad se centra en las secuencias

cuyos vectores de prioridad no se diferencien excesivamente (si β es grande) de γ(S). El β escogido ha sido 1-5/n.

La forma natural de añadir la doble justificación consiste en aplicársela a cada solución obtenida mediante Biased(S,β), y considerar la secuencia activa resultante como posible nueva solución. Es decir, insertaríamos un paso 2.1.1.b con S' = DJ(S'). Además, justificaríamos a derecha e izquierda la secuencia inicial antes de entrar en el paso 1.

La función resultante SA+DJ genera el triple de secuencias que la original por cada iteración, pero el límite sobre las dos será de 5000 secuencias, como en las anteriores comparaciones.

2.3.2. Resultados computacionales

Los resultados van a basarse en el siguiente algoritmo.

Figura 3.7. Algoritmo SAC(POPsize,niter,nsol,μ,ϕ,r)

1. POP = INITIAL_SET_1(POPsize).
2. POP' = F(POP,niter,nsol,μ,ϕ,r).
3. Devolver la mejor solución de POP'.

Donde F es SA o SA + DJ. Por abuso de notación llamaremos SA y SA + DJ a estos algoritmos, que simplemente aplican la F correspondiente a los mejores POPsize individuos de la población inicial con la que estamos trabajando habitualmente. Mediante unas pruebas preliminares hemos fijado la "mejor" combinación de los valores de los parámetros POPsize, nsol, ϕ y r, dentro de los límites {1, 2, 3, 5}, {1, 5}, {0.025, 0.05, 0.1} y {0.85, 0.9, 0.95} respectivamente; μ se fija a 0.01, porque la variación de Temp se puede obtener cambiando ϕ. Una vez fijados los parámetros anteriores, niter ha sido calculado para que el total de secuencias generadas sea menor o igual que 5000. La Tabla 3.6 muestra las medidas habituales para SA y SA + DJ.

	Σ	desv_UB	desv_CPM	med_CPU
SA	78331	5.07	38.03	3.16
SA + DJ	75813	1.98	33.59	2.21

Tabla 3.6. Resultados computacionales para SA, con y sin justificación.

Los resultados obtenidos demuestran lo pretendido, que las mejoras producidas por la justificación no se deben a que el algoritmo sea evolutivo o trabaje con poblaciones. La doble justificación mejora en un 4.44 (3.09)% la desviación media respecto del CPM (UB). Además, la desviación media respecto del CPM es sensiblemente inferior a la de los algoritmos de la Tabla 3.5, así como a la del resto de algoritmos considerados en el capítulo 2, a excepción de Merkle et al. (3), cuyo límite de secuencias es mucho mayor que en Merkle et al. (2), de 25 minutos de tiempo medio en un PC a 500 MHz. Para ello, SA + DJ necesita únicamente 2.21 segundos de media, un tiempo que en general parece menor a la mayoría de los algoritmos de la Tabla 3.5, incluso teniendo en cuenta los ordenadores empleados. También es casi un segundo menor que el de SA, el algoritmo original, lo que reafirma la utilidad de añadir la justificación a los algoritmos, incluso para rebajar en algunos casos el tiempo empleado. En el apartado 3 comentaremos con más detalle esta afirmación.

Es interesante remarcar la "sencillez" de esta función. Sería interesante comprobar si se pueden mejorar los resultados incorporando algunas de las posibilidades de los algoritmos de temple simulado, como podrían ser: una disminución no lineal de la temperatura, visitar un número no constante de soluciones en cada iteración, aumentar la temperatura tras un cierto número de iteraciones sin mejorar ('reheat'), etc.

2.4. La doble justificación en un algoritmo de calidad

En esta sección queremos analizar el efecto de aplicar la doble justificación a un algoritmo que proporciona soluciones de calidad previamente a incorporarle la justificación. Para ello hemos seleccionado el algoritmo de Hartmann, un algoritmo sencillo conceptualmente, rápido y uno de los mejores algoritmos de entre los que no utilizan la justificación.

En el experimento hemos utilizado una programación del algoritmo Hartmann (1) realizada por nosotros mismos y que denotaremos Hartmann (1'). Esta adaptación no obtiene exactamente los mismos resultados que el algoritmo original porque ambos algoritmos dependen de la aleatoriedad. Primero, hemos ejecutado el programa con un límite de 5000 secuencias; después, con un límite de 10000 secuencias y finalmente, con un límite total de 5000 secuencias pero habiendo incorporado primero la doble justificación (Hartmann + DJ). Realizamos unas pruebas preliminares para determinar los tamaños de las poblaciones más apropiados. El tamaño de la población en Hartmann (1') 5000, 10000 y Hartmann + DJ es 100, 200 y 50, respectivamente. Los resultados aparecen en la tabla 3.7.

	Σ	desv_UB	desv_CPM	med_CPU
Hartmann (1') 5000	77746	4.41	37.00	1.55
Hartmann (1') 10000	77283	3.81	36.18	3.08
Hartmann + DJ 5000	75616	1.74	33.24	1.60

Tabla 3.7. Resultados computacionales para Hartmann, con y sin justificación.

Podemos observar que, en el caso de un límite superior de 5000 secuencias, la doble justificación mejora la calidad de las soluciones en un 3.76 (2.67) % respecto del CPM (UB) aunque con un ligero incremento de 0.05 segundos en el tiempo de cómputo. Claramente el aumento de calidad no se debe a un aumento del tiempo de cómputo, pero este hecho es más evidente aún si la comparación se realiza con Hartmann sin justificación con un límite de 10000 secuencias. Hartmann + DJ 5000 mejora a Hartmann (1') 10000 tanto en calidad como en tiempo medio.

Es importante resaltar que el algoritmo Hartmann (1) ocupa el cuarto lugar en la Tabla 3.5 pero que, sin embargo, el nuevo algoritmo Hartmann + DJ que se obtiene del anterior con sólo añadirle la doble justificación, mejora notablemente a todos los mejores heurísticos de la Tabla 3.5. También es superior en calidad (y tiempo) al resto de algoritmos vistos en este capítulo (que calculan 5000 secuencias).

Hasta ahora hemos realizado la (doble) justificación sobre todas las secuencias proporcionadas por los algoritmos; así, en el algoritmo genético de Hartmann aplicamos DJ a cada secuencia obtenida al combinar dos listas de actividades, así como a las soluciones iniciales. Hemos observado en diferentes experimentos que proceder de este modo mejora los algoritmos, pero no sería descartable que las mejoras provinieran de la aplicación de la justificación a las primeras secuencias – de baja calidad – y que a partir de un cierto número secuencias (o de una cierta calidad) ésta no rebaje las duraciones de las soluciones e, incluso, sea más efectivo no emplearla. Recordemos que al justificar calculamos tres veces menos secuencias con las técnicas del algoritmo original, por lo que este último puede realizar tres veces más iteraciones. Para comprobar este extremo, y dado que el algoritmo Hartmann + DJ es el que mejores resultados obtiene, vamos a utilizarlo de la siguiente manera: dado un k entero, compararemos algoritmos que justifiquen las primeras k/3 secuencias y el resto no las justifique. Llamamos Hartmann + DJ (k) al algoritmo que aplica la doble justificación a las secuencias encontradas con la población inicial y el crossover hasta que en total se hayan generado k secuencias, las siguientes 5000 – k secuencias las obtiene como el algoritmo original, sin justificar ninguna. En particular Hartmann + DJ (0) es el algoritmo original y con k = 5000 tenemos el algoritmo con la doble justificación de la Tabla 3.7. En la Tabla 3.8 se puede observar los resultados

obtenidos por varios de estos algoritmos. Recordemos que todos calculan aproximadamente el mismo número de secuencias, acotado superiormente por 5000, simplemente se diferencian en el número de secuencias a las que aplican DJ (≈ las k/3 primeras).

k	desv_UB	desv_CPM
0	4.41	37.00
Sólo justificando la población inicial	3.78	36.42
500	3.43	35.84
1000	2.99	35.20
2000	2.41	34.32
3000	2.06	33.77
4000	1.88	33.47
5000	1.74	33.24

Tabla 3.8. Comparación de Hartmann + DJ (k) para diferentes k's.

Como se puede observar, cuantas más secuencias se justifiquen, mejores resultados se obtienen. Ya sólo con justificar la población inicial (y después aplicar el genético original) se mejoran los resultados de Hartmann. Esta mejora aumenta paulatinamente conforme se justifican más secuencias, hasta el punto de que el mejor algoritmo es Hartmann + DJ, cuando todas las secuencias se justifican. Si tuviéramos que extrapolar estos resultados, la conclusión debería ser que la aplicación de la justificación es útil aunque se emplee exclusivamente al inicio del algoritmo, pero es más poderosa si se utiliza a lo largo de todo el procedimiento. A pesar de esto, sí se aprecia una clara ralentización en las mejoras producidas al aumentar el número de secuencias que se justifican, poniendo de manifiesto que la aplicación de la justificación a soluciones de baja o media calidad causa un impacto mayor que cuando se trabaja con soluciones de relativamente buena calidad.

Sin embargo, la justificación es capaz de mejorar soluciones de calidad. Esta importante propiedad se ve reforzada con estas pruebas. Cuando se llevan 3000 ó 4000 secuencias las soluciones que produce el algoritmo genético ya poseen una cierta calidad. Sin embargo, la (doble) justificación es capaz de mejorarlas lo suficiente como para que Hartmann + DJ obtenga mejores resultados que Hartmann + DJ(3000) y Hartmann + DJ(4000). Si DJ no consiguiera mejorarlas, estos dos algoritmos llegarían al mismo nivel de calidad, y seguramente mayor, porque al no justificar

realizan más iteraciones y combinaciones de soluciones. Para completar este análisis, hemos calculado el porcentaje de soluciones mejoradas por la doble justificación en Hartmann + DJ, escogiendo para ello las instancias donde se realizan todas las iteraciones previstas a priori (en algunas se puede detener la ejecución antes porque se ha encontrado una solución con la duración del CPM y, por tanto, óptima). Ese porcentaje es del 57.96%, y hay que tener en cuenta que, en algunas instancias fáciles, a partir de cierta iteración, todas o casi todas las soluciones calculadas son óptimas y, por tanto, imposibles de mejorar. También hay que tener presente que las secuencias calculadas son, presumiblemente, bastante similares en las últimas iteraciones, debido a la evolución del algoritmo genético, y que son combinaciones de soluciones obtenidas mediante la aplicación de DJ a otras soluciones. A pesar de todo esto, casi el 60% de secuencias se mejoran con DJ y esta mejora es lo suficientemente grande como para que, incluso en las últimas iteraciones, sea conveniente emplear DJ.

Un caso atípico

En este apartado hemos visto lo sencillo que es incorporar la (doble) justificación al flujo de un algoritmo no diseñado para este propósito. Esto es lo que ocurre en una variedad importante de heurísticos; por poner dos ejemplos más, sería trivial añadir la doble justificación al heurístico lagrangiano de Möhring et al. y al basado en el metaheurístico de las hormigas de Merkle et. al. Sin embargo, también existen procedimientos que por sus características no admiten de una forma tan sencilla ese cambio. Un ejemplo interesante de este extremo es Hartmann (2). Su diferencia básica con Hartmann (1) es que una parte importante de sus secuencias (más o menos grande dependiendo de las instancias) son sin retraso. Si estas soluciones se justificaran a derecha y a izquierda perderían su condición, porque sólo se aseguraría que son activas. Por otra parte, no justificarlas significaría que serían de bastante peor calidad que el resto, por lo que no sobrevivirían al proceso de selección (sin tener en cuenta la duda importante de si Paralelo es capaz de llegar a la nueva y bastante mejor calidad media). En cualquier caso, el rasgo distintivo del algoritmo se perdería si se añadiera la doble justificación y sólo se obtendría algo muy similar a Hartmann +DJ.

2.5. El papel de la justificación en CARA e HIAC

Los resultados obtenidos en este capítulo plantean la duda de hasta qué punto es importante la justificación en los algoritmos CARA e HIAC. El aspecto clave es el porcentaje de soluciones que se justifican a derecha o a izquierda, que en estos dos

algoritmos es muy pequeño; algunos cálculos realizados sugieren que ese porcentaje está entre 1/100 y 1/500, sino es más pequeño. Las mejoras derivadas de la justificación en ese caso son obviamente mucho menores que cuando se justifica el 100% de las veces. Si en el mejor algoritmo obtenido en este capítulo, Hartmann+ DJ, se reduce a una de cada 100 las soluciones que se justifican a derecha y a izquierda (que se correspondería con un 0.02%), se obtiene una desviación media respecto del CPM del 36.08%, muy lejos de las de CARA e HIAC.

El uso de la justificación en CARA e HIAC se ha limitado al indispensable para poder aplicar el mecanismo oscilatorio. Ello ha permitido diferenciar la bondad de CARA e HIAC de la obtenible al añadir la (doble) justificación. Una vez comprobada la efectividad de CARA e HIAC se puede pensar en combinar en el futuro los movimientos de CARA, y especialmente los de HIAC, con la justificación.

Se podría pensar que los buenos resultados de CARA e HIAC (y HGA del capítulo 5) son debidos exclusivamente a la inclusión de la justificación o de métodos adelante-atrás. Ejemplos de que esto no es cierto son los algoritmos de Alcaraz y Maroto, Merkle et al. (4) y Tormos y Lova. Estos algoritmos emplean (alguna de) esas técnicas y, a pesar de ello, la calidad de las soluciones está lejana a la de nuestros algoritmos.

3. EL TIEMPO DE CÓMPUTO DE LA JUSTIFICACIÓN

Preliminares

Antes de calcular el tiempo necesario para llevar a cabo una justificación vamos a demostrar un resultado que nos facilitará la tarea.

Para justificar a la derecha (izquierda) una actividad i dentro de una secuencia S es necesario calcular el instante de tiempo más tardío (temprano) donde se puede secuenciar, teniendo en cuenta los recursos empleados por el resto de actividades, excepto por i (además de respetar las relaciones de precedencia). Está clara, por tanto, la relación estrecha existente entre secuenciar y justificar: vamos a demostrar que justificar es exactamente lo mismo que secuenciar, aunque de una cierta manera.

Recordemos que una AL representación de una secuencia S es una lista de actividades λ que cumple $s_i < s_j \rightarrow orden(i,\lambda) < orden(j,\lambda)$. De un modo simétrico se puede denominar **AL representación inversa de S** a una permutación de las n actividades λ que cumple $f_i < f_j \rightarrow orden(i,\lambda) > orden(j,\lambda)$, es decir, a mayor final, menor orden. Fijémonos en que λ es una lista de actividades para la red inversa.

Definición: Secuenciar por finales (inicios)

Sea S una secuencia y λ una AL representación (inversa) de S. **Secuenciar por inicios (finales)** consiste en aplicar el método Serie a λ, secuenciando cada actividad lo más pronto (tarde) posible, fijando $s_1 = f_1 = 0$ ($f_n = s_n = T(S)$). Llamaremos SI(S) (SF(S)) a la secuencia así obtenida.

Al justificar a la izquierda (derecha) una secuencia dada S y obtener S^L (S^R) se sigue el orden creciente de los inicios (decreciente de los finales), desempatando de una cierta forma entre las actividades con mismo fin (inicio). Es obvio que para cualquiera de estas formas existe una AL representación (inversa) λ de manera que justificando las actividades en S siguiendo el orden que marca λ se llega a S^L (S^R).

PROPOSICIÓN 3.1

$SF(S) = S^R$ ($SI(S) = S^L$) si ambas son obtenidas mediante la misma AL representación (inversa) λ.

Demostración

Veamos $SF(S) = S^R$, la otra demostración es simétrica.

Sea S_k^R la secuencia obtenida tras justificar la k-ésima actividad de λ. Esta secuencia cumple 1) inicio(i,S_k^R) = s_i $\forall$i con orden(i,λ) > k y 2) inicio(i,S_k^R) = inicio(i,S^R) $\forall$i con orden(i,λ) $\leq$ k. Sea $SF_k(S)$ la secuencia obtenida tras secuenciar la k-ésima actividad de λ. Vamos a demostrar por inducción que se cumple fin(i,$SF_k(S)$) = fin(i,S_k^R) $\forall$i con orden(i,λ) $\leq$ k.

k = 1. La primera actividad de λ, i, es la de mayor final en S de todas, por lo que su fin es T(S) y su único sucesor es n. Esto quiere decir que está secuenciada lo más tarde posible dentro de S, i.e., justificada a la derecha, por lo que fin(i,S_1^R) = fin(i,S) = T(S). La secuencia $SF_1(S)$ se construye con la actividad n colocada en T(S). Al estar todos los recursos disponibles, i puede terminar en T(S) (obviamente su fin debe ser menor o igual a T(S) porque s_n = T(S)). Esto lleva a fin(i,$SF_1(S)$) = fin(i,S_1^R), por lo que la hipótesis de inducción se cumple para k = 1.

k → k +1. Supongamos que es cierto para k e intentémoslo demostrar para k+1. Para todo i con orden(i,λ) $\leq$ k se cumple por hipótesis de inducción fin(i,$SF_k(S)$) = fin(i,S_k^R). Pero estas actividades quedan fijas para el resto de secuenciaciones/justificaciones de actividades, por lo que fin(i,$SF_{k+1}(S)$) = fin(i,S_{k+1}^R) $\forall$i con orden(i,λ) $\leq$ k. Sea j la k+1-ésima actividad en λ. Falta demostrar que fin(j,$SF_{k+1}(S)$) = fin(j,S_{k+1}^R).

(≥) Para construir S^R_{k+1} ($SF_{k+1}(S)$) se justifica (secuencia) j sobre S^R_k ($SF_k(S)$). En S^R_k y $SF_k(S)$ están secuenciadas las k actividades con orden menor o igual a k en λ en los mismos intervalos, pero en S^R_k también están secuenciadas el resto de actividades, mientras que en $SF_k(S)$ no hay ninguna actividad más secuenciada. Esto quiere decir que en cada unidad de tiempo hay consumidos los mismos o menos recursos en $SF_k(S)$ que en S^R_k, por lo que al secuenciar j lo más tarde posible puede llegar en $SF_k(S)$ al menos tan lejos como al justificarla en S^R_k.

(≤) Consideremos I = [inicio(j,$SF_{k+1}(S)$),fin(j,$SF_{k+1}(S)$)]. Basta demostrar que en S^R_k existen suficientes recursos en I para secuenciar j, y que no existen problemas con las relaciones de precedencia. Veamos esto último. Sea i sucesora de j, se cumple que i está antes que j en λ (porque λ es una lista de actividades para la red inversa). Por hipótesis de inducción fin(i,$SF_k(S)$) = fin(i,S^R_k), por lo que inicio(i,$SF_k(S)$) = inicio(i,S^R_k). Pero j se puede secuenciar en I en $SF_k(S)$, por lo que fin(j,$SF_{k+1}(S)$) ≤ inicio(i,$SF_k(S)$), o lo que es lo mismo, fin(j,$SF_{k+1}(S)$) ≤ inicio(i,S^R_k), por lo que j se puede secuenciar en I en S^R_k por relaciones de precedencia.

Para demostrar que existen suficientes recursos consideraremos tres casos, a) fin(j,S) = fin(j,$SF_{k+1}(S)$), b) fin(j,S) ≤ inicio(j,$SF_{k+1}(S)$) y c) fin(j,S) ∈]inicio(j,$SF_{k+1}(S)$),fin(j,$SF_{k+1}(S)$)[.

a) fin(j,$SF_{k+1}(S)$) = fin(j,S) = fin(j,S^R_k). Como fin(j,S^R_{k+1}) ≥ fin(j,S^R_k), se tiene lo que se quiere demostrar.

b) Sea i que se secuencie en I en S^R_k. Entonces fin(i,S^R_k) > inicio(j,$SF_{k+1}(S)$) e inicio(j,$SF_{k+1}(S)$) ≥ fin(j,S) por hipótesis del caso b), por lo que i está antes que j en λ. Por hipótesis de inducción fin(i,$SF_k(S)$) = fin(i,S^R_k), por lo que i se secuencia en $SF_k(S)$ en el mismo intervalo que en S^R_k. Esto implica que todas las actividades que se secuencian en I en S^R_k también se secuencian en $SF_k(S)$ en el mismo intervalo, por lo que en S^R_k existen al menos los mismos recursos disponibles en I que en $SF_k(S)$, donde sí cabe j, por lo que j se puede secuenciar por recursos en I en S^R_k.

c) fin(j,S) pertenece al intervalo]inicio(j,$SF_{k+1}(S)$),fin(j,$SF_{k+1}(S)$)[. Por construcción, en S^R_k en [inicio(j,$SF_{k+1}(S)$),fin(j,S)] = [inicio(j,$SF_{k+1}(S)$),fin(j,S^R_k)] se está secuenciando j, por lo que si la quitamos para justificarla existen suficientes recursos para secuenciarla. Esto implica que falta demostrar que en I' = [fin(j,S),fin(j,$SF_{k+1}(S)$)] hay bastantes recursos para secuenciar j. Sea i que se secuencie en I' en S^R_k. Entonces fin(i,S^R_k) > fin(j,S), por lo que estamos en el mismo caso que en b) pero con un intervalo distinto. Aplicando el mismo razonamiento llegamos a que i se secuencia en I' en $SF_k(S)$, en las mismas unidades de tiempo, por lo que j se puede secuenciar en S^R_k porque se puede secuenciar en $SF_k(S)$.

Hemos demostrado por inducción que para todo k se cumple fin(i,SF_k(S)) = fin(i, S_k^R) ∀i con orden(i,λ) ≤ k. En particular se cumple SF(S) = S^R. Q. E. D.

El tiempo de computación de la justificación

Si nos fijamos exclusivamente en la calidad de las soluciones obtenida, parece claro que la inclusión de la justificación mejora sustancialmente los algoritmos – al menos con los que hemos trabajado. Pero el tiempo de cómputo también es clave en el comportamiento de un heurístico: la justificación podría no ser interesante si el tiempo de ejecución de los algoritmos cuando la emplean fuera muy superior al de los originales.

Podemos dividir el tiempo de ejecución en dos partes, el empleado en secuenciar y el utilizado en el resto de procedimientos. Este último sirve básicamente para calcular nuevas codificaciones para secuenciar ('preparar las secuencias'). La relación de tiempos de ejecución entre un algoritmo más la doble justificación y el original dependerá de la diferencia entre preparar y calcular una secuencia en cada uno de ellos, y ésta variará según el método original.

En el terreno de secuenciar propiamente dicho, la justificación es, a priori, como mínimo igual de eficiente que la mayoría de heurísticos (que emplean Serie aplicado sobre una codificación), dado que utiliza el método Serie adaptado para listas de actividades. Se ha comprobado (cf. Hartmann, 2000, pags. 126 y 127) que los metaheurísticos basados en esta codificación son más rápidos, dado que al secuenciar no tienen que mantener el conjunto de elegibles ni escoger en cada iteración la actividad de mayor prioridad. En cuanto a la preparación de la codificación, la justificación necesita exclusivamente calcular un vector por finales o inicios de la secuencia que se quiere justificar, y la proposición 3.2 demuestra que esto se puede realizar en O(n) operaciones básicas. Esto quiere decir que la preparación consume mucho menos tiempo que la secuenciación a medida que crece n, ya que Serie es de $O(n^2K)$; además, un gran número de formas de obtener codificaciones son, como mínimo, lineales (obtener listas de actividades aleatorias, aplicar reglas de prioridad o muestreos, la mayoría de operadores binarios, entre ellos el de cruce de dos puntos de corte, ...) y muchas de ellas deben considerar las relaciones de precedencia del grafo, algo no necesario en el procedimiento que veremos. Si juntamos ambas propiedades resulta que en general la aplicación de la (doble) justificación no va a añadir tiempo al algoritmo original. De hecho, en algoritmos más complejos, la inclusión de la justificación puede reducir el tiempo total. Un ejemplo clarificador de estos comentarios lo tenemos en Tormos y Lova, 2001, donde la diferencia entre

calcular una secuencia mediante el procedimiento que no es justificar (un muestreo aleatorio sesgado basado en la peor elección) y justificar es tanta, que consideran oportuno contabilizar cada aplicación de DJ como una única secuencia.

PROPOSICIÓN 3.2

Dada una secuencia S, supongamos que disponemos de los finales de las actividades en un vector de n componentes *fin*, y de un vector de K componentes *t*, $K \leq n$, con los instantes de tiempo donde finaliza alguna actividad en orden creciente. Entonces, se puede calcular un vector por finales de S, λ, con O(n) operaciones básicas.

Demostración

Sea T = T(S).

Consideremos el siguiente algoritmo:

1. Sea *Acts* una matriz de enteros T x n (un vector de T vectores de n enteros cada uno), *nActs* un vector de T enteros. Acts(r) es un vector que va a contener las actividades que finalizan en r, nActs(r) va a indicar el número de las mismas.
2. Desde i = 1 hasta K, hacer nActs(t(i)) = 0.
3. Desde i = 1 hasta n, hacer
   3.1 nActs(fin(i)) = nActs(fin(i)) + 1.
   3.2 Acts(fin(i))(nActs(fin(i))) = i.
4. j = 1.
5. Desde i = K hasta 1, hacer
   5.1 Desde h = 1 hasta nActs(t(i)), hacer
      5.1.1 λ(j) = Acts(t(i))(h).
      5.1.2 j = j + 1.

Veamos primero que λ es un vector por finales de S, i.e., $f_i < f_j \rightarrow$ orden(i,λ) > orden(j,λ). Pero esto es claro, porque si f_i es menor de f_j entonces, si pos1 y pos2 son las posiciones que cumplen t(pos1) = f_i, t(pos2) = f_j, se tiene pos1 < pos2 y j está en el vector Acts(t(pos1)) e i en Acts(t(pos2)). Como en el bucle 5 pasamos por Acts(pos2) antes de por Acts(pos1), i está colocada en λ en una posición posterior a j.

Para calcular el orden lo haremos iteración a iteración. En la 1ª iteración no se realiza ninguna operación y en la 4ª sólo una. En la 2ª se recorre el índice i K veces y se asigna 0 una vez por cada i, en total 2K operaciones. En cada iteración del bucle de la 3ª iteración se realizan 2 operaciones, más otra más para recorrer el índice, en total 3n

operaciones. En el bucle de la 5ª iteración se efectúan sin contar los aumentos de índices, $\sum_{i=1}^{K}\sum_{h=1}^{nActs(t(i))} 2$ = 2n operaciones, ya que se pasa una vez por cada actividad, más K+ 2n - 1 aumentos de índices, K para i, n para h y n – 1 para j. Contabilizando todo, se tiene 0 + 2K + 3n + 0 + 4n + K = 7n + 3K ≤ 10n, dado que K ≤ n, por lo que el orden del algoritmo es n.

Q.E.D.

Las hipótesis consideradas restringen muy poco el resultado, porque los finales de las actividades han debido ser calculados necesariamente al secuenciarlas. Además, al secuenciar es habitual disponer de una lista con los instantes de tiempo donde pueden cambiar los recursos disponibles, i.e., aquellos donde finaliza alguna actividad, en orden creciente de tiempo. Basta entonces traspasar los datos de esa lista a t, y para ello son necesarias O(K) ≤ O(n) operaciones. Obviamente K, el número de instantes de tiempo distintos donde finalizan actividades es menor o igual de n y, en general, será bastante menor de n.

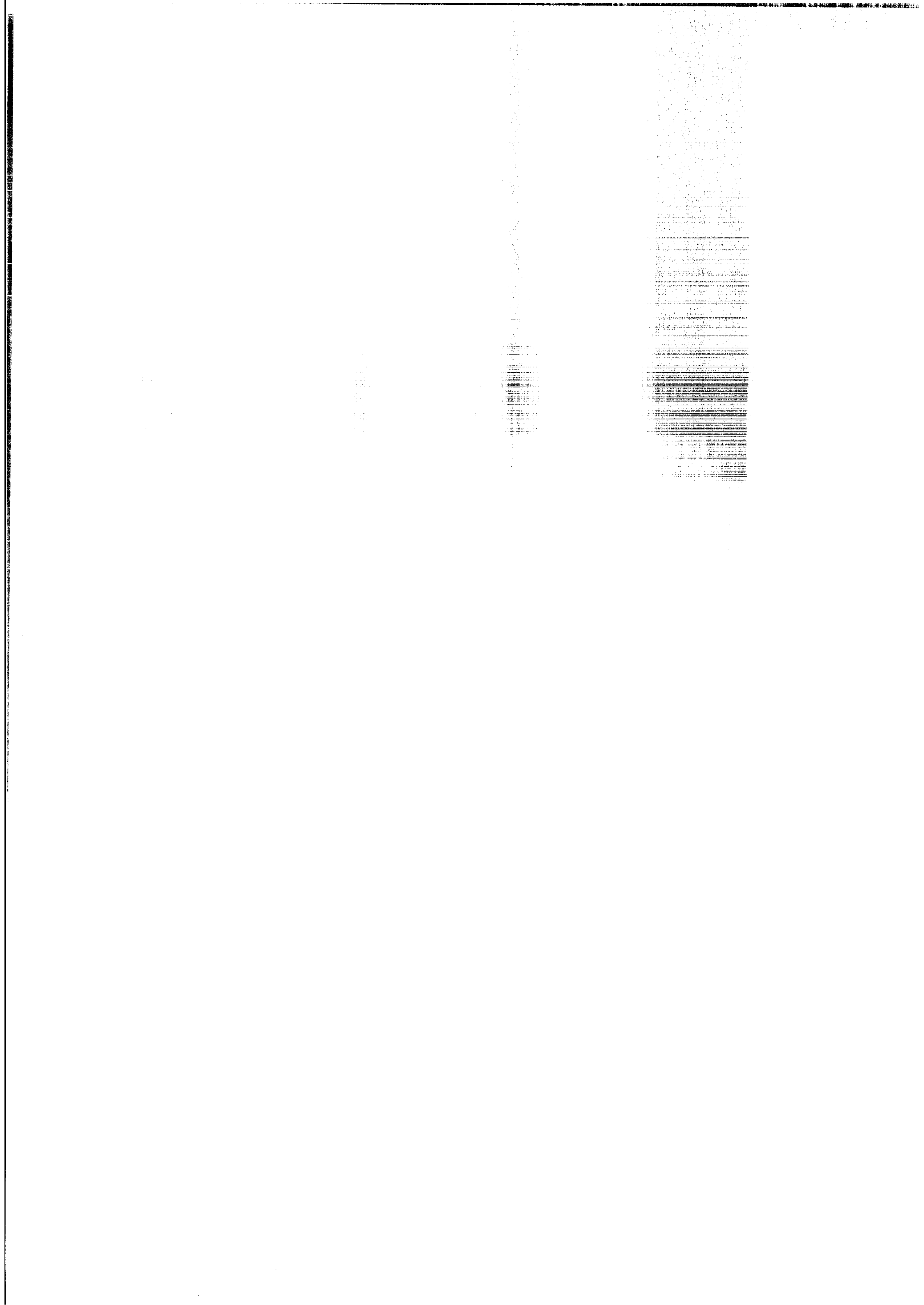
A pesar de las consideraciones efectuadas, también existe una desventaja en términos temporales en la aplicación de la justificación. En algunos algoritmos se puede abortar la secuenciación de una solución si se está seguro que va a proporcionar una duración mayor que un cierto valor. Así, en el genético de Hartmann, se puede no secuenciar completamente una solución siempre y cuando se pueda asegurar que la longitud del hijo será mayor que la del peor de los individuos de la población de los padres ya que, en ese caso, jamás sobrevivirá al proceso de selección. Por el contrario, en Hartmann + DJ no podemos no terminar esa secuencia, debido a que vamos a realizar la doble justificación sobre ella. Sí se puede abortar la justificación a la izquierda, pero sólo se puede intentar cortar una de cada tres soluciones que se secuencian. Por el contrario, en el algoritmo original se intenta no secuenciar hasta el final cada solución. Esto podría repercutir enormemente en el tiempo final del algoritmo, pero no es así debido a que las cotas que se emplean son las obtenidas con el CPM, bastante pobres en general, que no suelen permitir abortar la secuencia hasta que quedan muy pocas actividades por secuenciar. Un ejemplo de esto es que la inclusión de la justificación en el algoritmo de Hartmann, que dispone de este mecanismo, únicamente aumenta en 0.05 segundos el tiempo medio. Además, en algunas ocasiones necesitamos disponer de la secuencia entera, por ejemplo en el algoritmo SA visto antes o, también, si queremos aplicar una función de mejora a la solución obtenida.

Precisamente SA es un buen ejemplo de algunos de los aspectos explicados en este apartado. Prácticamente emplea el 100% de su tiempo en la aplicación de

Biased(S,β), y este procedimiento se divide en el cálculo de la lista de actividades perturbada de λ(S), λ', y en su posterior secuenciación mediante Serie para listas de actividades. En este caso, el método de secuenciar es el mismo que para justificar pero no así la forma de preparar las secuencias, que en el caso de Biased(S,β) (cf. apartado 2.3 capítulo 2) tiene que considerar las relaciones de precedencia, así como controlar la actividad con menor orden en cada momento y calcular las probabilidades de las actividades elegibles cuando se decide no escoger la actividad de menor orden. Todo esto, unido a lo rápido que se prepara la lista de actividades en la justificación y a que en SA no se puede abortar las secuencias, conduce a que SA + DJ sea un 30 % más rápido que SA. Más adelante (capítulo 5) trabajaremos con un algoritmo en el que se puede abortar las secuencias y, aun así, la versión con doble justificación es bastante más rápida.

Capítulo 4

Análisis teórico de la justificación



1. INTRODUCCIÓN

En el anterior capítulo se ha demostrado la utilidad práctica de la justificación. En este capítulo queremos dotar de un marco teórico a esta técnica, definiendo las secuencias que se pueden obtener mediante su aplicación y estudiando algunas de sus generalizaciones.

Características de las secuencias creadas por la justificación

Como se ha comentado antes, al justificar según la definición del apartado 2.2 del capítulo 2 una secuencia S a la derecha (izquierda) se obtiene una solución S' posible, justificada a la derecha (izquierda) y que cumple $s_i' \geq s_i$ ($s_i' \leq s_i$) $\forall i$ y $f_n' = T(S)$ ($s_0' = 0$), por lo que $T(S') \leq T(S)$ en ambos casos. Si $s_0' > 0$ ($f_n' < T(S)$), S' es de longitud s_0' ($T(S) - f_n'$) unidades menor que S.

Definición: Secuencia justificada a la derecha de una dada

> Denominaremos **secuencia justificada a la derecha** (**izquierda**) de una solución S a una secuencia S' que cumpla estas características, i.e., S' es posible, está justificada a la derecha (izquierda), $f_n' = T(S)$ ($s_0' = 0$) y $s_i' \geq s_i$ ($s_i' \leq s_i$) $\forall i$.

Fijémonos en que esta definición no prescribe cómo se obtiene S'.

Definición: El conjunto SJD(S) (SJI(S))

> Denominaremos **SJD(S)** (**SJI(S)**) al conjunto de secuencias justificadas a la derecha (izquierda) de S.

Parece claro, por el capítulo anterior, que el conjunto SJD(S) (SJI(S)) posee a menudo secuencias de menor longitud que S activa (a la derecha), por lo que es un conjunto muy interesante desde el punto de vista de la optimización heurística. Justificando a la derecha la secuencia S según el procedimiento empleado en el capítulo 3 podemos obtener alguno de los elementos de SJD(S) pero, como se demostrará más adelante, no se puede llegar a todos. Esto quiere decir que, en teoría, pueden existir elementos de ese conjunto de mejor calidad, inaccesibles con esa técnica. En los apartados 2-5 se definen otras técnicas que también producen secuencias justificadas a la derecha y otros tipos de conjuntos. Estudiaremos a lo largo del capítulo las relaciones entre los conjuntos definidos y, con un par de ejemplos, demostraremos que esta línea de investigación también es prometedora en la práctica. En el apartado 6 se adapta la

justificación a otros problemas de secuenciación de proyectos con recursos limitados y se demuestra su utilidad en un caso concreto.

2. LA JUSTIFICACIÓN

2.1. La justificación general

Definición: La justificación general de una secuencia

La **justificación general a la derecha (izquierda) de una secuencia** S consiste en obtener una secuencia S' justificada a la derecha (izquierda) a partir de aplicar a S un número finito de justificaciones a la derecha (izquierda) a las actividades. La justificación j-ésima se lleva a cabo sobre la secuencia que refleja las j - 1 justificaciones anteriores. Si $s_1' > 0$ ($s_n' < T(S)$), entonces S' es de menor longitud que S.

Fijémonos en que esta definición no prescribe el orden en que se justifican las actividades. Distintos órdenes pueden redundar en distintas secuencias justificadas.

Definición. El conjunto JD(S) (JI(S))

Denominaremos JD(S) (JI(S)) al conjunto de secuencias que se pueden obtener por medio de una justificación general a la derecha (izquierda) de una secuencia posible S.

Ejemplo

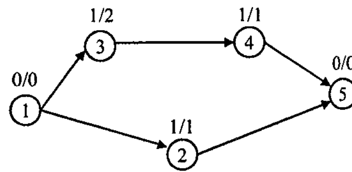
En este ejemplo se puede observar como la justificación (general) puede ser útil incluso con proyectos con muy pocas actividades. La secuencia S de la Figura 4.2 es posible y activa en el proyecto dado en la Figura 4.1. Las actividades 3 y 4 están justificadas a la derecha. Si justificamos la actividad 2 a la derecha, obtenemos la secuencia S' de la Figura 4.3. S' $\in$ JD(S), es una unidad menor que S y es óptima para el problema.

PROPOSICIÓN 4.1

JD(S) $\subseteq$ SJD(S) (JI(S) $\subseteq$ SJI(S)).

Demostración

Trivial.



$K = 1, R_1 = 2$

Figura 4.1

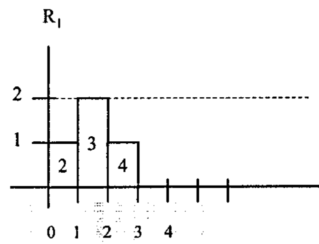


Figura 4.2

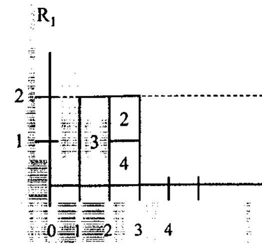
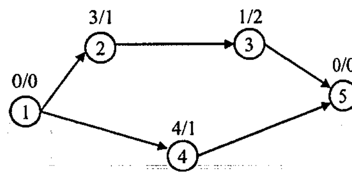


Figura 4.3



$K = 1, R_1 = 2$

Figura 4.4

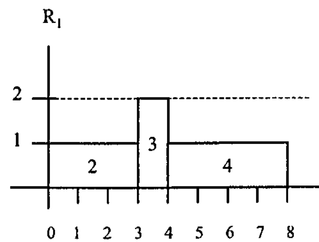


Figura 4.5

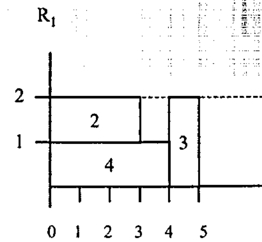


Figura 4.6

Ya hemos visto en el ejemplo anterior que se puede acortar la duración de una secuencia por medio de la justificación general. Hay en cambio otras secuencias, como la solución S de la Figura 4.5 (posible en el problema dado en la Figura 4.4), en la que esto no es posible. Este ejemplo demuestra, además, que una secuencia justificada a la derecha y a la izquierda puede no ser óptima. Esta secuencia S es tanto justificada a la izquierda como a la derecha. Eso quiere decir que JD(S) = JI(S) = {S}, por lo que es imposible mejorarla mediante la justificación general, aunque se

aplicara varias veces. Tampoco se puede encontrar una secuencia justificada a la derecha ni a la izquierda de S mejor, porque SJD(S) = SJI(S) = {S}. Sin embargo, S no es óptima, dado que la secuencia S' de la Figura 4.6 es 3 unidades menor. Este ejemplo nos sirve, además, para comprobar que la intersección entre las secuencias justificadas a la izquierda y a la derecha no es vacía en general, pero que sus elementos no tienen por qué ser especialmente interesantes desde el punto de vista de la calidad.

En los puntos siguientes introduciremos distintas formas concretas de justificación general. Para poder compararlas haremos uso de la siguiente notación. Si S es una solución posible y A(S) y B(S) denotan el conjunto de secuencias obtenibles a partir de S mediante las técnicas relacionadas con A y B, respectivamente, escribiremos $A \subseteq B$ para simbolizar $A(S) \subseteq B(S)\ \forall\ S$, i.e., cuando la técnica B sea capaz de obtener todas las secuencias que se pueden alcanzar con A para cualquier solución inicial. Del mismo modo, $A \subset B$ querrá decir que $\exists\ S\ /\ A(S) \subset B(S)$ (además de $A(S) \subseteq B(S)\ \forall\ S$), teniendo en cuenta que esto no implica $A(S) \subset B(S)\ \forall S$. Esto se traduce en que B es estrictamente más general que A, o que A no es capaz de alcanzar todas las soluciones de B. Podría ocurrir que esas secuencias que se le escapan a A no fueran importantes, en el sentido de que la calidad obtenible por A fuera igual que la de B. Esto lo escribiremos MA = MB, que quiere decir $\min\{T(S'), S' \in A(S)\} = \min\{T(S'), S' \in B(S)\}\ \forall\ S$. Análogamente, $MA \leq MB$ simboliza $\min\{T(S'), S' \in A(S)\} \geq \min\{T(S'), S' \in B(S)\}\ \forall\ S$ y $MA < MB \equiv \exists\ S\ /\ \min\{T(S'), S' \in A(S)\} > \min\{T(S'), S' \in B(S)\}$, además de $MA \leq MB$. El símbolo MA se podría considerar como las mejoras obtenibles mediante la técnica relacionada con A.

2.2. La justificación por extremos

La justificación que habíamos empleado antes de este capítulo es un caso particular de la justificación general, puesto que las justificaciones de las actividades se realizan según un cierto orden, el que marcan los finales (inicios) de las actividades. Es una de las formas más naturales de justificar, y es la única forma que se ha empleado con el RCPSP en la literatura, según nuestro conocimiento. En este capítulo la denominaremos **justificación por extremos** - y reservaremos justificación para la justificación general - porque cuando se justifica a la derecha se efectúan las justificaciones según el orden decreciente de finales (justificación por finales) y cuando se justifica a la izquierda, en orden creciente de inicios (justificación por inicios). Si S es una secuencia, denominaremos JD(S,finales) (JI(S,inicios)) a las secuencias obtenidas mediante la justificación a la derecha por finales (izquierda por inicios) de S.

Como se ha comentado, JD(S,finales) $\subseteq$ JD(S) $\subseteq$ SJD(S) $\forall$S (JI(S,inicios) $\subseteq$ JI(S) $\subseteq$ SJI(S) $\forall$S). Siguiendo la notación introducida podemos escribir JD(finales) $\subseteq$ JD y MJD(finales) $\leq$ MJD, donde JD(finales) (JI(inicios)) simboliza la técnica de justificar a la derecha por finales (izquierda por inicios) y MJD(finales) (MJI(inicios)) denota las mejoras obtenibles mediante la justificación por finales (inicios).

Una de las propiedades más importantes de la justificación por extremos es que sólo son necesarias n justificaciones como máximo para obtener la secuencia final, justificada a la derecha (izquierda). Además, justificar por extremos es equivalente a secuenciar según el orden en que se justifica, como se vio al final del capítulo anterior. Esto repercute enormemente en la rapidez con la que se puede justificar.

Como se ha comentado antes, para cada secuencia S existen a priori varias maneras de justificar por finales (inicios), una por cada forma en que se desempate las actividades que finalicen (comiencen) al unísono. No todas estas justificaciones conducen a la misma secuencia; de hecho, el ejemplo siguiente demuestra que la mejora al justificar por finales (inicios) puede depender de cómo se desempate.

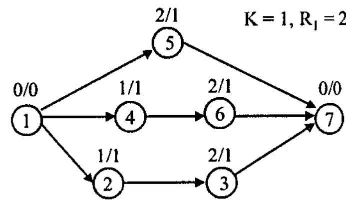


Figura 4.7

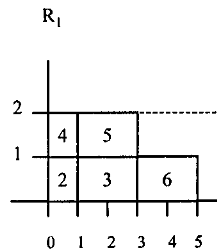


Figura 4.8

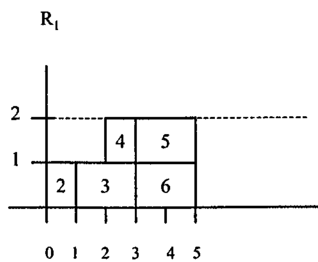


Figura 4.9

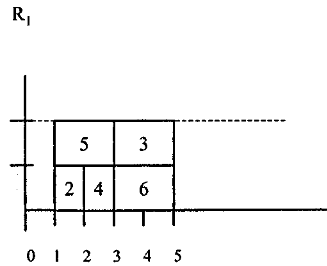


Figura 4.10

En la secuencia S de la Figura 4.8, posible para el proyecto representado en la Figura 4.7, los vectores λ = (7 6 5 3 4 2 1) y λ'= (7 6 3 5 4 2 1) son vectores por finales para S. Si se justifica por finales empleando λ se llega a la secuencia S' de la Figura 4.9, mientras que λ' conduce a S'' (Figura 4.10). La primera es de la misma longitud que S, mientras que S'' es una unidad menor, y óptima para el problema. La única diferencia

entre ambas es que λ prefiere justificar 5 antes que 3, mientras que λ' escoge lo contrario.

En el apartado 2.1 del capítulo anterior se vio que no existía demasiada diversidad en las duraciones de las diferentes secuencias de JD(S,finales) que se obtenían al desempatar de forma aleatoria. Sin embargo, sí existía una mayor diversidad en JD(S,inicios), a pesar de que las soluciones iniciales eran de mayor calidad y existía una mayor diferencia entre la mejor de esas secuencias y la media. No es descartable que existan reglas de desempate mejores que otras, al igual que algunas reglas de prioridad obtienen mejores resultados en media que otras. Siguiendo esa analogía las reglas de elección de la actividad a justificar pueden ser estáticas o dinámicas, además de poder contener información histórica de otras justificaciones. Dado que la incorporación de la (doble) justificación por extremos es tan útil, parece recomendable estudiar en el futuro esas reglas de desempate. Más adelante realizaremos algunos comentarios sobre la diversidad existente al aplicar un tipo de justificación un poco más general que la justificación por extremos.

2.3. El algoritmo de justificación general

La definición de justificación general dada en el apartado 2.1 no proporciona ningún método concreto para justificar una secuencia. Al igual que Serie es un método mediante el cual se puede obtener cualquier secuencia activa, sería interesante disponer de un procedimiento que en teoría pudiera calcular cualquier secuencia de JD(S) (JI(S)). El siguiente algoritmo cumple ese objetivo con JD(S) (con JI(S) sería simétrico). Denotaremos s_i' el inicio de la actividad i en la secuencia actual, obtenida tras las justificaciones que se hayan realizado, y s_i* el inicio de i si se justificara sobre esa secuencia.

Figura 4.11. Método de justificación general

1. Realizar $s_i' = s_i \; \forall i$.
2. Calcular $s_i^* \; \forall i$. Si $s_i' = s_i^* \; \forall i$, S' está justificada a la derecha, acabar.
3. Escoger j / $s_j' \neq s_j^*$ y justificarla a la derecha (i.e., hacer $s_j' = s_j^*$).
4. Recalcular $s_i^* \; \forall i$. Si $s_i' = s_i^* \; \forall i$, S' está justificada a la derecha, acabar. En otro caso, ir a 3.

Está claro que mediante este algoritmo se pueden obtener todas las secuencias de JD(S). Además, sean cuales sean las elecciones del paso 3, el proceso es finito, ya que el número de veces que se puede escoger una actividad i en la etapa 3 es, como máximo, el número de veces que se puede mover a la derecha, que a su vez está

acotado por T(S) – f_i. Luego el máximo número de iteraciones en el algoritmo está acotado por $\sum_{i=1}^{n}(T(S)-f_i) \leq n \cdot T(S)$.

Otro aspecto interesante a comentar es que el algoritmo descrito no es el que se emplearía en la práctica. En primer lugar, no se necesitaría calcular la mayoría de s_i^* tras cada justificación de una actividad, porque ésta no va a modificar el intervalo de justificación de la mayoría de actividades. Además, si denominamos actividades **fijas** a las que no se van a poder justificar más aunque se justifique cualquier conjunto de actividades de cualquier manera, en cada iteración se podrían fijar actividades atendiendo a diferentes criterios y no considerarlas más. Una condición suficiente para fijar una actividad i sería la de que $s_i^* = s_i'$ y $\forall j$ tal que $s_j' + d_j > s_i' + d_i$ se tiene $s_j^* = s_j'$. Esto último implica que las actividades que finalizan después de i están fijas, por lo que aunque se realicen más justificaciones de actividades no va a disminuir el consumo de recursos en ninguna unidad de tiempo de $[f_i', T(S)]$. Como con las disponibilidades de recursos actuales i no se puede justificar más ($s_i^* = s_i'$), se puede asegurar que i está fija. Fijémonos en que esta condición la cumplen todas las actividades que han sido justificadas por finales, por lo que no es necesario justificarlas más.

Muchos de los procedimientos del RCPSP no trabajan con todas las actividades a la vez sino que sólo se ocupan, en cada instante, de las 'elegibles', aquellas actividades cuyos predecesores (o sucesores, dependiendo por dónde empiece el algoritmo) han sido ya considerados en su momento y se ha acabado de trabajar con ellos. Dos de estos procedimientos son Serie y Paralelo, y, otros, son las diversas formas de muestreo, una de las formas de generar listas de actividades aleatorias y el método para convertir vectores de prioridad en listas de actividades. Dada la abundante presencia de algoritmos de este tipo y a su importancia dentro del RCPSP, parece apropiado definir una justificación que proceda de esa manera, y profundizar en el conocimiento de su potencialidad frente a las demás.

Definición: La justificación por elegibles de una secuencia

> La **justificación a la derecha** (**izquierda**) por elegibles **de una secuencia** S consiste en obtener una secuencia S' justificada a la derecha (izquierda) a partir de aplicar a S un número finito de justificaciones a la derecha (izquierda) a las actividades con una condición: una actividad puede ser escogida para ser justificada cuando todas sus sucesoras (predecesoras) están fijas. La justificación j-ésima se lleva a cabo sobre la secuencia que refleja las j - 1 justificaciones anteriores.

Definición: El conjunto JD(S,elegibles) (JI(S,elegibles))

Denominaremos **JD(S,elegibles)** (**JI(S,elegibles)**) al conjunto de secuencias que se pueden obtener al secuenciar una secuencia S posible a la derecha (izquierda) por elegibles. Cuando no tengamos en cuenta la solución inicial lo denotaremos JD(elegibles) (JI(elegibles)), y a las mejoras obtenibles mediante la justificación por elegibles las denotaremos MJD(elegibles) (MJI(elegibles)).

Esta forma de justificar es teórica, porque es complicado llevarla a la práctica en su forma original, dado que asegurar que una actividad está o no fija puede ser complicado. Es sin embargo sencillo modificar el algoritmo visto anteriormente para adaptarlo a una justificación a la derecha (izquierda) por elegibles modificada; basta trabajar con una condición suficiente de fijar actividades y considerar como elegibles en cada iteración aquellas actividades cuyas sucesoras (predecesoras) están fijas de acuerdo con esa condición – aunque alguna de esas actividades elegibles en realidad esté fija y podría dar paso a sus sucesoras –, comenzando con las predecesoras de n (sucesoras de 1), ya que ésta la fijamos en T(S) (0) antes de justificar. En los ejemplos que mostraremos más adelante se empleará la siguiente condición suficiente: en la justificación a la derecha (izquierda) se fija una actividad i justificada a la derecha (izquierda) si todas las actividades con inicio mayor o igual que el final de i (final menor o igual que el inicio de i) están justificadas a la derecha (izquierda). A la justificación por elegibles con esta condición suficiente de fijar actividades la denominaremos **justificación por elegibles práctica.** En principio pueden existir secuencias S en las que la justificación por elegibles práctica no sea capaz de calcular todas las soluciones de JD(S,elegibles).

Hemos presentado tres maneras diferentes de justificar, la general, por extremos y por elegibles. Sería interesante conocer las relaciones existentes entre estos tipos de justificación, especialmente si las más restrictivas a priori son capaces de obtener todas las secuencias del conjunto JD(S) para todo S. También es importante comprobar si mediante la justificación se pueden obtener todas las secuencias de SJD(S). La siguiente proposición nos revela esas relaciones y nos contesta no a la últimas incógnitas.

PROPOSICIÓN 4.2

1) JD(finales) $\subset$ JD(elegibles) $\subset$ JD $\subset$ SJD (JI(inicios) $\subset$ JI(elegibles) $\subset$ JI $\subset$ SJI).

2) MJD(finales) $<$ MJD(elegibles) $<$ MJD $<$ MSJD (MJI(inicios) $<$ MJI(elegibles) $<$ MJI $<$ MSJI).

Demostración

Las inclusiones son triviales por la propia definición: la justificación por finales es claramente un tipo de justificación por elegibles, dado que los sucesores de una actividad i finalizan después de i, por lo que son considerados antes. Los contraejemplos los veremos a continuación.

(1) JD(finales) ≠ JD(elegibles), MJD(finales) < MJD(elegibles).

Consideramos el proyecto de la Figura 4.12; la secuencia S de la Figura 4.13 es activa para ese proyecto. Sólo existen dos maneras de justificar S por finales, (7 6 4 3 5 2 1) y (7 6 4 5 3 2 1), dado que las únicas actividades que finalizan a la vez son 3 y 5. Ambas formas conducen a la secuencia S' de la Figura 4.14, de la misma longitud que S. Es obvio que justificar a la izquierda por inicios S' llevaría de nuevo a S, por lo que la justificación por extremos no es capaz de mejorar S, aunque se aplique de forma reiterada. Sin embargo, la justificación por elegibles permite justificar primero la actividad 2, que es elegible desde la primera iteración, y obtener S", la secuencia de la Figura 4.15, justificada a la derecha y 3 unidades menor que S.

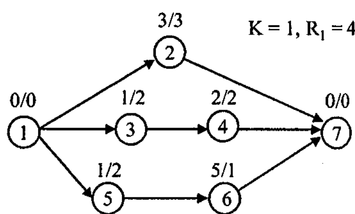


Figura 4.12

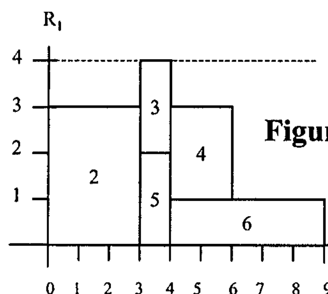


Figura 4.13

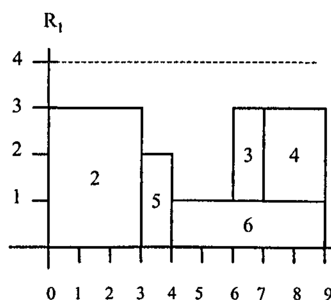


Figura 4.14

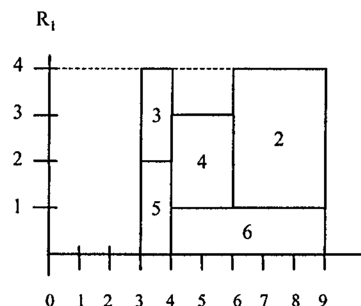


Figura 4.15

(2) JD(elegibles) ≠ JD(general), MJD(elegibles) < MJD(general).

La secuencia S de la Figura 4.17 es activa en el proyecto de la Figura 4.16. Veamos que la única secuencia que se puede obtener mediante la justificación por elegibles es S', representada en la Figura 4.19. No vamos a considerar las actividades 11, 9, 8, 7, 5 y 3

porque están fijas y no se pueden justificar más a la derecha; sólo consideraremos pues 2, 4, 6, 10 y 12. En la primera iteración, tras fijar la actividad ficticia 13 en T(S) = 9, las actividades elegibles son 12 y 10. La actividad 12 no se puede mover a la derecha, por lo que tenemos que escoger 10, que se pasa a secuenciar en [8,9]. En la 2ª iteración, las actividades elegibles son 12 y 6, la primera de las cuales sigue estando justificada a la derecha. Debemos elegir 6, que tras su justificación se secuencia en [7,8]. En la 3ª iteración son elegibles 12 y 2. Ambas se pueden justificar, supongamos en primer lugar (caso a)) que justificamos 2. En ese caso, 2 se secuencia en [1,2], y en la 4ª iteración a) la única elegible es 12, que al justificarla se secuencia en [4,7]. En la 5ª iteración a) la actividad elegible es 4, que está justificada a la derecha, por lo que la secuencia obtenida, S' (Figura 4.19), es la secuencia resultante. Si por el contrario escogemos en la 3ª iteración la actividad 12 (caso b)), ésta se secuencia en [4,7] y en la iteración 4ª b) las elegibles son 2 y 4. La última no se puede mover más a la derecha, por lo que justificamos 2 hasta [1,2]. En la siguiente iteración sólo 4 es elegible y no se puede justificar, por lo que llegamos también a S'. Esto demuestra que mediante la justificación por elegibles no se puede mejorar S, pero mediante la justificación general sí, porque se puede llegar a la secuencia S" de la Figura 4.18. Primero justificamos 6, que pasa a secuenciarse en [6,7]. Después justificamos 10 hasta [8,9], y luego 12 hasta [5,8]. Esto deja un hueco en [3,5] suficientemente grande para justificar 4 allí, lo que al justificar 2 a [1,2] lleva a S", justificada a la derecha y 1 unidad menor que S.

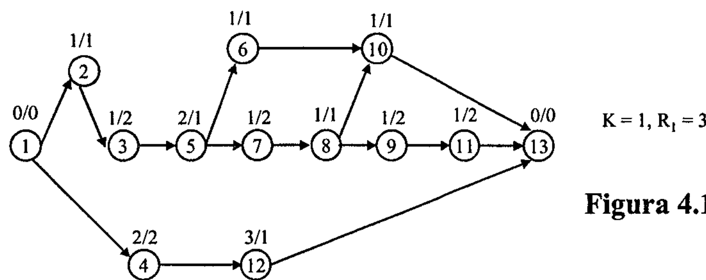


Figura 4.16

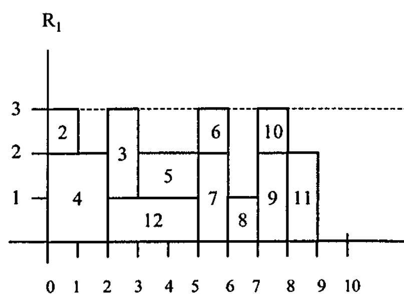


Figura 4.17

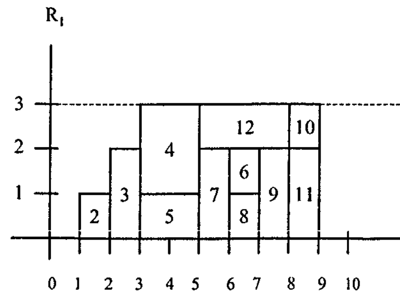


Figura 4.18

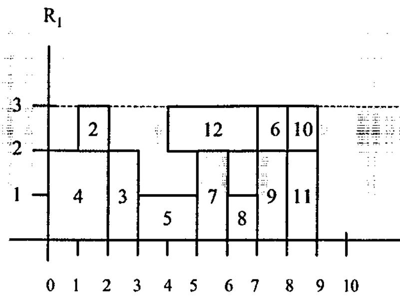


Figura 4.19

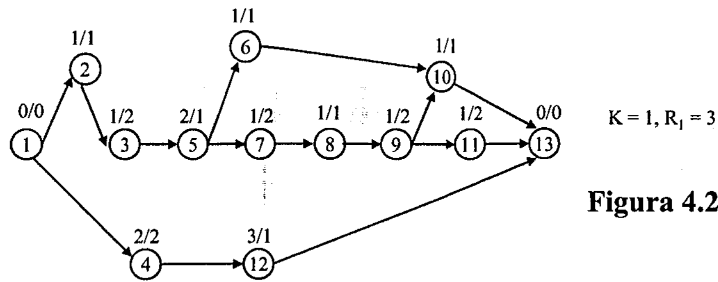
(3) JD ≠ SJD, MJD < MSJD

La Figura 4.21 muestra una secuencia S, posible para el proyecto de la Figura 4.20. Vamos a demostrar que JD(S) = {S'}, con S' la secuencia de la Figura 4.23, de la misma duración que S. Las actividades 11, 10, 9, 8, 7, 5 y 3 están fijas y no se pueden mover, por lo que las actividades que debemos considerar son 2, 4, 12 y 6. Al principio sólo se pueden mover 6 y 2. El intervalo de justificación de 2 es [1,2], que no puede interrumpir la justificación de ninguna actividad y no puede cambiar aunque justifiquemos otras actividades. Teniendo en cuenta esto justificamos 2 a [1,2], y la única actividad no justificada es la 6, por lo que tenemos que elegirla y secuenciarla en [7,8]. Una vez hecho esto la actividad 12 es la única que pasa de estar justificada a la derecha a no estarlo, por lo que escogemos esa actividad y la justificamos hasta [4,7]. Esta acción lleva a S' (Figura 4.23), justificada a la derecha, por lo que es la única solución obtenible mediante la justificación general. Sin embargo, la secuencia S" de la Figura 4.22 es justificada a la derecha de S, y es una unidad mejor que S y S', por lo que queda demostrado que mediante la justificación general no se pueden obtener ni todas las secuencias ni todas las mejoras alcanzables de SJD(S).

Q. E. D.

Hemos comentado antes que la justificación por extremos asegura que se puede obtener la secuencia justificada a la derecha o izquierda con n justificaciones. Acabamos de demostrar que esta justificación no es suficiente para obtener todas las mejoras alcanzables mediante la justificación general, por lo que, en general, se

necesitarán más de n justificaciones para calcular algunas de esas secuencias. Se puede demostrar con un ejemplo que, a veces, es estrictamente necesario justificar más de n veces para obtener la mejor solución de SJD(S).



K = 1, R_1 = 3

Figura 4.20

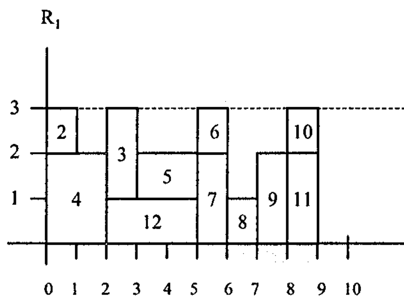


Figura 4.21

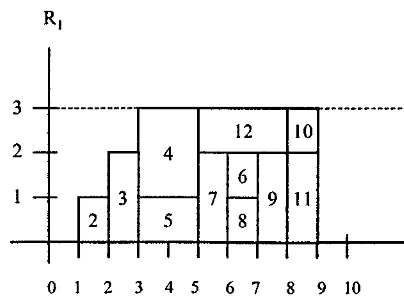


Figura 4.22

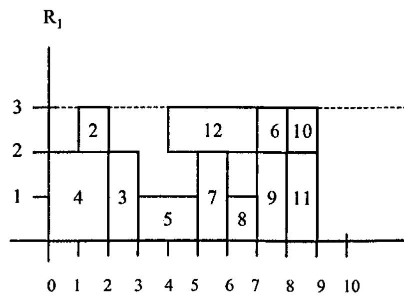


Figura 4.23

3. UN PAR DE EJEMPLOS PRÁCTICOS

Hemos demostrado con ejemplos pequeños que, en ocasiones, es posible obtener mejores secuencias si se justifica de forma general o por elegibles que si se realiza por extremos. Pero bien podría ocurrir que en general y/o para problemas más grandes no hubiera diferencias significativas entre las tres formas de justificar. Vamos a ver un par de experimentos que demuestran que esto no es cierto.

Secuencias aleatorias

Vamos a estudiar algunas diferencias existentes entre la justificación por extremos y la justificación por elegibles al aplicarlas a secuencias activas aleatorias. Para ello hemos generado 1666 secuencias activas aleatorias en cada instancia del conjunto j120 y a cada solución le hemos aplicado la justificación práctica por elegibles, una o varias veces. En un primer caso escogeremos de forma aleatoria la actividad a justificar. En el segundo caso se empleará una regla distinta.

Para poder comparar los conjuntos JD(elegibles) con JD(extremos) hemos replicado uno de los experimentos realizados en el capítulo anterior (Tabla 3.3, pag. 117), justificando cada secuencia por elegibles de forma aleatoria varias veces y calculando la media de las mejoras relativas tras 1 justificación a la derecha (1ª JD) y la media, el máximo y la desviación típica tras 10 justificaciones a la derecha (10 JD). Hemos calculado los mismos valores pero sobre la justificación a la izquierda (1ª JI y 10 JI), aplicada sobre una justificación a la derecha de las soluciones iniciales. Los resultados se presentan en la Tabla 4.1.

	media	máximo	desviación típica
1ª JD	7.5159 %	-	-
10 JD	7.5152 %	10.5349%	1.8667 %
1ª JI	1.3712 %	-	-
10 JI	1.3704 %	3.2460 %	1.0003 %

Tabla 4.1. Desviación media tras 1 justificación por elegibles aleatoria y media, máxima y desviación típica tras 10 justificaciones por elegibles aleatoria.

La media tras la primera justificación a la derecha, así como tras 10 justificaciones, es menor en este caso que en la justificación por extremos; incluso el máximo tras 10 justificaciones es menor que la media tras la primera justificación a la derecha por

finales. Al comparar estos resultados parece claro que la calidad media de las soluciones de JD(elegibles) es sensiblemente peor que la de JD(finales). También se aprecia que la desviación tras 10 justificaciones por finales es mucho menor que en el caso de la justificación por elegibles, especialmente si se tiene en cuenta la diferencia en las medias. Esto implica una clara mayor diversidad en JD(elegibles) que en JD(finales). No podemos comparar las justificaciones a la izquierda por extremos y por elegibles en términos absolutos porque parten de soluciones de diferente calidad, pero se sigue apreciando una diferencia mucho mayor entre el máximo tras 10 justificaciones y la media y una clara mayor diversidad en la justificación por elegibles.

También hemos aplicado la doble justificación práctica por elegibles guiada por una regla para escoger la actividad a justificar primero. Pruebas preliminares nos han llevado a elegir la siguiente regla: al justificar a la derecha (izquierda) se elige primero la actividad que más tarde (pronto) pueda finalizar (empezar). En caso de empate se escoge la que finalice (empiece) más tarde (pronto) en la secuencia original. Con esta regla sólo son necesarias n justificaciones, ya que al justificar una actividad quedará fija, y la justificación por elegibles práctica es equivalente a la justificación por elegibles.

Los resultados se presentan en la Tabla 4.2. Los números media_JD y %mejoradas_JD simbolizan, respectivamente, la media sobre todas las secuencias de las mejoras relativas tras la justificación a la derecha y el porcentaje de soluciones mejoradas tras esa justificación. Análogamente se calcula media_JI y %mejoradas_JI con la justificación a la izquierda. Los términos Σ, desv_CPM y desv_UB fueron introducidos en el capítulo 2.

	Σ	desv_CPM	desv_UB	media_JD/ %mejoradas_JD	media_JI/ %mejoradas_JI
Aleatorias	77405	36.39	3.83	12.76/98.66	1.73/61.16

Tabla 4.2. La justificación por elegibles en secuencias aleatorias.

Si comparamos estos resultados con los de la justificación por extremos (Tabla 3.1, pág. 114 y Tabla 3.2, pág. 115) observamos que ha habido una mejora en la desviación con respecto del CPM (UB) de casi 2 (1.3)%. Esta diferencia no se refleja en media_JD, 12.76 v. 12.43 de la justificación por finales, aunque sí hay más diferencia en media_JI, 1.73 v. 1.14. Los porcentajes de mejora tampoco varían mucho, e incluso %mejoradas_JD es mayor en la justificación por extremos.

La mejora media obtenida con esta regla, 12.76%, es bastante mayor que el máximo alcanzado tras 10 justificaciones por elegibles aleatorias (10.53%). Esto reafirma la gran diversidad de JD(elegibles).

Parece un claro signo del potencial de la justificación el que, justificando doblemente secuencias aleatorias hasta generar un máximo de 5000 soluciones, se pueda mejorar heurísticos de la literatura como el genético de Hartmann, que obtiene desv_CPM = 36.74% con ese mismo límite de secuencias.

A la luz de los dos experimentos se pueden extraer varias conclusiones que, en principio, son aplicables a las secuencias activas aleatorias. La justificación por elegibles produce, en general, peores resultados que la justificación por extremos. Como dato adicional podemos aportar que si justificamos cada secuencia aleatoria doblemente por elegibles aleatoriamente y contabilizamos la mejor de las secuencias calculadas en cada instancia, se obtiene desv_CPM = 41.50%, muy lejano de lo obtenido con la justificación por extremos aleatoria (38.36%). Sin embargo, emplear la justificación por elegibles con una regla apropiada puede producir mejores soluciones que utilizando la justificación por extremos. Esto se debe a la mayor diversidad en la justificación por elegibles.

Nuestra conjetura es que ocurrirá algo similar con la justificación general. La calidad media de las secuencias será peor que en la justificación por elegibles y habrá una mayor diversidad, pero las mejores soluciones – que a lo mejor resultan difíciles de conseguir – serán mejores que las alcanzables por las otras dos justificaciones.

La justificación por elegibles en un algoritmo de calidad

Como segundo ejemplo de aplicación de la justificación por elegibles hemos escogido el algoritmo de Hartmann, el que mejores resultados obtenía en el capítulo anterior al aplicarle la doble justificación (DJ). En esta ocasión, tras unas pruebas preliminares, hemos escogido, por proporcionar los mejores resultados, otra regla para decidir qué actividad justificar primero en la justificación a la derecha (izquierda): se particiona el conjunto de las actividades elegibles en subconjuntos, de acuerdo con el instante de tiempo en que puedan finalizar. De entre las actividades de los conjuntos con el primer o segundo instante más grande (pequeño), se escoge aquella actividad que actualmente finalice más tarde (comience más pronto). En la Tabla 4.3 se puede comparar el algoritmo sin justificación, el algoritmo con justificación por finales y el de justificación por elegibles práctica con esta regla.

	Σ	desv_UB	desv_CPM
Hartmann (1')	77746	4.41	37.00
Hartmann + DJ extremos	75616	1.74	33.24
Hartmann + DJ elegibles	75350	1.46	32.78

Tabla 4.3. Hartmann sin justificación y con justificación por extremos y por elegibles.

Como se puede observar el tercer algoritmo mejora al segundo en calidad, aunque la diferencia (casi 0.5 % en desv_CPM) no es comparable con la obtenida en el caso de las secuencias aleatorias. Además, el tiempo medio empleado (no mostrado porque no se intentó optimizar) es actualmente muy superior; de hecho, es bastante mayor que el de Hartmann + DJ con un límite de 10000 secuencias, que obtiene desv_CPM = 32.73%. En cualquier caso parece claro que las extensiones de la justificación por extremos merecen ser estudiadas, puesto que pueden dar lugar a mejores soluciones.

4. La traslación

Justificar a la derecha o izquierda una actividad i dentro de una secuencia significa moverla hasta el intervalo más extremo donde se pueda secuenciar, a la derecha si justificamos a la derecha y a la izquierda en el otro caso. Pero, en general. pueden existir otros intervalos posibles donde secuenciarla entre el intervalo de secuenciación de i en S y el más extremo. Si se permitiera escoger uno de esos intervalos se obtendría otro tipo de 'justificación', que de hecho ya no lo sería, y que, en principio, es más general. En este apartado vamos a definir formalmente este nuevo tipo, que denominaremos traslación, demostraremos que es estrictamente más general que la justificación y que es capaz de calcular todas las secuencias de SJD(S), por lo que es el tipo más general posible.

Definición: La traslación de una actividad

La **traslación a la derecha (izquierda) de una actividad i ≠ n (1) con movimiento** k ≥ 0 dentro de una secuencia posible S consiste en obtener la secuencia S' con $s_j' = s_j$ $\forall j \neq i$ y $s_i = s_j + k$ ($s_i = s_j - k$), con S' posible. Si S' no resulta posible se considera que no se puede trasladar esa actividad a la derecha (izquierda) con ese movimiento.

Definición: La traslación completa de una secuencia

La traslación a la derecha (izquierda) completa de una secuencia S consiste en obtener una secuencia S' posible y justificada a la derecha (izquierda) a partir

de aplicar a S un número finito de traslaciones a la derecha de actividades con movimientos mayores de 0.

Definición: El conjunto TDC(S) (TIC(S))

El conjunto de secuencias trasladadas completas a la derecha (izquierda) de S lo llamaremos **TDC(S)** (**TIC(S)**). Cuando no tengamos en cuenta la solución lo denotaremos TDC (TIC), y a las mejoras alcanzables mediante este concepto, MTCD (MTIC).

Por la definición se observa que la traslación generaliza la justificación, porque una justificación a la derecha (izquierda) de una actividad i es una traslación máxima, con $k^* = \max\{k: i \text{ se puede trasladar a la derecha (izquierda) con movimiento } k\}$.

PROPOSICIÓN 4.3

TDC $\subseteq$ SJD (TIC $\subseteq$ SJI).

Demostración

Evidente.

Esta proposición demuestra que la traslación comparte las buenas propiedades que poseía la justificación. En el siguiente apartado estudiaremos la relación entre ambas.

4.1. La relación entre la traslación y la justificación

La justificación de una actividad es una traslación máxima, lo que implica JD $\subseteq$ TDC (JI $\subseteq$ TIC). Las preguntas obligadas son si los dos conjuntos son iguales y si las mejoras por traslación se pueden obtener con la justificación. La siguiente proposición responde negativamente a las dos preguntas.

PROPOSICIÓN 4.4

JD $\subset$ TDC, MJD < MTCD (JI $\subset$ TIC, MJI < MTCI).

Demostración

Ya se ha comentado la inclusión, veamos el contraejemplo. En la demostración de que JD no era capaz de conseguir todas las secuencias de SJD hemos empleado el proyecto de la Figura 4.20 y la secuencia S de la Figura 4.21, activa para ese proyecto. La secuencia S", dibujada en la Figura 4.22, no se podía calcular mediante la justificación

general. Basta demostrar pues que sí se puede llegar a ella a través de la traslación. Pero esto es trivial, basta trasladar la actividad 6 a [6,7]. Esto crea un espacio en [5,8] donde trasladar la actividad 12, lo que a su a su vez permite trasladar 4 a [3,5]. Por último trasladamos 2 a [1,2], y llegamos a S'. Como S' es mejor que S, y con la justificación general a la derecha de S no se puede mejorar S, se demuestra que ni los conjuntos ni las mejoras son iguales. Q. E. D.

Esto demuestra que la traslación no es una generalización innecesaria de la justificación. De acuerdo con el resultado obtenido, la traslación puede conducir a regiones diferentes y a mejoras no alcanzables mediante la justificación. No sabemos si esta diferencia existente en la teoría será significativa en la práctica y si será posible explotar todo su potencial mediante algoritmos eficientes. Más adelante ofreceremos dudas razonables sobre esto último.

4.2. La traslación como caso más general posible

La traslación es una generalización de la justificación general que, a su vez, generaliza la justificación por elegibles y ésta, a la justificación por extremos. Tiene sentido preguntarse si existe una forma más general que la traslación de obtener secuencias justificadas a la derecha de una dada. En el siguiente teorema se demuestra que la traslación es suficiente para calcular todas las secuencias de SJD por lo que no es necesario ninguna generalización adicional.

TEOREMA 4.5

TDC = SJD (TIC = SJI).

Demostración

TDC $\subseteq$ SJD está claro. Veamos la otra inclusión. Sea S secuencia posible y S' $\in$ SJD(S). Tenemos que encontrar una manera de realizar traslaciones sobre S hasta obtener S'. Para ello construiremos una lista de actividades para la red inversa λ y demostraremos que trasladando las actividades en ese orden de una cierta manera se puede obtener S'.

Construimos λ ordenando las actividades en orden decreciente de sus inicios en S', i.e., inicio(i,S') > inicio(j,S') $\rightarrow$ i antes que j en λ, desempatando por número de actividad (por tener bien definida λ, pero se puede desempatar de cualquier manera sin que afecte a la demostración). Es obvio que λ es una lista de actividades para la red inversa.

Vamos a construir S' a partir de realizar traslaciones sobre S según λ. El movimiento que aplicaremos a la actividad i será de orden inicio(i,S') - inicio(i,S) (≥ 0 porque inicio(i,S') $\geq$ inicio(i,S)). Sea S_k la secuencia tras trasladar las primeras k actividades de λ. Demostraremos por inducción que S_k es posible, k = 1, ..., n.

S_1 es posible, porque en [inicio(n,S'),fin(n,S')] = [inicio(n,S),fin(n,S)] hay bastantes recursos para secuenciar n, ya que no la movemos. La hipótesis de inducción es que S_{k-1} es posible y queremos demostrar que S_k es posible.

Sea i la actividad de orden k en λ. Por construcción inicio(j,S_{k-1}) = inicio(j,S_k) $\forall j \neq i$. Luego sólo falta comprobar que en [inicio(i,S_k),fin(i,S_k)] hay suficientes recursos para secuenciar i. Si inicio(i,S_k) = inicio(i,S_{k-1}), no es necesario demostrar nada. Supongamos pues el caso contrario, que son distintos.

Por construcción, inicio(i,S_k) > inicio(i,S_{k-1}). En el intervalo [inicio(i,S_{k-1}),fin(i,S_{k-1})] $\cap$ [inicio(i,S_k),fin(i,S_k)] hay suficientes recursos para i por ser S_{k-1} posible. Luego sólo falta comprobar que en [A,fin(i,S_k)] existen suficientes recursos, donde A = max{fin(i,S_{k-1}),inicio(i,S')}. Para ello demostraremos que si t $\in$ [A,fin(i,S_k)[y j se secuencia en [t,t+1] en S_k, se secuencia también en [t,t+1] en S'. Como S' es posible, el teorema quedará demostrado.

Sean t, j tal que t $\in$ [A,fin(i,S_k)[$\cap$ [inicio(j,S_k),fin(j,S_k)[. Si orden(j,λ) $\leq$ orden(i,λ), j ha sido ya trasladada, por lo que inicio(j,S_k) = inicio(j,S') y, en consecuencia, j se secuencia en [t,t+1] en S'.

Si por el contrario orden(j,λ) > orden(i,λ) entonces, por definición de λ, inicio(j,S') $\leq$ inicio(i,S') $\leq$ A $\leq$ t. Por otra parte, fin(j,S') $\geq$ fin(j,S_k) $\geq$ t+1. Luego inicio(j,S') $\leq$ t < t+1 $\leq$ fin(j,S'), por lo que j se secuencia en [t,t+1] en S'.

Q.E.D.

COROLARIO 4.6

TDC(elegibles) = SJD = TDC (TIC(elegibles) = SJI = TIC), y sólo son necesarias n traslaciones para obtener cualquier secuencia de SJD (SJI).

Demostración

El vector λ de la demostración anterior es compatible con las relaciones de precedencia y tiene n elementos, uno por cada actividad. Q. E. D.

Es interesante que, en este caso, la traslación por elegibles sea equivalente a la traslación, algo que no ocurría con la justificación. Este resultado es útil desde el punto de vista práctico puesto que limita los algoritmos necesarios para trabajar con la traslación a los que trasladen por elegibles.

4.3. La traslación como problema de optimización

En este apartado realizaremos unos comentarios sobre la complejidad del problema de encontrar el mejor elemento de SJD(S) para una secuencia S. Para ello definiremos el problema de optimización asociado y demostraremos que es equivalente, en cuanto a complejidad, a un problema que, de forma natural, se puede ver como un RCPSP con fechas de disponibilidad.

Definición: Problema de la Traslación Completa

Denominaremos **Problema de la Traslación Completa** (PTC) al problema de, dada una secuencia S, hallar una secuencia S' ∈ TDC(S) de longitud mínima.

De forma análoga se puede definir el problema de la justificación para sus diferentes variantes.

Por el teorema del apartado anterior se tiene que

$$\left.\begin{array}{ll}\text{Min} & T(S') \\ \text{s.a.} & S' \in TDC(S)\end{array}\right\} \equiv \left.\begin{array}{ll}\text{Min} & T(S') \\ \text{s.a.} & S' \in SJD(S)\end{array}\right\} \equiv \left.\begin{array}{ll}\text{Min} & T(S') \\ \text{s.a.} & S' \text{ posible y justificada a la derecha} \\ & T(S) - d_i \geq s_i' \geq s_i \quad \forall i\end{array}\right\}.$$

Todos estos problemas se pueden considerar el PTC. En ninguno se exige $s_0' = 0$, por lo que T(S') se calcula como $f_n' - s_0'$, al igual que en el resto de problemas que consideraremos. Es obvio que la solución óptima del PTC no proporciona, en general, la solución óptima del RCPSP.

Veamos que el PTC es NP-duro si se permite que la secuencia inicial S no sea activa. Para cada actividad i predecesora directa de n construimos una solución S_i en la que i finaliza en $2\sum_{i=2}^{n-1} d_i$ y el resto de actividades se secuencia consecutivamente en orden topológico, comenzando en 0. Obviamente esas n-2 actividades se secuencian en las primeras $\sum_{i=2}^{n-1} d_i$ unidades, dejando las siguientes $\sum_{i=2}^{n-1} d_i$ unidades sólo con i. Al resolver el PTC asociado a S_i obtenemos el óptimo para el RCPSP en que se exige que max{f_j / j ∈ **V**} = f_i. Claramente el óptimo del RCPSP original es la mejor de las secuencias

obtenidas al resolver los PTC asociados a cada S_i, por lo que el PTC es NP-duro al serlo el RCPSP. No podemos utilizar una solución donde se fije simplemente $f_n = 2\sum_{i=2}^{n-1} d_i$ porque, por definición, hemos impuesto $f_n = \max\{f_i \,/\, i \in Pred_n\}$.

Este argumento no es sencillo realizarlo si la solución inicial debe ser activa, por lo que intentaremos equiparar este problema con uno conocido. Para ello vamos a introducir una serie de definiciones y propiedades.

Definición: La traslación no completa

La traslación a la derecha (izquierda) de una secuencia S consiste en obtener una secuencia S' posible a partir de aplicar a S un número finito de traslaciones a la derecha (izquierda) de actividades con movimientos mayores de 0.

Definición: El conjunto TD(S) (TI(S))

El conjunto de secuencias trasladadas a la derecha (izquierda) de S lo llamaremos **TD(S) (TI(S))**. Cuando no tengamos en cuenta la solución lo denotaremos TD (TI), y a las mejoras alcanzables mediante este concepto, MTD (MTI). Fijémonos que la única diferencia con la traslación completa es que las secuencias obtenidas no tienen por qué ser justificadas a la derecha.

PROPOSICIÓN 4.7 (análoga para la izquierda)

Dada una secuencia posible S y $S' \in TD(S)$, se cumple:

1) $s_i' \geq s_i$, $f_i' \leq T(S)$ $\forall i$.

2) $T(S') \leq T(S)$.

3) $TDC \subset TD$, lo que implica $MTDC \leq MTD$.

4) Existe $S'' \in TDC(S)$ de manera que $s_i'' \geq s_i'$ $\forall i$ y $T(S'') \leq T(S')$, en particular $MTC(S) = MT(S)$ $\forall S$, por lo que $MTDC = MTD$.

Demostración

Las propiedades 1) – 3) son obvias por la definición. En 4), la solución S" se puede obtener justificando por finales (o por cualquier otro método de justificación o traslación completa) S'. Q. E. D.

La demostración del teorema del anterior apartado puede servir para demostrar que TD(elegibles) = TD = STD, donde STD es el conjunto de las secuencias trasladadas

de una dada, y se diferencia de SJD únicamente en que sus elementos no son necesariamente justificados a la derecha. También podemos definir el problema de la traslación (PT), que teniendo en cuenta que TD = STD se puede escribir como

$$\left.\begin{array}{lc} \text{Min} & T(S') \\ \text{s.a.} & S'\,\text{posible} \\ & T(S) - d_i \geq s_i' \geq s_i \quad \forall i \end{array}\right\} (PT).$$

Como hemos comentado, SJD(S) ⊆ TD(S), por lo que todas las soluciones posibles de PTC lo son de PT, con lo que min(PT) ≤ min(PTC). Pero hemos demostrado también que ∀ S' ∈ TD(S) existe una secuencia S'' ∈ TDC(S) de manera que T(S'') ≤ T(S'). Eso implica que todas las soluciones de PT tienen asociada una solución de PTC de cómo máximo la misma longitud, en particular las óptimas. Eso se traduce en que min(PT) = min(PTC) (lo que hemos denotado en la preposición anterior como MTDC = MTD). No sólo eso, sino que con una solución óptima de un problema tenemos una solución óptima del otro, para pasar de PTC a PT es quedarse con la misma solución, y para pasar de PT a PTC basta con justificar por finales esa solución. Todo esto implica que ambos problemas comparten la misma dificultad algorítmica.

Las restricciones $T(S) - d_i \geq s_i'$ son superfluas en el problema (PT) para todo i distinto de n, porque ya están implícitas al exigir que S' sea posible. Esto deja la única cota superior de T(S) para s_n', y ésta es equivalente a $s_0 = 0$ en el RCPSP (porque también tenemos $s_n' \geq s_n = T(S)$). Esto implica que PT es un RCPSP con cotas inferiores o, lo que es lo mismo, un RCPSP con fechas de disponibilidad, que es un problema NP-duro. Esto parece implicar claramente que es NP-duro, pero no deja de ser una conjetura (con bastantes visos de ser cierta). Esto implicaría que el PTC sería también NP-duro, y quedaría por determinar la dificultad del problema de justificación en sus distintas versiones. Que estos problemas sean (o al menos parezcan ser) NP-duros implica que no vamos a ser capaces de emplear todo su potencial resolviéndolos exactamente, por lo que será necesario desarrollar heurísticos eficaces y muy rápidos, puesto que, en principio, queremos aplicarlos a muchas secuencias.

5. TÉCNICAS RELACIONADAS CON LA TRASLACIÓN/JUSTIFICACIÓN

Reiteración de la justificación/traslación

Ya hemos comentado en el anterior capítulo que es posible justificar una secuencia que ya haya sido justificada varias veces. Así, se puede aplicar la doble justificación a

una secuencia que es el resultado de otra doble justificación. Es interesante preguntarse si esta reiteración de la (doble) justificación es útil, o si se puede conseguir todas las mejoras de una secuencia con una única (doble) justificación. El siguiente ejemplo demuestra que con la reiteración se puede lograr más mejoras que con la aplicación una sola vez de la doble justificación/traslación.

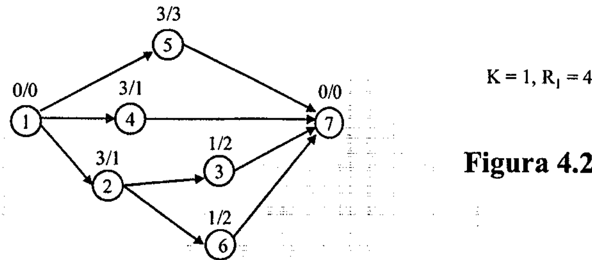


Figura 4.24

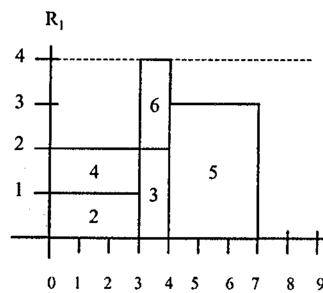


Figura 4.25

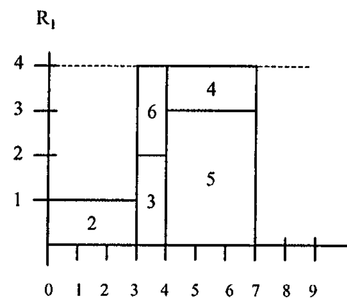


Figura 4.26

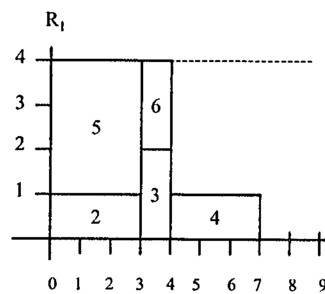


Figura 4.27

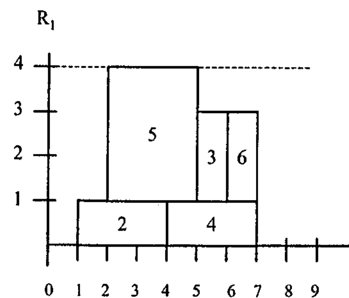
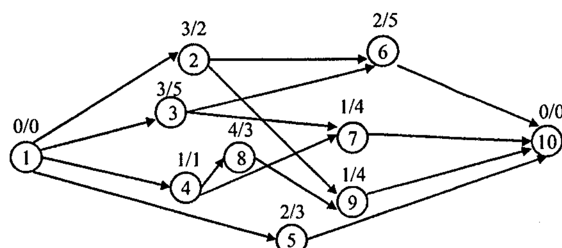


Figura 4.28

La secuencia S de la Figura 4.25 es posible y activa para el proyecto de la Figura 4.24. Es evidente que JD(S) = SJD(S) = {S'}, con S' la solución de la Figura 4.26, dado que 4 es la única actividad no fija. También es obvio que JI(S') = SJI(S') = {S, S''}, con S''

la secuencia de la Figura 4.27. En S', 4 y 5 son las únicas actividades no justificadas a la izquierda y, dependiendo de cuál se elija para justificar primero, se llega a S o a S''. Esto implica que ni mediante la doble justificación ni mediante la doble traslación se puede mejorar S. Tampoco se puede mejorar empleando la (doble) traslación no completa. Sin embargo, si ahora justificamos S'' por finales, mediante el vector (7 4 6 3 5 2 1) se obtiene S''', la solución de la Figura 4.28, una unidad menor que S.



K = 1, R_1 = 8

Figura 4.29

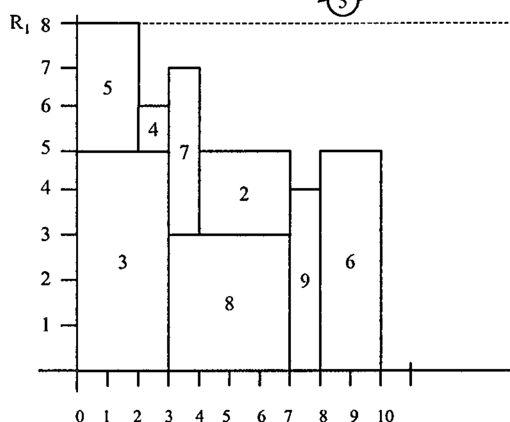


Figura 4.30

Consideremos ahora el proyecto de la Figura 4.29 y la secuencia de la Figura 4.30, activa en este proyecto. Veamos primero que SJD(S) = {S'}, con S' la solución de la Figura 4.31. Las actividades 2, 4, 6, 8 y 9 están fijas y sólo hay que considerar 3, 5 y 7. En [8,10] sólo cabe la actividad 5 y en [7,8] sólo la 7 (sin tener en cuenta la 5); una vez relizados estos cambios la actividad 3 es la única no justificada a la derecha y, por tanto, para obtener una secuencia justificada a la derecha tenemos que moverla a [1,4], hasta obtener S'. Supongamos que queremos trasladar o justificar S'. Dado que S' no comienza realmente en 0, y que es una unidad menor que S, tiene sentido trabajar con (S')* = S' – s_0' = S' – 1. De otro modo, al trasladar/justificar podríamos llegar a secuencias que tuvieran una duración de T(S') + 1 (por ejemplo podríamos llegar a S), lo cual rompería la condición que cumplen todos los movimientos que realizamos, de no incrementar las longitudes de las soluciones originales. Si, como parece lógico y natural, trabajamos con (S')*, tenemos que SJI((S')*) = {S''}, la secuencia de la Figura 4.32, de duración igual que (S')*. Por otra parte, la secuencia S''' de la Figura 4.33 pertenece al conjunto STD(S) (los mismos movimientos que de S a S' salvo con la actividad 3, que no la justificamos), y la secuencia $S^{(iv)}$ (Figura 4.34)

pertenece a SJI(S''') (2 a [0,3], 5 a [3,5], 6 a [5,7]). $S^{(iv)}$ es de menor longitud que S'' y S', lo que quiere decir que pasando por secuencias trasladadas no completas se pueden obtener mejoras que no se logran con las completas al trasladar reiteradamente. Esto no ocurriría si se trabajara con S' en lugar de (S')*.

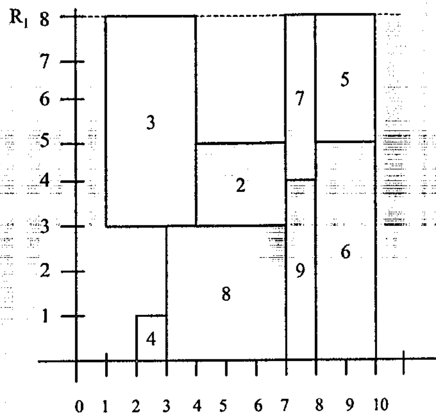


Figura 4.31

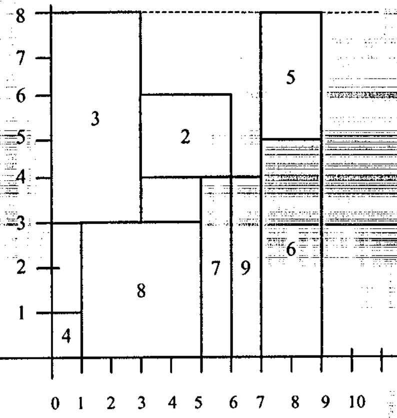


Figura 4.32

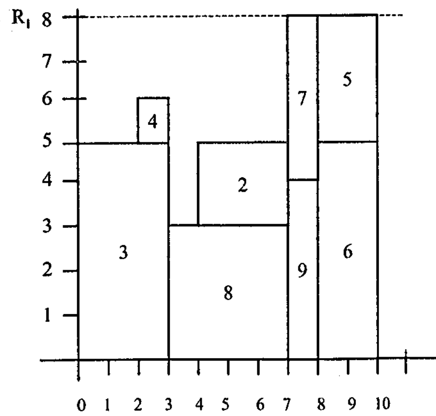


Figura 4.33

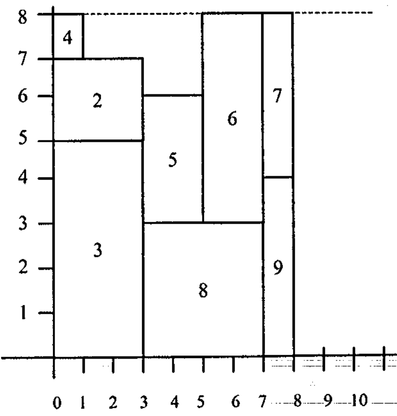


Figura 4.34

Justificar en [0,T(S)+k]

Del ejemplo anterior se desprende otra propiedad muy interesante. Si al trasladar/justificar una secuencia a la derecha (izquierda) se fija $f_n' = T(S) + k$ ($s_0' = -k$) con $k > 0$, se pueden obtener soluciones mejores que si, como hasta ahora, se toma $k = 0$. Eso es lo que pasa con $(S')^*$. Si el k se toma igual a 1 es como si se trasladara/justificara sobre $T(S) = T((S')^*) + 1$ y en ese caso se puede llegar a $S^{(iv)}$.

De hecho, si definimos el conjunto $SJD(S;k) = \{S'$ posible, justificada a la derecha/ $f_n' = T(S)$, $s_i' \geq s_i - k\}$ (que es equivalente a $\{S'$ posible, justificada a la derecha/ $f_n' = T(S) + k$, $s_i' \geq s_i\}$), se cumple que $SJD(S;k) \subseteq SJD(S;k+1)$ $\forall$ k. Esto es así porque las restricciones sobre los inicios son menos restrictivas cuanto más grande sea el k. Y lo mismo ocurre con la justificación a la izquierda.

Además, $\forall S$ $\exists k$ / $SJD(S;k)$ $(SJI(I,k))$ contiene una solución óptima. Basta tomar $k = T(S)$, aunque en general el k mínimo será mucho más pequeño. Obviamente, en la práctica no se va a emplear un k grande dado que, como contrapartida, las secuencias obtenidas sólo aseguran $T(S') \leq T(S) + k$, pero k pequeños (por ejemplo 1, 2 ó 3) se pueden combinar con reglas adecuadas de traslación/justificación para lograr resultados muy buenos y/o formar parte de heurísticos. Fijémonos que, por ejemplo, puede servir como método de muestreo capaz de obtener soluciones diversas que no excedan en duración k unidades de una dada para cualquier $k \geq 0$, algo no existente en la actualidad.

Un dato adicional interesante es que la justificación por extremos es indiferente a estos cambios de k, dado que JD(S,finales;k) = JD(S,finales) (JI(S,inicios;k) = JI(S,inicios)) $\forall$ S, $k \geq 0$. Esto es consecuencia de que justificar por extremos es lo mismo que secuenciar en un cierto orden.

La justificación en soluciones imposibles

Otra posibilidad es que el k sea negativo, aunque eso supone trabajar con soluciones imposibles e intentar trasladar/justificar actividades para que se conviertan en posibles.

Algo similar ocurre si justificamos una secuencia suponiendo que existen en cada unidad de tiempo (o en determinadas unidades) más recursos disponibles de los que realmente hay. Un ejemplo simple es fijar $K_r = K_r + 1$ $\forall r$ durante la justificación a la derecha y volver a la disponibilidad original al justificar a la izquierda.

Esto puede ser útil para romper estructuras fijas que no se mueven al justificar de la forma usual y, especialmente, para diversificar, para obtener secuencias que no hayan

aparecido en el algoritmo. En sucesivas justificaciones se puede intentar convertir en solución posible esa solución imposible, o se puede construir una solución posible aplicando Serie a un vector por inicios de esa solución.

Un ejemplo mucho más controlado de esta técnica se da cuando al justificar a la derecha una actividad i no se puede justificar hasta [a,b] (y si no se justifica ahí se queda fija) únicamente porque en [b-1,b] no cabe. Temporalmente se puede relajar las restricciones en esa unidad y justificar i en [a,b]. Tras calcular la secuencia 'justificada a la derecha' S', imposible en [b-1,b], se intenta convertirla en posible en la justificación a la izquierda, buscando que se pueda mover alguna actividad de [b-1,b].

Secuencias mejorables y no mejorables

Definición: Secuencia mejorable

> Una secuencia S es **mejorable a la derecha** (**izquierda**) si existe S' ∈ SJD(S) (SJI(S)) / T(S') < T(S). De un modo análogo se puede definir mejorable por extremos, por elegibles y por justificación general a derecha y a izquierda.

Según los diferentes experimentos que hemos realizado en este y, especialmente, en el capítulo anterior hay un número muy grande de secuencias que son mejorables a la derecha e izquierda. Como ejemplo de secuencias no mejorables a la izquierda (derecha) están todas las secuencias activas (a la derecha). Obviamente, existen muchas secuencias que son activas (a la derecha) y no son mejorables a la derecha (izquierda). Sería muy interesante poder trabajar exclusivamente con este tipo de soluciones, ya que es seguro que contienen un óptimo del problema. Al igual que Serie es un procedimiento capaz de producir secuencias activas, sería potencialmente muy útil disponer de uno que calculara soluciones activas no mejorables a la derecha (al menos no mejorables por extremos).

6. La justificación en otros problemas de secuenciación de proyectos con recursos limitados

En los anteriores apartados, y en el capítulo anterior, hemos estudiado la justificación en el RCPSP, desde un punto de vista tanto teórico como práctico. Teniendo en cuenta los excelentes resultados obtenidos al aplicar esta técnica, es natural preguntarse si es posible emplearla en problemas de secuenciación de proyectos similares al RCPSP. La respuesta es afirmativa y, de hecho, ya se ha empleado puntualmente la justificación y, más extensamente, técnicas de secuenciación hacia

atrás en problemas de multiproyecto (Lova et al., 2000), con diferentes funciones objetivo (Özdamar y Ulusoy, 1996, tenían el doble objetivo de mejorar la duración del proyecto y el NPV; Smith-Daniels y Aquilano, 1987, intentan optimizar el NPV; Lova et al., 2000, manejan varios objetivos simultáneamente, tanto temporales como no temporales) y/o distintas restricciones (Li y Willis, 1992, recursos no renovables; Özdamar, 1999 y Alcaraz, 2001, multimodo; Neumann y Zimmermann, 1998, restricciones generalizadas de tipo máximo). Pero estas aplicaciones, y especialmente las de la justificación, son igual de aisladas, en general, que en el RCPSP y muchos heurísticos no emplean estas técnicas en la mayoría de las extensiones del RCPSP.

Nosotros opinamos que la justificación y la secuenciación hacia atrás pueden ser muy útiles en este tipo de problemas. De hecho, la aplicación de la justificación es directa en un número importante de ellos, puesto que las soluciones son secuencias de un cierto RCPSP y se pueden justificar de exactamente la misma forma que en el RCPSP. Esto quiere decir que, si la función objetivo es la misma, las mejoras secuencia a secuencia son las mismas que en el RCPSP, por lo que esos problemas se podrán beneficiar del gran potencial de la justificación, al menos si se escoge adecuadamente el algoritmo heurístico donde se utilice.

En otros problemas habrá que adaptar significativamente la forma en que se aplica la justificación pero, aun así, puede resultar beneficioso emplear esta técnica.

A continuación, veremos unos ejemplos donde la aplicación directa de la justificación es inmediata. Además, estudiaremos cómo adaptar esta técnica considerando las características específicas de cada problema. Esto proporcionará una justificación, a priori, más eficaz. También realizaremos comentarios sobre cómo modificar algunos de los algoritmos heurísticos estudiados para resolver estos problemas.

6.1. El RCPSP con interrupción

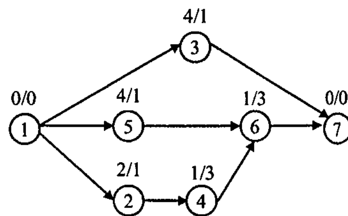
La primera extensión del RCPSP que trataremos es la que permite interrumpir las actividades. En particular, consideraremos que la secuenciación de cada actividad puede ser interrumpida como máximo una vez, i.e., una actividad puede ser secuenciada en dos partes, siempre y cuando cada una de ellas tenga una duración entera (no determinada a priori). Denominaremos **RCPSP_inter** al problema así definido. La adaptación de la justificación que consideraremos se podría generalizar o modificar fácilmente a otros casos de interrupción:

a) Sólo las actividades de un cierto subconjunto de V, V', se pueden interrumpir, mientras que el resto deben ser secuenciadas ininterrumpidamente.

b) Cada actividad i se puede interrumpir como máximo un cierto número de veces prefijado n_i, cada una de las partes con un número entero de unidades de duración.

c) Se prohibe, a priori, algunas de las posibles duraciones que pueden tomar las diferentes partes de las actividades si se interrumpen. Así, una actividad de 5 unidades puede estar prefijado que se pueda partir en 2+3 ó 3+2, pero no en 1+4 ni 4+1. Todas estas restricciones pueden venir dadas por las particularidades de cada actividad, dadas por un condicionante práctico real (por ejemplo, que no sea rentable en ningún caso procesar únicamente una unidad de una actividad).

Sea **P** una instancia concreta del RCPSP_inter. Cada solución S de P debe reflejar las actividades interrumpidas y proveer de un inicio para las actividades que se ejecutan de forma ininterrumpida, así como de un inicio para cada parte y las duraciones correspondientes en el caso de que se ejecuten con interrupción. Una manera de reunir la información de la secuencia S de la Figura 4.36, posible para el proyecto de la Figura 4.35, sería dar el conjunto de actividades interrumpidas, Interr = {3}, un inicio para cada actividad no interrumpida, $s_2 = 0$, $s_4 = 2$, $s_5 = 3$, $s_6 = 7$, y un inicio para cada parte de la actividad interrumpida y su duración (es suficiente con la duración de la primera parte), $s_{3a} = 5$, $s_{3b} = 8$, $d_{3a} = 2$, $d_{3b} = 2$. (Denotaremos por ia e ib a la primera y segunda parte de una actividad i).



K = 1, R_1 = 3

Figura 4.35

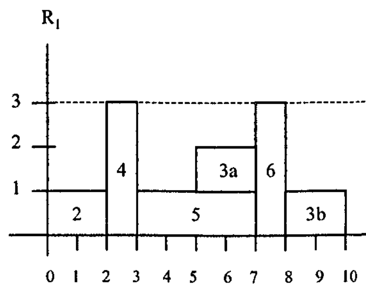


Figura 4.36

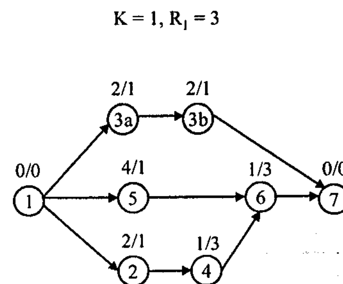


Figura 4.37

6.1.1. Justificación directa

Dada una secuencia S de P, existe un RCPSP (V',A') para el que S es una secuencia posible, el que tiene las mismas restricciones de recursos que P y cada actividad o parte de actividad de S da lugar a una actividad de V', con las mismas necesidades de recursos. Las relaciones de precedencia en A' son las naturales, una actividad i es predecesora de j si i es la última o única parte de una actividad en V que es predecesora de una actividad de la que j es la primera o única parte. Además, se añade una relación entre la primera y la última parte de cada actividad de V interrumpida. Todas las secuencias posibles para este problema lo son para P, pero el recíproco no es cierto y el óptimo, en general, tampoco será el mismo.

Ejemplo

Consideramos de nuevo la secuencia S de la Figura 4.36. El RCPSP que le asociamos a S es el que se basa en el grafo de la Figura 4.37, donde la actividad 3 (única interrumpida en S) se ha dividido en dos (con las duraciones correspondientes a S) y entre ellas se ha creado una relación de precedencia. Las relaciones de precedencia que llegaban a 3 llegan ahora a 3a ($1 \rightarrow 3a$) y las que partían de 3 lo hacen ahora de 3b ($3b \rightarrow 7$), mientras que el resto de relaciones no se modifican. Las restricciones de recursos tampoco cambian, basta con dotar a las dos partes de 3 con las necesidades de 3.

Así pues, cualquier secuencia S posible para el RCPSP_inter es una secuencia posible para un cierto RCPSP y se puede justificar según ese RCPSP, obteniéndose otra secuencia posible para P. Esto quiere decir que el RCPSP_inter puede beneficiarse, secuencia a secuencia, de las mejoras que provoca la justificación que, como se ha demostrado, son amplias.

No obstante, si queremos aplicar la justificación en un algoritmo diseñado específicamente para el RCPSP_inter no es preciso generar para cada una de las secuencias el RCPSP asociado. Basta con utilizar una de las dos siguientes codificaciones, que son capaces de reflejar las interrupciones y sus duraciones.

La primera posibilidad sería utilizar una lista de actividades '**doble**', i.e., un vector de cómo máximo 2n elementos donde cada actividad aparece 1 ó 2 veces dependiendo de si está interrumpida o no, pero siempre después de la última aparición de sus predecesoras. Además, sería necesario otro vector de n elementos, con vector(i) = d_{ia} si i está interrumpida y vector(i) = d_i si no lo está. En el primer caso la duración d_{ib} se puede calcular mediante $d_i - d_{ia}$.

Otra codificación apropiada, bastante atractiva para un genético, emplearía listas de actividades **'dobles puras'**, de exactamente 2n componentes. Cada actividad aparecería dos veces, y las actividades que en un principio no se interrumpen se colocarían de forma consecutiva. Además, se emplearía el mismo vector de las duraciones que en la anterior codificación. Cuando Serie encuentra una parte con duración igual a la original sabe que no tiene que considerar la siguiente actividad, ya que es ficticia. Es obvio que con Serie y una de estas dos codificaciones se puede obtener una solución óptima para cualquier instancia.

6.1.2. Justificación adaptada al problema

Al justificar a la derecha una secuencia S del RCPSP_inter según el RCPSP (V',A') se obtiene S', una secuencia justificada a la derecha para ese RCPSP pero, ¿se puede decir que es una secuencia justificada a la derecha para el RCPSP_inter, i.e., teniendo en cuenta las nuevas restricciones? Para plantearnos siquiera este interrogante debemos definir este concepto para la secuenciación con interrupción o, al menos, ver si la definición que conocemos es válida. Recordemos que una actividad i está justificada a la izquierda (derecha) en S si está secuenciada lo más pronto (tarde) posible, siendo S posible. Secuenciar una actividad lo más pronto (tarde) posible es equivalente en el RCPSP a exigir que comience o termine lo antes (después) posible. Estas tres exigencias (secuenciar, empezar y finalizar) son la misma en el RCPSP, pero no así en otros problemas de secuenciación de proyectos. En el RCPSP_inter existen diferentes posibilidades de definir qué es o cuándo está una actividad justificada a la izquierda (derecha). Nosotros hemos escogido una definición que asegura que no existe otra manera de secuenciar la actividad que comience y finalice antes.

Definición: Actividad justificada a la izquierda en el RCPSP_inter

> Sea S una secuencia e i una actividad con inicio (de la primera parte si está interrumpida) s_i y final (de la segunda parte si está interrumpida) f_i. La actividad i está justificada a la izquierda en S en el RCPSP_inter si no existe una manera de secuenciar i en una o dos partes de manera que $s_i' \leq s_i$ y $f_i' < f_i$ o $s_i' < s_i$ y $f_i' \leq f_i$.
>
> De un modo análogo se define una actividad justificada a la derecha en el RCPSP_inter.

Justificar a la izquierda una actividad i o comprobar si lo está puede ser relativamente costoso porque, en principio, puede ser necesario considerar todas las posibles

interrupciones. A esta justificación la denominaremos **justificación total**. En la práctica puede ser más interesante utilizar el siguiente concepto.

Definición: Parte de una actividad justificada a la izquierda (derecha) en el RCPSP_inter

Sea S una secuencia e ix la primera (segunda) parte de una actividad i. La parte ix está justificada a la izquierda (derecha) si lo está en el RCPSP al considerarla como una actividad aislada.

Sea ix la segunda (primera) parte de i. La parte ix está justificada a la izquierda (derecha) si lo está en el RCPSP al considerarla como una actividad aislada y no se puede secuenciar ninguna unidad de ix al final (principio) de la primera (segunda) parte.

Está definición es adecuada para lo que significa 'justificado a la izquierda', puesto que se refiere a una parte. Cuando la primera parte de i, ia, está justificada a la izquierda, la definición asegura que no existe ninguna manera de interrumpir la actividad de manera que la nueva primera parte (ia)' comience antes que ia y la contenga, i.e., la duración de (ia)' sea mayor que la de ia. Cuando la segunda parte de i, ib, está justificada a la izquierda, la definición asegura que cada unidad de i está secuenciada lo más a la izquierda posible si fijamos el inicio de la primera parte. Fijémonos que pueden existir interrupciones de i con las que comience y termine antes que una en la que ambas partes estén justificadas a la izquierda.

Mediante la definición de parte justificada podemos construir otro tipo de justificación, que denominaremos **justificación rápida**:

- Al justificar a la derecha (izquierda) una actividad ininterrumpida se busca el último (primer) instante de tiempo donde existan suficientes recursos para secuenciar i, y se secuencia antes (a partir) de ese instante todas las unidades de i que quepan consecutivamente. Si i no cabe entera, las siguientes unidades se secuencian lo más tarde (pronto) posible, todas consecutivas, puesto que no es posible interrumpir de nuevo. Esta es una manera sencilla y rápida de aprovechar la posibilidad de la interrupción para secuenciar después (antes) i, y las partes obtenidas están justificadas a la derecha (izquierda).

- Cuando una actividad está interrumpida en la solución original, al justificar a la derecha (izquierda) la segunda (primera) parte de una actividad, la trataríamos como en el RCPSP, por lo que sería fácil justificarla. Justificar la primera (segunda) parte ia (ib) es un poco diferente al RCPSP, ya que hay que calcular si inmediatamente antes (después) de la segunda (primera) parte ib (ia) se puede

secuenciar alguna unidad de ia (ib), y secuenciar todas las unidades que se pueda. En el caso en que no se pueda secuenciar ninguna unidad, simplemente se justifica ia (ib) como en el RCPSP. Sin embargo, hay que añadir otro matiz, cuando toda ia (ib) se puede secuenciar junto a ib (ia), produciéndose la fusión completa de las dos. En ese caso volvemos a tener una actividad ininterrumpida, que es necesario intentar justificar de nuevo a la derecha (izquierda).

Cuando justifiquemos una actividad o una secuencia con esta justificación, la denominaremos actividad o secuencia rápidamente justificada.

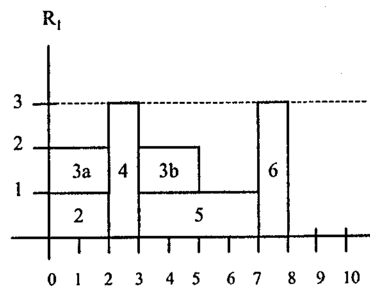


Figura 4.38

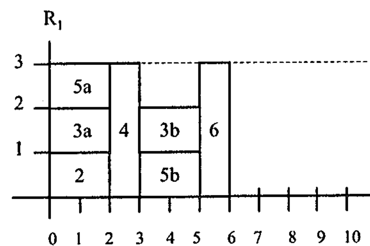


Figura 4.39

Ejemplos

(1) En la secuencia S de la Figura 4.38, la actividad 5 está justificada a la izquierda según el RCPSP correspondiente (grafo Figura 4.37). Pero los condicionantes de este problema permiten secuenciarla más a la izquierda. Al aplicarle la justificación rápida (y justificar después 6 a la izquierda) obtenemos la secuencia S' de la Figura 4.39, óptima para el problema.

(2) En la secuencia de la Figura 4.41, posible para el problema dado en la Figura 4.40, todas las actividades están justificadas a la izquierda de acuerdo con la definición de la justificación rápida, exceptuando 3b. Esta actividad no se podría mover más a la izquierda si justificáramos como en el RCPSP, considerándola una actividad. Sin embargo, con la justificación rápida sí puede moverse a la izquierda, puesto que 3a finaliza en 4 y en [4,5] existen suficientes recursos para secuenciarla. Al

secuenciar una unidad de 3b en ese intervalo (lo que se corresponde con interrumpir 3 de otra manera) también podemos adelantar la segunda unidad de 3b, llegando a la secuencia de la Figura 4.42, una unidad inferior que la original.

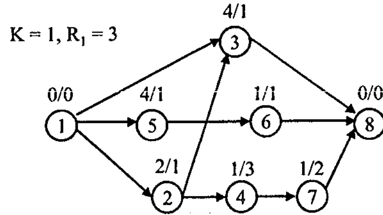


Figura 4.40

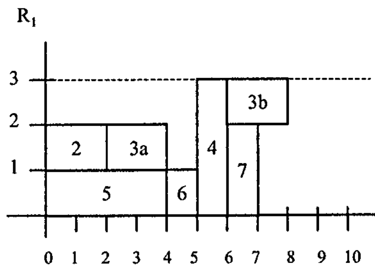


Figura 4.41

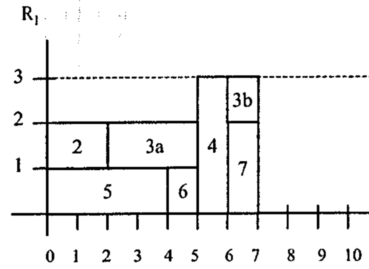


Figura 4.42

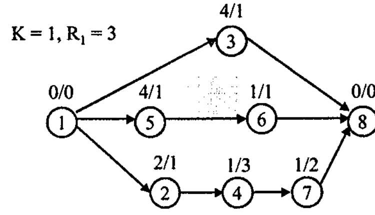


Figura 4.43

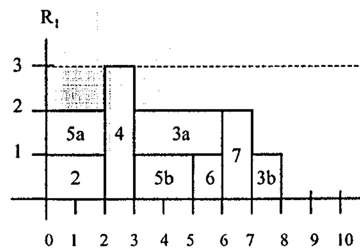


Figura 4.44

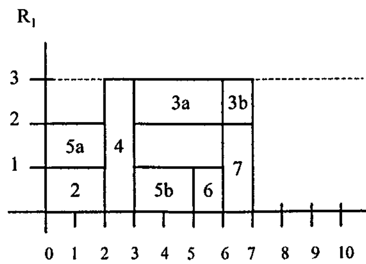


Figura 4.45

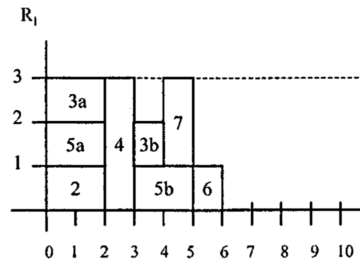


Figura 4.46

(3) En este ejemplo trabajaremos con la justificación rápida. La secuencia de la Figura 4.44 es posible en el proyecto dado en la Figura 4.43. Todas las actividades están justificadas a la izquierda exceptuando la 3b. Al justificar esta actividad llegamos a la secuencia S' de la Figura 4.45. Las partes a y b de 3 están juntas, por lo que en

realidad no está interrumpida. Teniendo en cuenta la nueva definición la actividad 3 se puede justificar más a la izquierda, concretamente hasta secuenciarla en [0,2] y [3,4]. Esto repercute en que 7 pase de estar justificada a la izquierda a no estarlo. Tras justificarla llegamos a la secuencia de la Figura 4.46, que es 2 unidades menor que S.

Tanto la justificación total como la rápida aplicada sobre las actividades de una secuencia pueden llevar a una secuencia justificada a la derecha o izquierda, en el sentido que cada actividad esté justificada de acuerdo con la definición correspondiente.

Es obvio que aplicando tanto la justificación rápida como la justificación total es posible emplear mejor los recursos actividad a actividad que aplicando la justificación directa del RCPSP por lo que, en general, conducirán a mejores secuencias. Sería posible que partiendo de una secuencia S y realizando una justificación directa se obtenga mejor resultado que con la justificación rápida o total. No obstante, una secuencia justificada a la izquierda (derecha) según el RCPSP se puede mejorar en ocasiones justificando a la izquierda (derecha) según el RCPSP_inter de cualquiera de las dos formas. El recíproco no es cierto (Figuras 4.38 y 4.39), ya que toda secuencia justificada a la izquierda (derecha) según la justificación total o la rápida lo es también según el RCPSP correspondiente.

La definición de actividad rápidamente justificada a la izquierda (derecha) en el RCPSP_inter no sólo es válida para justificar, sino también para secuenciar. Mediante esta definición se puede crear un método de generación de secuencias para P, **Serie_inter**, que calcule secuencias rápidamente justificadas a la izquierda (derecha). Este procedimiento procedería a secuenciar cada actividad lo más pronto (tarde) posible de acuerdo con la nueva definición, teniendo en cuenta si la actividad es una parte o está ininterrumpida, y si al secuenciarla se fusiona con su otra parte, en cuyo caso debe ser considerada de nuevo (si logra justificarse, otras actividades pueden pasar de justificadas a no justificadas, por lo que se deberían considerar de nuevo). De este modo aparecerán de forma natural, según lo que convenga en cada secuencia e instancia, nuevas interrupciones, y desaparecerán aquellas que sean innecesarias porque se fusionen las partes. Además, Serie_inter es mejor que Serie actividad a actividad puesto que, en ocasiones, es capaz de llevarla más a la izquierda (derecha), mientras que el recíproco no es cierto. También es mejor globalmente, en el sentido de que puede reducir la longitud de una secuencia calculada con Serie, mientras que de nuevo el recíproco no es cierto. Mediante la definición de actividad completamente justificada también se puede crear un nuevo método de generación de secuencias.

Por otra parte, el conjunto de secuencias activas (justificadas a la izquierda con una u otra justificación) para el RCPSP_inter, **AS_inter**, conserva las propiedades más importantes que se cumplen en el RCPSP: para toda solución S del RCPSP_inter existe una solución S' de AS_inter de manera que $T(S') \leq T(S)$ y $s_i' \leq s_i$, $f_i' \leq f_i$ $\forall i$. En particular, AS_inter contiene siempre una solución óptima. Al igual que en el RCPSP, lo más importante es que se puede calcular (más fácilmente en el caso de la justificación rápida) una de esas S' mediante la justificación. El conjunto de secuencias activas a la derecha cumple propiedades similares.

Análogamente a lo que ocurre con Serie y el RCPSP, con Serie o Serie_inter y las dos codificaciones vistas en el apartado anterior – las listas de actividades dobles y dobles puras – se pueden codificar todas las soluciones justificadas a la izquierda para el RCPSP_inter. Esto conlleva que ambos métodos (Serie y Serie_inter) aplicados a estas codificaciones pueden obtener una solución óptima en cualquier instancia. Serie_inter aplicado a las listas de actividades 'simples' (las empleadas en el RCPSP) no es capaz de obtener el óptimo para cualquier P. Sin embargo, esta combinación listas de actividades + Serie_inter puede servir para ofrecer soluciones de buena calidad en un heurístico, al menos para ofrecer soluciones iniciales, con la ventaja añadida de que algunas actividades estarán interrumpidas.

Teniendo en cuenta que, actividad a actividad Serie_inter y la justificación rápida son mejores que Serie y la justificación del RCPSP, respectivamente, parece que los algoritmos que utilicen los primeros pueden obtener mejores resultados. Una ventaja adicional es que, con la nueva justificación o Serie_inter, las interrupciones surgen de forma espontánea, sin tener que forzarlas. Esto simplifica enormemente la adaptación de algoritmos heurísticos del RCPSP al RCPSP_inter.

6.1.3. Pruebas computacionales de la justificación en el RCPSP_inter

Hemos realizado un experimento para comprobar que la justificación es efectiva en este problema. En particular, hemos construido dos algoritmos basados en Serie_inter. El primero, Aleat_inter, calcula listas de actividades dobles puras y las secuencia mediante Serie_inter. El segundo, Hartmann_inter, es una adaptación simple del genético de Hartmann para el RCPSP_inter. A estos algoritmos les hemos añadido la doble justificación rápida, obteniendo Aleat_inter + DJ y Hartmann_inter + DJ. Los cuatro algoritmos tienen un límite de 5000 secuencias. Tanto en el caso de Serie_inter como en el de la justificación rápida no hemos incorporado, por simplicidad, una de las características comentadas anteriormente. Existen casos en que al secuenciar o justificar a la izquierda (derecha) la segunda (primera) parte de una actividad se puede

secuenciar junto a la primera (segunda) parte, produciéndose la fusión. En esa ocasión es posible que la actividad se pueda mover más a la izquierda (derecha), interrumpiéndola si es necesario. Esto puede llevar a que otras actividades o partes de actividades que habían sido consideradas y estaban por tanto justificadas a la izquierda (derecha) se puedan mover ahora más a la izquierda (derecha). En las pruebas computacionales no hemos considerado esta última posibilidad, sino que una vez secuenciamos o justificamos ambas partes de una actividad, éstas quedan fijas. Por tanto, no trabajamos necesariamente con secuencias rápidamente justificadas a la izquierda o derecha, algo que sí ocurriría si incorporáramos esa característica y que en teoría podría conducir a mejores resultados (aunque a costa de emplear más tiempo en cada secuenciación o justificación).

El experimento se ha realizado sobre el conjunto j120, añadiendo la opción de poder interrumpir cada actividad una vez. El CPM es también una cota inferior para el RCPSP_inter, por lo que emplearemos la desviación respecto del CPM para comparar los algoritmos construidos entre sí. También resultará útil para comprobar si resulta influyente introducir la interrupción en el RCPSP, i.e., si conseguimos resultados significativamente mejores que sin ella.

La Tabla 4.4 muestra los resultados de los cuatro algoritmos, en la segunda columna las desviaciones del CPM correspondientes al RCPSP (comentadas en el capítulo 3) y en la tercera los correspondientes al RCPSP_inter.

Algoritmo	RCPSP	RCPSP_inter
Aleat	47.51%	43.74%
Aleat + DJ	38.36%	34.26%
Hartmann (1')	37.00%	35.72%
Hartmann + DJ	33.24%	30.36%

Tabla 4.4. Resultados de Aleat y Hartmann con y sin justificación y con y sin interrupción.

A partir de estos resultados se pueden extraer varias conclusiones:

1) La (doble) justificación rápida mejora sensiblemente Aleat_inter y Hartmann_inter. De hecho, la mejora respecto del CPM (9.48 % y 5.36% respectivamente) es superior a la observada en el RCPSP (9.15% y 3.76% respectivamente). Esto demuestra nuestras afirmaciones sobre la utilidad de la justificación en el RCPSP_inter, así como también que la justificación se puede adaptar a otros problemas similares al RCPSP con esperanzas fundadas de éxito.

2) Existe una diferencia importante entre los algoritmos con y sin interrupción, obteniéndose mejores resultados cuando ésta se permite. Esta diferencia es considerablemente menor en el caso del algoritmo de Hartmann, que ofrece unos pobres resultados en el RCPSP_inter (peores que Aleat_inter + DJ, cuando en el RCPSP ocurre lo contrario). Esto puede ser debido a que, por simplicidad, no hemos incluido en el procedimiento ningún mecanismo para producir interrupciones, si exceptuamos Serie_inter. Sería sencillo introducirlos, por ejemplo en la población inicial o en la mutación, y con ellos seguramente mejoraría el algoritmo. Si así sucediera, Hartmann_inter + DJ también se vería beneficiado, aunque posiblemente en menor medida, puesto que la inclusión de la justificación rápida parece mitigar las carencias de Hartmann_inter, dado que la diferencia entre Hartmann_inter + DJ y Aleat_inter + DJ, 3.9%, no es mucho más pequeña que la existente en el RCPSP, 5.1%.

3) Ya se ha comentado en el punto anterior la diferencia existente entre un algoritmo en el RCPSP y el RCPSP_inter. Demeulemeester, en su tesis de 1992, estudió en dos casos diferentes el problema de la interrupción, cuando la disponibilidad de recursos es constante (el caso considerado aquí) y cuando ésta es variable. En ese trabajo la interrupción se abordaba de otra forma, cada actividad se podía dividir en subactividades de una unidad. Esto conlleva un número considerablemente mayor de soluciones posibles y la posibilidad de poder obtener mejores soluciones. Según sus propias palabras (pag. 155), 'Los resultados computacionales indican que la introducción de la interrupción en la secuenciación de proyectos provoca un efecto pequeño, salvo cuando las disponibilidades de los recursos son variables'. Este comentario proviene del hecho de que cuando las disponibilidades son constantes, en 80 de los problemas de Patterson (cf. apartado 3 capítulo 1) las soluciones óptimas del problema con interrupción tienen la misma longitud que las del RCPSP, y en los 30 restantes sólo existe una unidad de diferencia. Dado que el tiempo empleado en resolverlos era 33 veces mayor en el caso con interrupción, obviamente no merece la pena interrumpir las actividades de acuerdo con esos datos. Los resultados obtenidos por Hartmann_inter + DJ aportan un nuevo enfoque sobre este tema. Podemos afirmar que la interrupción sí es útil, al menos heurísticamente, dado que, con aproximadamente el mismo esfuerzo de programación y esfuerzo computacional, hemos obtenido soluciones considerablemente mejores que si no se permite interrumpir las actividades. Este hecho se demostró en Lino, 1997, en el caso sin recursos y con relaciones de precedencia generalizadas. Según nuestro conocimiento, es la primera vez que se demuestra en el RCPSP. Como dato significativo se puede añadir que la desviación respecto del CPM de las mejores soluciones almacenadas en PSPLIB

el 10 de septiembre del 2001 era de 30.41%, ligeramente superior a la de Hartmann_inter + DJ. Esto implica que un algoritmo sencillo con interrupción (y doble justificación) es capaz de superar en calidad media a la combinación de los mejores heurísticos existentes para el RCPSP – no creemos exagerado afirmar que se han empleado horas intentando resolver esas instancias, con un número importante de algoritmos de muy diversa índole – por lo que parece claro que incluir la interrupción facilita enormemente la obtención de buenas soluciones. Con respecto a la utilización de la interrupción en la resolución óptima, las diferencias entre los heurísticos con y sin interrupción parecen indicar que existen más diferencias entre los óptimos de las detectadas al emplear los problemas de Patterson (ver el apartado 6 del capítulo 1 para ver las deficiencias de este conjunto de instancias). Sin embargo, la posibilidad de una diferencia que puede ser pequeña puede no contrarrestar el enorme esfuerzo computacional adicional necesario para resolver de forma óptima el problema si se añade interrupción.

6.2. El RCPSP con múltiples modos y recursos no renovables

Este problema es, probablemente, el más general y, por esto, el más difícil de los problemas de secuenciación de proyectos (cf. Kolish, 1995, pag. 26). Incluso el problema de saber si existe alguna solución posible es NP completo. Para una formulación formal del RCPSP_MM, ver Talbot, 1982.

Este problema tiene dos diferencias con el RCPSP. La primera es que cada actividad j se puede secuenciar de varias maneras o modos m = 1, ..., M_j. En cada uno de ellos la actividad necesita de diferentes unidades de cada tipo de recurso y la duración también puede ser distinta, cumpliéndose $d_{j1} \leq d_{j2} \leq \ldots \leq d_{jM_j}$. La otra diferencia es la inclusión de otro tipo de restricción de recursos (pertenecientes al conjunto N), los no renovables (cf. apartado 1.3 capítulo 1). La disponibilidad de estos recursos no está limitada en cada unidad de tiempo, sino de forma total (K_r unidades, $r \in N$). Es esta limitación la que provoca la dificultad de encontrar una solución posible, dado que el problema de decisión consistente en decidir si existe una combinación de modos ($\sum_{m=1}^{M_j} x_{jm} = 1 \quad \forall j$) de manera que no se consuman más recursos de los disponibles ($\sum_{j=1}^{n} \sum_{m=1}^{M_j} k_{jmr} x_{jm} \leq K_r \quad \forall r \in N$, con k_{jmr} la cantidad de recurso r que emplea la actividad j si se ejecuta en el modo m) es NP completo si existe más de un recurso no renovable y más de un modo para cada actividad (cf. Kolisch, 1995, pag. 26) o más de dos recursos no renovables (cf. Kolisch y Drexl, 1996). Para este problema en particular hablaremos en general de secuencias/soluciones cuando se cumplan todas las

restricciones salvo quizás las de los recursos no renovables, y de secuencias/ soluciones posibles cuando tampoco éstas se violen.

6.2.1. Justificación directa

Sea S una secuencia (es decir, que cumpla las restricciones de recursos renovables y las de relaciones de precedencia). Si m(j,S) denota el modo en que se secuencia cada actividad j en S, los recursos no renovables consumidos por S son $K_r(S) := \sum_{j=1}^{n} k_{jm(j,S)r}$, $r \in N$. S es una secuencia posible sii $K_r(S) \le K_r \; \forall r \in N$

Supongamos que efectivamente S es posible. Entonces, existe una instancia del RCPSP para el que S es posible, aquella en la que:

- el grafo es el mismo que en el problema original
- se fija el modo de cada actividad a m(j,S), el que está empleando en S; por tanto la duración y los recursos renovables (y no renovables) que consume cada actividad son fijos
- las restricciones sobre los recursos renovables son las mismas que en P
- las restricciones sobre los recursos no renovables no son consideradas, dado que se cumplen y no se va a modificar los modos – y por tanto el consumo de recursos no renovables – de las actividades

Así pues, cualquier secuencia S posible para el RCPSP_MM es una secuencia posible para un cierto RCPSP y se puede justificar a izquierda o derecha según ese RCPSP, obteniéndose otra secuencia posible para P. Esto quiere decir que el RCPSP_MM puede beneficiarse, secuencia a secuencia, de las mejoras que provoca la justificación directa. Dada la potencialidad de la justificación en el RCPSP, esto puede ser muy beneficioso también para este problema.

Para adaptar los procedimientos del RCPSP a este problema es necesario disponer de una codificación adecuada. Una posibilidad que ya ha demostrado su eficacia es la empleada en Hartmann, 2001, donde cada solución lleva asociada una lista de actividades y un vector de n elementos con el modo en que se secuencia cada actividad. El algoritmo estudiado en ese artículo ha proporcionado uno de los mejores heurísticos para el RCPSP_MM y es una extensión natural para este problema del genético desarrollado por el mismo autor con el que hemos trabajado en el capítulo 3.

Teniendo en cuenta las mejoras obtenidas por Hartmann + DJ con respecto al GA original y que cada secuencia posible del RCPSP_MM se puede justificar como una del RCPSP, la modificación con la justificación del genético de Hartmann para el RCPSP_MM parece muy prometedora. Además, es posible que empleando la justificación comentada en el apartado siguiente se puedan conseguir aún mejores resultados.

6.2.2. Justificación adaptada

Al justificar una solución S como si de una secuencia de un RCPSP se tratara se obtiene una solución activa para ese RCPSP. Pero este concepto se puede ampliar al RCPSP_MM, como se hizo con el RCPSP_inter. Speranza y Vercellis, 1993, y Sprecher et al., 1997, introdujeron para algoritmos exactos los conceptos de movimiento a la izquierda multi-modo y secuencia ajustada.

Definición: Actividad justificada a la izquierda en el RCPSP_MM

> Dada una secuencia posible S y una actividad i, un **movimiento a la izquierda multi-modo** ('multi-mode left shift') de la actividad i es una operación sobre S que reduce la finalización de i sin cambiar los modos o finales de las demás actividades y sin violar las restricciones. Una secuencia es **ajustada** ('tight schedule') si no se le puede aplicar ningún movimiento multi-modo a la izquierda a ninguna actividad.

De acuerdo con nuestra nomenclatura, i **está justificada a la izquierda para el RCPSP_MM** cuando no se le puede aplicar ningún movimiento multi-modo y una secuencia ajustada es una secuencia justificada a la izquierda para el RCPSP_MM. De un modo análogo se definen los términos para la justificación a la derecha.

Esto quiere decir que para justificar una actividad a la izquierda (derecha) no sólo podemos trasladarla en su actual modo, sino que podemos cambiarle el modo para intentar secuenciarla más a la izquierda (derecha), siempre y cuando la secuencia se mantenga posible.

Un cambio de modo en una actividad i al justificarla a la izquierda que implique únicamente un cambio en los recursos empleados renovables puede afectar a la justificación de las actividades con final mayor que el antiguo inicio de i. Sin embargo, uno que disminuya los recursos no renovables empleados puede afectar a toda la secuencia ya que, en principio, cualquier actividad puede optar, empleando los recursos liberados, a un cambio de modo en el que antes no podía ejecutarse, y con este cambio poder secuenciarse antes. Esto quiere decir que no va a ser igual de

sencillo que en el RCPSP encontrar secuencias justificadas a la izquierda o derecha y, como ejemplo, la justificación por extremos puede proporcionar secuencias no justificadas. Teniendo esto en cuenta se puede decidir entre diferentes opciones, desde aplicar como máximo un movimiento multi-modo a la izquierda a cada actividad (el primero que se encuentre o el mejor) hasta considerar y aplicar movimientos a todas las actividades el número de veces que sea necesario hasta que la secuencia resultante sea justificada a la izquierda para el RCPSP_MM.

Dos de estas posibilidades (siempre justificando a la izquierda) han sido utilizadas en el algoritmo genético de Hartmann para el caso multi-modo (Hartmann, 2001). A cada solución producida por la combinación de secuencias le añade una búsqueda local, aplicándole movimientos multi-modo a la izquierda. La mejor variante de su GA consiste en aplicar un movimiento multi-modo a cada actividad, el primero que sea posible, comenzando con los modos de menor duración. Según sus resultados, esa variante es, independientemente del tamaño de población, siempre mejor que el GA simple. Esto parece implicar que la (doble) justificación empleando movimientos multi-modo puede funcionar mejor que la directa del RCPSP sin permitir cambios de modo.

Hartmann, 2001, emplea exclusivamente movimientos multi-modo a la izquierda, y solamente con soluciones posibles. Parece evidente que se van a poder mejorar esos resultados si además se aplican movimientos a la derecha y, especialmente, si se combinan las técnicas de justificar (como si fueran secuencias de un RCPSP) y los movimientos multi-modo a izquierda y derecha.

Hasta este momento, tanto en el artículo de Hartmann como en este apartado, sólo se ha justificado o aplicado movimientos multi-modo a soluciones posibles, pero en una parte importante de un heurístico puede ser que se trabaje con secuencias imposibles. Si únicamente se pudiera utilizar la justificación con soluciones posibles su aplicación quedaría bastante restringida; pero podemos ir más allá y justificar también soluciones imposibles.

Antes de ver diferentes modos de hacerlo, es importante resaltar que estamos intentando aplicar la justificación del mismo modo que en el caso del RCPSP. Es decir, no estamos considerando las soluciones de un modo aislado, sino dentro de un hipotético algoritmo heurístico. Por ello, suponemos que se han calculado un cierto número de soluciones y que, después de trabajar con esta solución, se calcularán más. Con la solución que obtengamos al justificar doblemente la solución imposible se trabajará del mismo modo que se hubiera hecho con la solución original, aplicándole movimientos, combinándola con otras, rechazándola si no interesa, etc. Por tanto,

puede ser interesante encontrar soluciones más cortas e igual o menos imposibles. Veamos diferentes formas de justificar una solución imposible.

Supongamos que S es una secuencia imposible, en el sentido que alguna de las restricciones no renovables no se cumplen. Sea N' el conjunto de los recursos donde se consumen más recursos de los disponibles, es decir, $K_r(S) > K_r$ $\forall r \in N'$ y $K_r(S) \leq K_r$ $\forall r \in N/N'$.

(1) Una posibilidad trivial es fijar los modos y justificarla como en el RCPSP. La secuencia S' final consumirá exactamente la misma cantidad de recursos no renovables que S pero será de duración igual o más pequeña.

(2) Otra posibilidad para intentar acortar más la duración es la de aplicar movimientos multi-modo que no incrementen la imposibilidad de S. La definición formal sería la siguiente. Justificar a la derecha una actividad i que se secuencie en el modo m(S) consiste en obtener la secuencia S' en la que sólo cambia el modo y la secuenciación de i. Ese modo m y el tiempo de finalización t se calculan de manera que: (i) $k_{imr} \leq k_{im(S)r}$ $\forall r \in N'$, (ii) $K_r(S) - k_{im(S)r} + k_{imr} \leq K_r$ $\forall r \in N \backslash N'$ (el consumo de recursos de N\N' sigue estando dentro de los límites admitidos), (iii) las restricciones sobre los recursos renovables se cumplen y (iv) $t \geq fin(i,S)$ y t es el máximo posible.

(3) Si i no consume recursos de N', la justificamos como en (2). En caso contrario, podemos intentar cambiarle el modo a la vez que la justificamos, de manera que rebajemos ese consumo, pudiendo llegar incluso a cumplir alguna de las restricciones que en S no se satisfacen. Este cambio nos interesa hacerlo siempre y cuando el resto de restricciones se sigan cumpliendo. La definición formal sería análoga a la de (2), pero el modo m se escogería de manera que se minimizara el consumo de recursos de N' (cumpliéndose el resto de restricciones) y t se maximizaría una vez escogido el m. El procedimiento de justificar una actividad descrito se puede aplicar a todas las actividades que empleen recursos de N' para intentar 'suavizar' la imposibilidad de S.

Las primeras dos formas de justificar son perfectamente válidas si no nos importa excesivamente en esta fase del heurístico si la solución resultante es posible o no, porque después vamos a aplicarle movimientos que la hagan posible o por otra causa. La tercera forma busca más acercarse a la región factible que mejorar la función objetivo. Vamos a analizar con detenimiento esta última forma de justificación.

La manera de minimizar el consumo de recursos de N' sería minimizar (asegurando el cumplimiento del resto de restricciones) una medida de recursos g prefijada, por ejemplo $g(m) = \sum_{r \in N'} \frac{\max\{K_r, K_r(S) - k_{im(S)r} + k_{imr}\}}{K_r}$. En esta medida cuando $K_r(S) - k_{im(S)r} + k_{imr} = K_r$ quiere decir que se va a pasar de incumplir esa restricción a cumplirla. Si $K_r(S) - k_{im(S)r} + k_{imr} < K_r$, ya no nos interesa especialmente minimizarla más, por eso está el máximo con K_r. Esta función pondera igual (salvo la división por K_r) llegar a cumplir una restricción que rebajar un poco la falta de cumplimiento en varias. Esto se podría modificar cambiando la medida.

Como puede ser relativamente costoso calcular siempre el 'mejor' movimiento multi-modo, otra posibilidad es la de aplicar movimientos multi-modo a las actividades que reduzcan el consumo de recursos no renovables de N', aunque no minimicen g.

Una técnica comentada en el apartado 5 puede ser útil en este caso y es la de fijar un k > 0 y justificar en [0,T(S) + k] (o, equivalentemente, fijar n en T(S) + k y justificar las demás actividades de acuerdo a la definición dada). De este modo, cuanto mayor sea el k mayores posibilidades habrá de obtener una solución posible al justificar las actividades, porque habrá más recursos renovables para utilizar en lugar de los no renovables. Como desventaja está la de que la duración de la solución final será menor o igual de T(S) + k, en lugar de menor o igual de T(S).

Es obvio que de esta manera no se va a descubrir soluciones posibles que no se pudieran obtener simplemente cambiando los modos de las actividades uno a uno según esa medida. La diferencia es que, en nuestro caso, se asegura que la duración va a ser como máximo la inicial + k, mientras que en el otro caso el incremento puede ser enorme y la solución desvirtuarse por completo. Esto puede ser muy importante si se utiliza dentro de un heurístico. La elección del k dependerá de la fase del heurístico en la que estemos y de si nos interesa más la duración o la posibilidad. Algunos de los factores para determinarlo serán lo que se vaya a hacer luego con la solución y si ya se dispone de alguna solución posible, en cuyo caso el k no será muy grande. Mientras no sea así se preferirá seguramente un k mayor.

Esta justificación generalizada hereda de la original el intento de aprovechar el potencial de la solución de la que parte. Esta técnica emplea la estructura de la solución y los recursos disponibles a lo largo de su longitud (+k) para intentar aproximarla al espacio de las soluciones posibles mientras que, a la vez, pretende acortar su longitud o no alargarla en demasía. Incluso, aunque se alarge algo, puede introducir en el algoritmo nuevas combinaciones de modos que cumplen o están cerca de cumplir las restricciones no renovables, y esto llevar a las primeras soluciones

posibles o a mejores soluciones al combinar esta secuencia con otras o aplicarle movimientos. Este procedimiento parece muy apropiado para explorar la frontera entre el espacio de las soluciones posibles e imposibles, una región donde pueden encontrarse soluciones de muy alta calidad ya que, en general, un alto consumo en recursos no renovables significa duraciones menores y consumo también menor en recursos renovables, por lo que las duraciones pueden ser menores.

Dados los condicionantes, es posible que en este problema una justificación no tan restrictiva como la de por finales, la justificación por elegibles o la justificación general, pueda proporcionar mejores resultados. Veamos las razones:

- Las actividades importantes en este problema son aquellas que emplean recursos de N', y se pueden encontrar a lo largo de toda la secuencia. Esto contrasta con el caso del RCPSP, donde las actividades que queríamos poder justificar a la derecha eran las que cumplían $s_i = ES_i$, y generalmente se encontraban al principio de la secuencia. Para que éstas pudieran ser secuenciadas, primero había que justificar la mayoría del resto de actividades, por lo que esta información no era demasiado útil. En este caso podemos encontrar actividades 'críticas' (que emplean recursos de N') a lo largo de la secuencia, por lo que sí puede ser una táctica interesante (al justificar a la derecha) justificar primero sus predecesoras y, en el momento en que una actividad crítica se pueda justificar rebajando el número de unidades de recurso de N', realizarlo.

- En el RCPSP, en la justificación a la derecha, cuando existía más de una actividad que comenzara en su ES, no se obtenía ninguna mejora si se lograba cambiar el inicio de todas ellas salvo el de una. En este problema cualquier justificación de una actividad 'crítica' con cambio de modo produce una 'mejora', i.e., un acercamiento de la solución al espacio de las soluciones posibles.

En cualquier caso parece obvio que existe un campo bastante amplio que estudiar dentro de la justificación (en el sentido más amplio) en este problema; cabe incluso la posibilidad de que se puedan crear algoritmos basados en ella muy potentes.

6.3. El RCPSP con fechas de entrega en la función objetivo

Vamos a considerar el problema de optimización en el que las restricciones son las mismas que en el RCPSP y lo que cambia es la función objetivo. En este caso, cada actividad lleva asociada una fecha de entrega ('due date') dd_i y el objetivo es minimizar la desviación con respecto a una solución ideal en la que todas las actividades

estuvieran procesadas antes de su fecha ($f_i \leq dd_i \; \forall i$). La función objetivo que vamos a considerar es la suma ponderada de la tardanza de cada actividad, i.e., $\sum_{i=1}^{n} \omega_i (f_i - dd_i)^+$, donde ω_i es un peso que marca la importancia de no sobrepasar la fecha de entrega de la actividad i. Lo que se va a explicar a continuación se podría adaptar fácilmente a otras funciones objetivo similares como, por ejemplo, la de minimizar el número de actividades **retrasadas** (que finalizan después de su fecha de entrega) o una en la que además se tuviera en cuenta la duración del proyecto.

6.3.1. Justificación directa

El conjunto de soluciones de este problema es el mismo que en el RCPSP, dado que las restricciones no han variado. Dada S una secuencia posible del RCPSP con fechas de entrega, se puede justificar a derecha y a izquierda como en el RCPSP aunque, a priori, no se puede asegurar $f(DJ(S)) \leq f(S)$, aunque sí $T(DJ(S)) \leq T(S)$. En la secuencia de la Figura 4.48, posible para el proyecto de la Figura 4.47, con una única fecha de entrega $dd_6 = 2$ y $\omega_6 = 1$, al justificar a la derecha por finales se obtiene la secuencia S' de la Figura 4.49. Esa secuencia, debidamente trasladada para que comience en 0, es justificada a derecha y a izquierda y es una unidad más corta que S, pero f(S') = 2 > 1 = f(S).

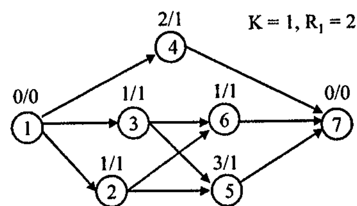


Figura 4.47

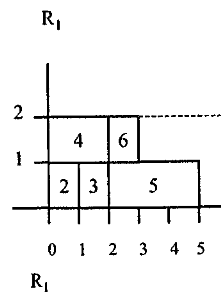


Figura 4.48

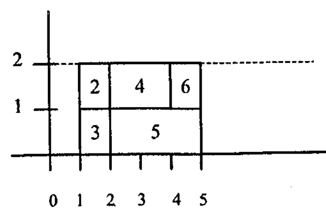


Figura 4.49

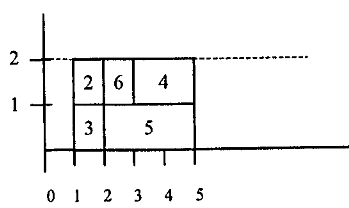


Figura 4.50

Los resultados del capítulo anterior indican que al aplicar la (doble) justificación se obtiene una mejora importante en la duración del proyecto, especialmente si la

secuencia original es de una calidad media o baja con respecto a la duración. Obviamente una reducción significativa lleva asociada que un conjunto importante de actividades, entre ellas todas las de mayor inicio (muchas de ellas son susceptibles de estar retrasadas), se secuencien antes.

Para confirmar esta aseveración hemos realizado el siguiente experimento. Hemos calculado 1666 secuencias aleatorias en cada instancia del conjunto j120, y las hemos justificado doblemente por extremos. Al hacerlo se calcula la mejora relativa con respecto a la suma de inicios de actividades, i.e., $\frac{\sum_{i=1}^{n}(s_i - s_i')}{\sum_{i=1}^{n} s_i}$, con S' = DJ(S). Al calcular la media de estas mejoras relativas se ha obtenido un 6.86%, con un 98.82% de secuencias mejoradas.

6.3.2. Justificación adaptada

El experimento anterior nos asegura que la (doble) justificación es capaz de reducir el inicio de un número importante de actividades. Lo único que debemos lograr para que la (doble) justificación mejore la nueva función objetivo es que entre esas actividades se encuentren el mayor número de actividades retrasadas en la solución original. Obviamente la DJ por finales original no es adecuada en este caso, puesto que puede cambiar los inicios de las actividades equivocadas y que sean las actividades ya retrasadas (o las que están al límite) las que pasen a secuenciarse más tarde. Por eso, es conveniente crear una justificación adaptada a los condicionantes del problema, que denominaremos **justificación restringida**.

En el caso de la justificación a la derecha, consiste en fijar la actividad n en el máximo entre su fecha de entrega y T(S) (si no tiene fecha de entrega, podemos crear una ficticia, $dd_n = \sum_{i=2}^{n-1} d_i$). Cada actividad se justificará como máximo hasta su fecha de entrega; las actividades retrasadas se consideran justificadas a la derecha. Sólo se consideran puesto que, por las restricciones del problema, no lo están. Si al terminar la justificación se reduce la longitud de la secuencia, se disminuyen los inicios de las actividades en consecuencia. Esto aumenta la diferencia entre el final de las actividades no retrasadas y su fecha de entrega y convierte las actividades retrasadas en menos retrasadas, llegando incluso a poder convertir alguna(s) en no retrasada(s). Esto puede llevar a que algunas actividades se puedan justificar más a la derecha

dentro de su fecha de entrega, por lo que se debe (intentar) justificar de nuevo a la derecha la secuencia, hasta que no se reduzca la longitud.

Sea i una actividad que no tenga asignada una fecha de entrega y, por tanto, no se considere en la función objetivo. Si i tiene un sucesor j con fecha de entrega, existe un instante de tiempo (una especie de fecha de entrega para i) de manera que si i finaliza más tarde j se retrasa inevitablemente. Al justificar a la derecha las actividades como i, es conveniente justificarlas como máximo hasta ese instante de tiempo.

En todas las secuencias (salvo las óptimas en casos muy aislados) existen unidades de tiempo donde no se emplean todos los recursos, más si se justifica en $[0,dd_n]$ en lugar de en [0,T(S)] (si $dd_n > T(S)$). Los resultados del capítulo anterior afirman que al justificar las actividades a la derecha en secuencias activas se rellenan estos 'huecos' y se puede conseguir rebajar la duración. Al obligar a que las actividades no se justifiquen más allá de su fecha de entrega limitamos el movimiento de recursos a la derecha pero permitimos que otras actividades, no tan restringidas, empleen esos huecos. Si incluso así rebajamos la duración, la función objetivo mejorará (las actividades retrasadas lo estarán menos o dejarán de estarlo, porque las habíamos fijado). Si no conseguimos una rebaja en la duración, al menos habremos creado huecos en la parte inicial de la secuencia que se intentarán aprovechar al justificar a la izquierda para adelantar las actividades retrasadas.

Volviendo a la secuencia S de la Figura 4.48, si la justificamos de forma restringida a la derecha (fijando n en T(S)) obtenemos la secuencia S'' de la Figura 4.50 que, debidamente trasladada para que comience en 0, tiene un valor en la función objetivo de 0, menor que el de S.

A la hora de justificar a la izquierda tras la justificación a la derecha se debe intentar mover las actividades retrasadas, dado que cualquiera de estos movimientos repercutirá en la función objetivo. Para ello, se puede proceder, por ejemplo, a justificar primero las actividades retrasadas y sus predecesoras, teniendo en cuenta los pesos de las actividades retrasadas para determinar el orden concreto entre ellas, y terminar con el resto de actividades hasta obtener una secuencia justificada a la izquierda. Es obvio que se pueden crear diferentes variantes a partir de la justificación general o por elegibles. Y, también, que es más sencillo especificar reglas para la elección de la actividad a justificar en este problema que en el RCPSP ya que, en este último, lo que se busca es una mejora global de la duración mientras que en el de fechas de entrega bastan mejoras locales. De igual forma, al justificar a la derecha, se podría considerar primero las actividades con mayor fecha de entrega o sin ella y las que no fueran predecesoras de actividades retrasadas para que, al justificar a la

izquierda posteriormente, hubiera recursos disponibles y unidades de tiempo suficientes como para justificar a la izquierda las actividades retrasadas.

Esta nueva justificación asegura $f(DJ(S)) \leq f(S^R) \leq f(S)$, pero puede resultar ineficaz cuando se aplica sobre soluciones donde la gran mayoría de actividades están retrasadas. En estos casos puede ser más provechoso no restringir la justificación de las actividades mediante las fechas de entrega, sino que éstas sirvan como guía para la selección de actividades en la justificación, teniendo en cuenta los pesos de las actividades y las actividades retrasadas. Al justificar todas las actividades, el cambio en la pauta de utilización de recursos será mucho mayor y esto podrá ser utilizado en la justificación a la izquierda. En un número importante de instancias este tipo de soluciones se corresponde en general con una longitud de secuencia larga. Como se ha comprobado la rebaja en la duración en estos casos al aplicar DJ es muy grande, lo que es equivalente a un gran número de recursos desaprovechados en la solución original. Esto implica que vamos a ser capaces de mover muchas actividades y únicamente habrá que preocuparse de mover las adecuadas.

Resumiendo, parece que sería acertado aplicar la justificación sin restricciones, pero guiada, cuando las soluciones contengan un número importante de actividades retrasadas y la restringida, en otro caso. En nuestra opinión esta justificación mejorará sustancialmente los algoritmos heurísticos para el problema. Al igual que en los anteriores apartados, una opción sencilla sería emplear el genético de Hartmann más las nuevas justificaciones, teniendo en cuenta la nueva función objetivo para introducir o eliminar las soluciones de la población.

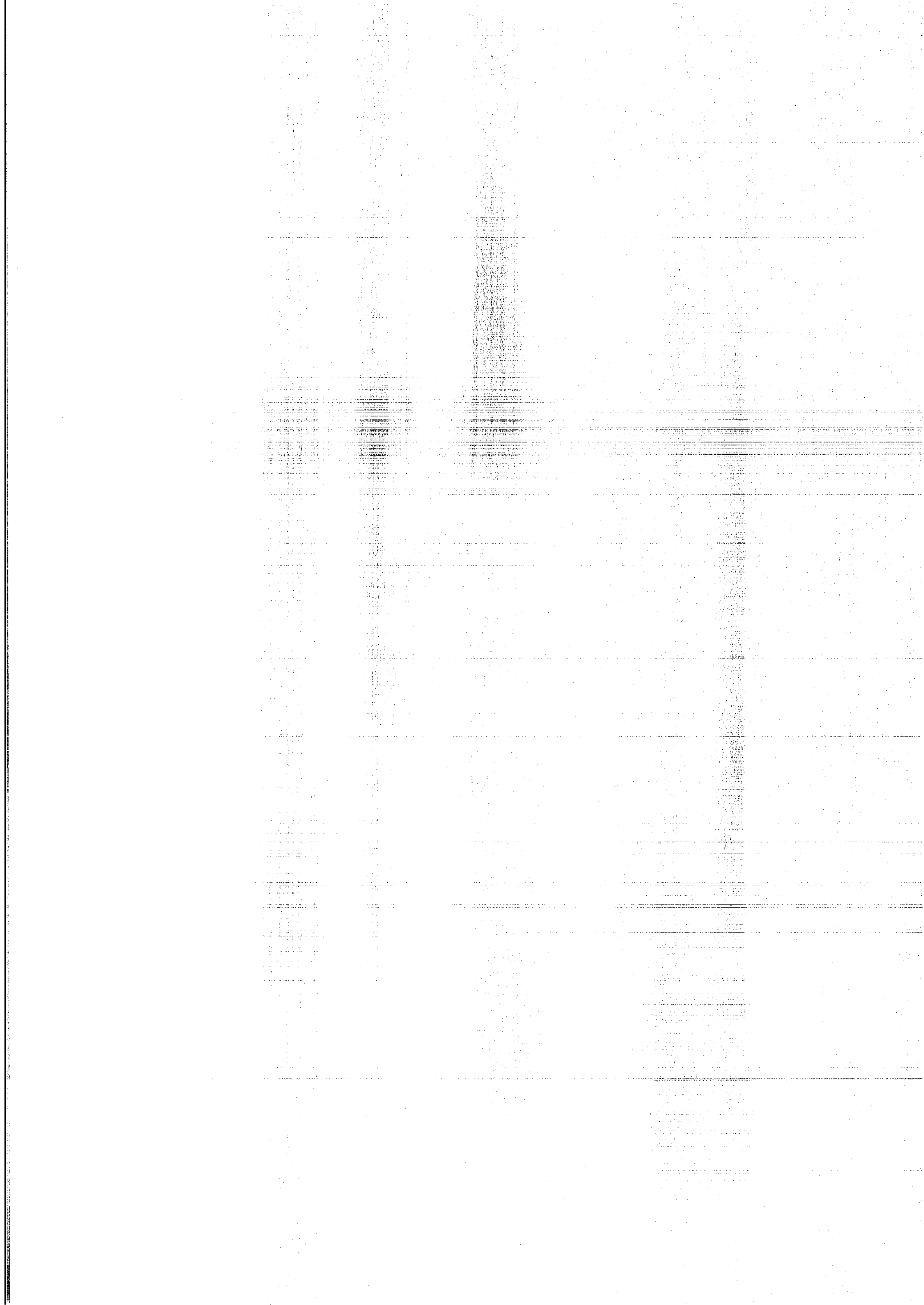
Antes se ha comentado que una mejora en la longitud de la secuencia inicial no conlleva necesariamente una mejora en la función objetivo. De hecho, la justificación puede ser incapaz de mejorar la función objetivo. Como contrapartida, en las soluciones en las que no es posible mejorar la longitud con la (doble) justificación sí puede ser posible mejorar la función objetivo, ya que en ella intervienen todos o varios de los finales de las actividades, en lugar de exclusivamente el máximo de los finales, como ocurre en el RCPSP. Una secuencia obtenida a partir de la doble justificación con la misma duración que la original puede haber cambiado convenientemente los inicios de las actividades de forma que se haya producido una mejor solución.

Teniendo en cuenta este comentario se pueden crear movimientos basados en la justificación que se pueden aplicar a una secuencia para mejorarla, en los que no sea necesario justificar la secuencia completa. Supongamos que encontramos un conjunto A y otro B de actividades de manera que al justificar (o trasladar) las actividades de A a la derecha las actividades de B se puedan justificar a la izquierda. El cambio en la

función objetivo al aplicar este movimiento es de $\sum_{i \in A \cup B} \omega_i ((f_i' - dd_i)^+ - (f_i - dd_i)^+)$. Lo único que se debe hacer es buscar los conjuntos A y B adecuados de manera que el cambio sea negativo. Un ejemplo simple de esto es que A contenga 2 ó 3 actividades no retrasadas que se justifiquen a la derecha hasta su fecha de entrega y esto permita que una actividad retrasada (B estaría formado por esa actividad) pase a no serlo al justificarse a la izquierda. Pero este movimiento es mucho más general, ya que el conjunto A puede contener actividades retrasadas y las actividades de B no tienen por qué pasar a ser no retrasadas, pero los pesos de unas y otras y las unidades de tiempo que se mueven determinan si el movimiento mejora la función objetivo.

Capítulo 5

HGA, un algoritmo genético híbrido



1. INTRODUCCIÓN

Se ha comprobado en el capítulo 3 el buen funcionamiento del genético de Hartmann más la (doble) justificación. Ésta ha potenciado enormemente un algoritmo que ya de por sí obtenía soluciones de relativa calidad; de hecho, hasta hace un par de años se podía considerar como uno de los dos mejores heurísticos para el RCPSP (cf. Kolisch y Hartmann, 1999), y el mejor en j120.

El operador con el que trabaja ese procedimiento es el de dos puntos de corte adaptado para permutaciones, un operador muy general y empleado en problemas combinatorios de muy diversa índole. Las preguntas naturales que surgen a partir de este hecho son por qué un operador tan general funciona en el RCPSP y si no es posible idear una manera de combinar soluciones más específica del problema que pueda superarlo. En este capítulo vamos a desarrollar un algoritmo genético híbrido, HGA, de dos fases, en cada una de las cuales se empleará un proceso evolutivo similar al de Hartmann + DJ, pero fundamentado en un nuevo operador binario. En los resultados computacionales se demostrará que HGA supera al resto de algoritmos de la tesis, en particular a Hartmann + DJ.

Una de las razones primordiales para el buen funcionamiento del operador de cruce de dos puntos de corte es la aplicación del SGS Serie a la lista de actividades obtenida. El conocimiento del problema no proviene tanto del operador como de este esquema. Pero esto, por sí solo, no es la única clave de la calidad final de las soluciones, el operador binario también influye. Así, ni el operador suma ni el operador porcentaje, dos de los mejores empleados en la tesis, consiguen igualar en calidad al operador de cruce de dos puntos de corte al emplearlos en Hartmann + DJ en lugar de éste, por lo que debe existir en este último una componente que lo destaque por encima de otros como los dos citados. Este factor es fácil encontrarlo si se razona qué características heredan las listas de actividades – y por ende las secuencias – tras la aplicación de cada operador o, más concretamente, los rasgos traspasados de padres a hijos en el caso del operador de cruce de dos puntos de corte frente a los demás. En este último se conservan los agrupamientos de actividades, las actividades que se encuentran juntas en las listas de actividades padres se posicionan en general juntas en los hijos. Esto se traslada a que las actividades que se secuencian juntas en los padres tienden a secuenciarse juntas en los hijos, a menos que consigan adelantarse debido al mecanismo que emplea Serie.

Esta propiedad no la comparten ni el operador suma ni el operador porcentaje, ni el resto de operadores binarios del apartado 2.2.1 del capítulo 3. El primero suma las posiciones absolutas en las listas y el segundo 'intercala' actividades de A y B, si una actividad ocupa una posición predominante en A o en B también la ocupará con mucha probabilidad en la combinación de A y B. Pero la experiencia – la comparación en calidad HIA v. CARA y la mejora proveniente de la aplicación de la justificación – nos indica que no parece que en general, y menos en proyectos grandes, sea determinante el intervalo de secuenciación de una única actividad, su posición exacta dentro de la secuencia. Tal vez esto sea más relevante cuanto más cerca se esté de la solución óptima, pero la práctica parece apuntar que lo verdaderamente importante es la combinación de actividades que se están secuenciando juntas, cómo están aprovechando los recursos.

Dado que este rasgo parece ser más determinante que las posiciones absolutas o relativas para la calidad de una solución, es natural que las soluciones obtenidas por el operador de cruce de dos puntos de corte sean de mayor calidad que las calculadas a partir de la suma o el porcentaje, dado que estos operadores destruyen en general las estructuras de actividades de los padres (en el sentido de las actividades que se secuencian en paralelo).

2. HGA, EL ALGORITMO DE LOS PICOS

Este razonamiento puede ser útil en la búsqueda de un operador binario más eficiente que el operador de cruce de dos puntos de corte. El nuevo operador debería conservar las estructuras acertadas de los padres, permitiendo que, en general, las actividades que se secuencian juntas en ellas permanezcan así en las soluciones calculadas. Pero, a diferencia de los operadores de cruce de uno o dos puntos de corte, que dejan al azar las partes de los padres que heredan los hijos sin tener ninguna garantía, o incluso indicio, de que realmente sean de calidad, parece más adecuado explotar el conocimiento del problema para identificar y combinar partes de buenas soluciones que realmente han contribuido a su calidad. Esta es la finalidad del **operador de cruce de los picos**, un operador específico para el RCPSP que introduciremos más adelante. Para ello vamos a introducir el concepto de pico de una secuencia que se define a partir de un intervalo temporal donde los recursos consumidos se acercan a los disponibles.

2.1. El operador de cruce de los picos

Consideremos el proyecto de la Figura 2.18 y la secuencia S de la Figura 2.19 (pág. 66) cuya duración es 17 unidades. Para cada uno de los 17 períodos de la duración podemos sumar las unidades del único recurso existente utilizadas por S y reflejar dicha información en el gráfico de utilización de recursos de la Figura 5.1 donde en el eje horizontal aparecen los períodos de tiempo (período i = [i-1,i]) y en el eje vertical, las unidades del recurso utilizadas. Podemos observar que en el gráfico existen picos y valles correspondientes a altas y bajas utilizaciones del recurso, respectivamente. Hay intervalos de tiempo (p.e. [0,3]) en los que la utilización del recurso es alta (100 % de las 6 unidades disponibles del recurso) mientras que hay otros (p.e. [13,17]) en los que la utilización es mucho más baja (50 % de las unidades disponibles del recurso). La utilización del recurso en el intervalo [6,8] también es alta (100 %) mientras que la clasificación de los intervalos [8,9] o [9,12] en intervalos de utilización alta o baja dependerá de la definición de utilización alta que adoptemos. Si, por ejemplo, definimos que un intervalo de tiempo es de utilización alta del recurso si en todos los períodos de tiempo comprendidos en él utiliza más del 80 % de las unidades disponibles del recurso y baja en caso contrario, entonces el intervalo [6,9] es de utilización alta mientras que el intervalo [9,12] es de utilización baja. Si el límite lo ponemos en el 65%, todo el intervalo [6,12] es de utilización alta. En cualquier caso, la alta utilización de recursos es una característica deseable en una secuencia que sería deseable transmitir a los descendientes en un proceso de cruce.

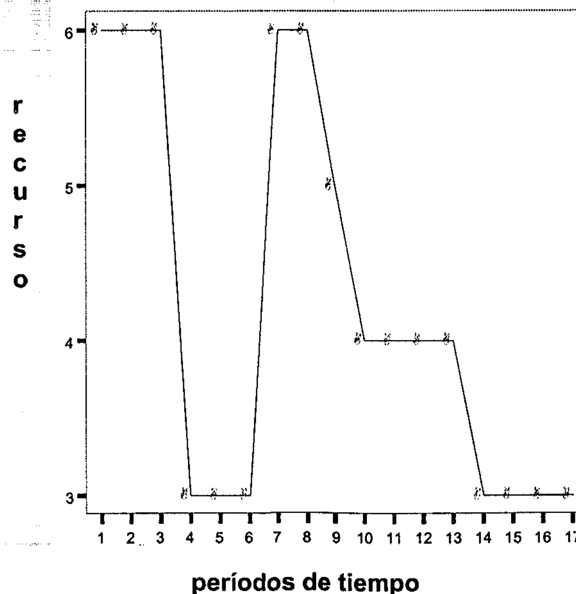


Figura 5.1. Gráfico de utilización de recurso.

Volviendo a la solución S de la Figura 2.19 (pág. 66), vemos que la alta utilización de recursos en el intervalo [0,3] se ha producido porque se han secuenciado en serie las actividades 3, 2 y 4 seguidamente y en este orden. (3,2,4) es una sublista de la representación AL de S que llamaremos pico. Si establecemos una utilización mínima del 80 % para ser considerado pico, entonces [6,9] es el único otro pico de S. Ambos picos están ordenados de forma natural por su ubicación en la escala de tiempos. Dadas dos secuencias, el padre y la madre, el operador de cruce de los picos genera dos nuevas soluciones, el hijo y la hija, de manera que el hijo (hija) hereda los picos del padre (de la madre) y la madre (el padre) determina la posición del resto de actividades que se van a poner delante y detrás de cada pico. Una vez vistas las ideas motivadoras del operador de cruce de los picos, y antes de definirlo con precisión, necesitamos algunas definiciones.

Sea S una secuencia activa y sea t un instante de tiempo. Sea **Scheduled(t)** el conjunto de actividades que, según S, se secuencian en el período [t,t+1], es decir, Scheduled(t) = $\{i \in \mathbf{V};\ [t,t+1] \cap]s_i,f_i[\neq \varnothing\}$. Una medida agrupada de la proporción de recursos que utiliza la solución S en el período [t,t+1] es la razón de utilización de recursos, vista en el capítulo 2:

$$RUR(t) = RUR(Scheduled(t)) = \left(1/K\right) * \sum_{j \in Scheduled(t)} \sum_{k=1}^{K} \frac{r_{j,k}}{R_k}.$$

Notar que $0 \leq RUR(t) \leq 1$.

Dado un umbral **densidad** diremos que un período [t,t+1] es de utilización alta para S si RUR(t) ≥ densidad. Diremos que un intervalo de tiempo $I = [I_s, I_f]$, con $I_s < I_f$, es de utilización alta para S si todo período incluido en I es de utilización alta para S. Sea λ una representación AL y sea I un intervalo de tiempo de utilización alta para $S(\lambda)$. Entonces diremos que la sublista de λ formada por las actividades que se secuencian en algún periodo de I es un **pico** de λ. Fijémonos en que un pico P de λ es una lista ordenada de actividades, por lo que P(j) es la actividad que ocupa la posición j-ésima en P. Fijémonos, también, en que las posiciones en λ de las actividades de un pico no tienen que ser consecutivas. Sea i la primera actividad de un pico P de λ. Entonces definimos la posición de P en λ como la posición de i en λ, es decir p(P) = orden(i,λ).

La **lista ordenada de picos disjuntos**, $P_1, P_2, \ldots, P_q$, de una representación AL λ dada se construye de la siguiente manera. P_1 es el primer pico de λ maximal respecto de la inclusión. P_j es el primer pico maximal por la derecha a partir de $t_j = \max\{ s_i + d_i ;\ i \in P_{j-1}\}$. La lista puede ser vacía.

Si los picos calculados no fueran disjuntos tendríamos que eliminar en general algunas actividades de algunos de ellos (degradarlos) antes de poder introducirlos en el hijo. En ese caso, no tendríamos controlada la razón de utilización de recursos de los picos degradados y podríamos insertar partes de los padres con una utilización por debajo de lo deseado. Al calcular los picos disjuntos evitamos ese problema y el cálculo es sencillo y rápido.

Sean λ (el padre) y λ' (la madre) dos representaciones AL de la población actual que queremos cruzar. El operador de cruce de los picos genera dos nuevas listas de actividades: el hijo λ^s y la hija λ^d. Vamos a ver cómo se genera el hijo. La hija se genera de manera análoga con sólo intercambiar los papeles del padre y de la madre.

Se escoge aleatoriamente el valor del umbral densidad en el intervalo [*lthreshold*, *uthreshold*], donde *lthreshold* y *uthreshold* son parámetros del sistema y 0 ≤ *lthreshold* < *uthreshold* ≤ 1. Sea P_1, P_2, ..., P_q la lista ordenada de picos disjuntos de λ según el umbral densidad. Sea *Pred(P_i)* la lista de las actividades que son predecesoras de alguna actividad del pico P_i, ordenadas según λ'. *Pred(P_i;j)* denota la actividad que ocupa la posición j-ésima en la lista $Pred(P_i)$.

El siguiente procedimiento construye el hijo λ^s asignando las n actividades a las n posiciones de una en una y en orden creciente de posición. En cada momento, una actividad es candidata si no pertenece a ningún pico y no ha sido asignada todavía pero todas sus predecesoras sí lo han sido.

Figura 5.2.

1. h = 1.
2. Desde i = 1 hasta q, hacer
    2.1. Desde j = 1 hasta n, mientras $j < p(P_i)$, hacer
        2.1.1. Si $\lambda'(j)$ es candidata, hacer $\lambda^s(h) = \lambda'(j)$ y h = h+1.
    2.2. Desde j = 1 hasta $|Pred(P_i)|$, hacer
        2.2.1 Si $Pred(P_i;j)$ es candidata, hacer $\lambda^s(h) = Pred(P_i;j)$ y h = h+1.
    2.3. Desde j = 1 hasta $|P_i|$, hacer
        2.3.1. $\lambda^s(h) = P_i(j)$ y h = h+1.
3. Desde j = 1 hasta n, hacer
    3.1. Si $\lambda'(j)$ es candidata, hacer $\lambda^s(h) = \lambda'(j)$ y h = h+1.

Este esquema se interpreta de la siguiente manera:

Figura 5.3.

1. h = 1.
2. Para cada pico i, i = 1 hasta q, hacer:
   2.1. Poner las actividades correspondientes de la madre delante del pico i.
   2.2. Completar las predecesoras del pico i.
   2.3. Poner el pico i.
3. Poner el resto de actividades detrás del último pico, en el orden que marca la madre.

En el caso de que la lista ordenada de picos disjuntos sea vacía, entonces $\lambda^s = \lambda'$.

2.2. Una población en el vecindario de una solución

Si consideramos la doble justificación como una función de mejora F, Hartmann + DJ se puede interpretar desde el punto de vista de MetaRCPSP (cf. apartado 7 capítulo 2). Al principio (Fase 0 de MetaRCPSP) se construye la población inicial mediante muestreo y la regla LF, después (Fase 1 de MetaRCPSP) se aplica F a cada individuo de la población inicial y, por último (Fase 2 de MetaRCPSP), se combinan las soluciones y se intenta mejorarlas mediante la función F. El modo de combinar las secuencias es diferente de la de HIAC y CARA, empleando un genético y un operador binario 'simple' (no se realiza reencadenamiento de trayectorias) para ello. Esto es posible debido a que esta F es muy rápida y se puede aplicar muchas veces.

El algoritmo genético híbrido HGA va a seguir estas pautas, salvo algunos cambios como el del operador. Pero, además, vamos a añadir la Fase 3 de MetaRCPSP, no empleada en Hartmann + DJ. La forma concreta será la de dividir el número total de secuencias que queramos generar en dos mitades. La primera mitad la generaremos con el proceso evolutivo habitual, aplicado sobre la población fruto del muestreo sobre LF. Una vez se haya llevado a cabo el número de iteraciones predeterminado, la solución obtenida será presumiblemente de una cierta calidad. En algunos genéticos (Hartmann, 2001) se ha experimentado con el método 'multi-start', se construyen diferentes poblaciones con el mismo método y se aplica el GA a cada una de ellas por separado, pero este enfoque no suele funcionar debido a que el número de secuencias que se calculan se divide por el número de poblaciones y cada GA comienza de nuevo con soluciones de baja calidad. En lugar de eso, y dado que la experiencia con los anteriores heurísticos parece indicar que se pueden obtener mejoras si se explora en las inmediaciones de las mejores soluciones, vamos a aplicar el mismo proceso evolutivo pero sobre una población distinta. La segunda población se construirá aplicando β-Biased sobre la mejor solución obtenida hasta ese momento,

tantas veces como marque el tamaño de la población. Esta población será de una calidad inferior a la de esa mejor solución, pero muy superior a la de una población formada por el muestreo sobre LF.

2.3. Esquema algorítmico de HGA

Población inicial

La función que construye la población inicial de la primera mitad del algoritmo, **Construir_Pob1**, emplea el muestreo aleatorio sesgado basado en la peor elección con la regla LFT y $\alpha = 1$ para construir POPsize individuos. Después de que aproximadamente nsche/2 secuencias han sido generadas, la función **Construir_Pob2** construye la población inicial de la segunda parte, aplicando β-Biased sobre la mejor solución de la primera parte POPsize/2 (POPsize/2 – 1 si POPsize/2 es impar) veces con $\beta = 1 - 20/n$.

Cada individuo de ambas poblaciones se justifica doblemente y se vuelve a introducir en la población.

Selección de los padres

Si *size* es el tamaño de la población actual (su valor depende de la fase pero es siempre par) y π, $0 < \pi \leq 1$, es un parámetro del sistema, entonces se genera $\lfloor \pi \, \text{size}/2 \rfloor$ parejas de individuos (padres) de la siguiente manera. Las parejas se escogen una a una. Cada vez que se escoge una pareja se elimina temporalmente (hasta que finalice el proceso) de la población. El primer elemento de la pareja es el individuo de la población actual con longitud más pequeña; el segundo elemento se escoge aleatoriamente de entre el resto de individuos de la población actual. La función que selecciona las parejas la denominaremos **EscogerParejas**. Fijémonos en que π es el porcentaje de soluciones de la población que se van a combinar.

Esta forma de seleccionar los padres generaliza en cierta forma la utilizada por Hartmann, dado que un valor de $\pi = 1$ equivale a escoger aleatoriamente POPsize/2 parejas, el método empleado en ese algoritmo genético. Si π es lo suficientemente pequeño, únicamente combinamos el mejor individuo de la población con otra solución, escogida aleatoriamente. Para un π determinado, se tiene la seguridad de que los k mejores individuos de la población se van a combinar (con otra solución o entre sí), donde ese k crece a medida que π aumenta y se acerca a 1($k = \lfloor \pi \, \text{size}/2 \rfloor$). Así pues, el parámetro π mide en cierta forma lo elitista que es la manera de

seleccionar los padres a combinar. Que el segundo elemento del par que se va a combinar se escoja aleatoriamente es para garantizar la diversidad y para que todos los individuos de la población tengan una probabilidad positiva de combinarse en cada iteración.

Mutación y Selección

La mutación y la selección se aplica como en Hartmann (ver apartado 8.2 del capítulo 1).

En la Figura 5.4 se describe el esquema algorítmico de HGA.

Figura 5.4. HGA(nsche,POPsize,π)

1. POP = Construir_Pob1(POPsize).
2. Calcular iter a partir de nsche, POPsize y π.
3. Desde i = 1 hasta iter, hacer
   3.1. EscogerParejas(POP,π).
   3.2. Obtener hijos aplicando:
      3.2.1. El operador de cruce de los picos a cada pareja de padres.
      3.2.2. La mutación y la doble justificación a cada hijo.
   3.3. Añadir los hijos a la población.
   3.4. Eliminar los POPsize peores individuos de la población.
4. POPsize = POPsize/2. Si POPsize es impar, hacer POPsize = POPsize – 1.
5. POP = Construir_Pob2(POPsize).
6. Aplicar las etapas 2 y 3 a la nueva población.
7. Devolver la mejor solución encontrada.

La fórmula de calcular iter es iter = $\frac{nsche/2}{6\lfloor \pi \text{ POPsize/2} \rfloor}$ -1, dado que en cada iteración se calculan $6\lfloor \pi \, POPsize/2 \rfloor$ secuencias.

3. Resultados computacionales

Vamos a emplear los mismos conjuntos de instancias y las mismas medidas para calibrar la bondad del algoritmo que en el resto de la tesis. Teniendo en cuenta que el conjunto j120 es el más difícil, donde mejor se observaban las diferencias entre los algoritmos y que ha sido empleado por los autores de los mejores algoritmos, nos centraremos especialmente en ese conjunto para comparar HGA con el resto de heurísticos.

Elección de Parámetros

Hemos fijado los valores de lthreshold = 0.75, uthreshold = 0.9 y pmutation = 0.05 (este último heredado de Hartmann).

El conjunto j120

Las pruebas computacionales se han realizado para valores de nsche = 2500, 5000, 10000, 25000 y 100000. Para cada valor de nsche se fijaron los parámatros POPsize y π con unas pruebas preliminares. Los valores de POPsize son 24, 50, 100, 200 y 400 para nsche = 2500, 5000, 10000, 25000 y 100000, respectivamente. El valor de π es 0.6 para 2500, 0.4 para 5000, 10000 y 25000 y 0.9 para 100000.

Los conjuntos j90, j60 y j30

Los valores de POPsize son 50 para j90 y j60 y 100 para j30, con unos valores de π de 0.9, 0.7 y 0.8 respectivamente.

Resultados computacionales

La siguiente tabla muestra la calidad de HGA5000 con las medidas habituales.

	ΣPSPLIB	ΣHGA	desv_CPM HGA	desv_UB HGA	max_desv HGA	mejor_sol HGA	med_CPU HGA	max_CPU HGA
j120	74013	75222	32.54	1.36	6.35	206	2.03	3.46
j90	45741	45974	10.46	0.40	5.48	374	0.61	2.31
j60	38368	38551	11.10	0.38	5.30	375	0.46	1.42
j30	28316	28339	13.47	0.06	3.45	462	0.31	0.66

Tabla 5.1. Resultados computacionales para j30, j60, j90 y j120.

Las desviaciones obtenidas son pequeñas, especialmente si tenemos en cuenta el tiempo medio y máximo. El aumento de este tiempo medio al aumentar n es menor que en otros algoritmos. Pasar de j60 a j120 implica multiplicar por 4.4, mientras que otros heurísticos tienen un factor de: 6.2 (CARA), 12.7 (HIAC), 5.26 (Hartmann), 5.5 (Hartmann (2)), 17.1 (Möhring et. al), 24.35 (Nonobe e Ibaraki). Existen dos algoritmos con un factor considerablemente mejor que HGA, el de Tormos y Lova, 2.6, y el de Alcaraz y Maroto, 2.

Las medidas más importantes de HGA sobre j120 para valores de nsche de 2500, 5000, 10000, 25000 y 100000 se presentan en la Tabla 5.2.

	Σ	desv_UB	desv_CPM	mejor_sol	med_CPU	max_CPU
HGA2500	75584	1.77	33.18	195	1.02	1.76
HGA5000	75222	1.36	32.54	206	2.03	3.46
HGA10000	74933	1.05	32.04	226	4.01	6.97
HGA25000	74640	0.71	31.51	287	10.02	17.69
HGA100000	74319	0.37	30.95	392	39.51	69.26

Tabla 5.2. Resultados computacionales de HGA con diferentes límite sobre el nº de secuencias.

Podemos observar que la calidad aumenta con el número de secuencias, aunque también aumenta el tiempo medio, de forma lineal. Este tiempo es muy pequeño en 2500 secuencias y no es excesivo en 100000. Cabe destacar que el máximo tiempo empleado en una instancia en j120 es menos del doble de la media en todos los casos. El número de mejores soluciones que se consigue crece con el número de secuencias.

En el capítulo 3 se ha comentado que, en ocasiones, al añadir la (doble) justificación a un algoritmo se podía rebajar el tiempo empleado. En este caso HGA tarda aproximadamente 0.6 segundos de media menos que HGA sin la doble justificación, a pesar de que en el segundo se intenta abortar la secuenciación de cada solución si se puede asegurar que su duración será excesiva, mientras que en HGA esto sólo se puede hacer con una de cada tres secuencias (cf. apartado 3 capítulo 3).

Hemos querido comprobar si la introducción de la segunda fase y la posibilidad de modificar π mejoran el algoritmo. Para ello hemos comparado HGA5000 con el mismo algoritmo pero con sólo la primera fase (con el doble de iteraciones) y π = 1. Los resultados obtenidos (Tabla 5.3) muestran que las dos fases y modificar π mejoran la calidad del algoritmo.

	Σ	desv_UB	desv_CPM
HGA5000	75222	1.36	32.54
HGA5000 con 1 fase y $\pi = 1$	75457	1.61	32.96

Tabla 5.3. HGA y HGA con 1 fase y $\pi = 1$.

Comparaciones con otros heurísticos de la literatura

La Tabla 5.4. refleja los datos ya comentados en el capítulo 2 para j120.

	desv_CPM	med_CPU	ordenador	nº secuencias	estimación desv_UB
Alcaraz y Maroto	36.57	9.37	PC 166	5000	3.86
Dorndorf et al.	37.1	205	PC 200	5000	4.41
Hartmann (1)	36.74	13.15	PC 133	5000	3.88
Hartmann (2)	35.35	14.05	PC 133	5000	2.99
Hartmann (3)	35.55	≥ 17 (=CARA)	PC 400	-	*3.32*
Merkle et al. (1)	36.65	25	PC 500	5000	3.86
Merkle et al. (2)	33.68	25 minutos	PC 500	-	1.98
Merkle et al. (3)	32.97	>> 25 min.	PC 500	-	1.62
Merkle et al. (4)	35.43	25	PC 500	5000	2.99
Möhring	36.2	65	Sun Ultra 2 200	-	3.81
Möhring (2)	35.3	≈ 65	Sun Ultra 2 200	-	2.99
Nonobe e Ibaraki	34.99	645	Sun Ultra 2 300	-	*2.95*
Tormos y Lova	35.62	29.85	PC 200	5000 ó 7500	3.11

Tabla 5.4. Resultados de otros algoritmos en j120.

HGA5000 obtiene un nivel de calidad superior a todos los algoritmos de la Tabla 5.4, mientras que su tiempo medio parece inferior al de la mayoría. Los únicos algoritmos donde, a primera vista, no está clara esta relación temporal son Hartmann (1) y (2) y Alcaraz y Maroto, pero si tenemos en cuenta que HGA2500 tiene una desviación respecto del CPM (UB) de 33.18 (1.77)% con 1.02 segundos de media, parece claro que el procedimiento HGA es también superior a esos tres heurísticos.

Cabe destacar la diferencia en calidad entre HGA5000 y los algoritmos de la tabla con un límite de 5000 secuencias; existe una diferencia en desv_CPM (desv_UB) de 2.8 (≈1.63)%. También es importante reseñar la diferencia entre la máxima calidad media conocida en la literatura, 32.97 (≈1.62)%, y la obtenida por HGA100000, 30.95 (0.37)%. Para la primera media se necesitaron bastante más de 25 minutos de media mientras que HG100000 emplea 40 segundos.

En los conjuntos j90 y j60, HGA5000 obtiene una mejor calidad media que el resto de algoritmos (ver Tabla 2.9, pág. 90) con un tiempo sensiblemente inferior (sin tener en cuenta los ordenadores). En j30, HGA ofrece una desviación con respecto a los óptimos similar a Nonobe e Ibaraki y por debajo del resto, pero el de Nonobe e Ibaraki con un tiempo varias veces superior (incluso teniendo en cuenta los ordenadores).

Comparación con CARA y HIAC

En la Tabla 5.5 se encuentran, por un lado, las versiones más rápidas de CARA e HIAC; por otro, las versiones estándares y, por último, las de mayor calidad media.

	Σ	desv_UB	desv_CPM	med_CPU
CARA(5,20)	76503	2.76	34.78	8.13
CARA	76356	2.58	34.53	17.00
CARA(20,40)	76206	2.42	34.27	22.55
HIAC(10,10)	75384	1.62	32.84	4.96
HIAC	75009	1.17	32.18	14.52
HIAC_look_ahead_Paralelo	74671	0.81	31.58	59.43

Tabla 5.5. Mejores versiones de CARA e HIAC en j120.

HGA5000 mejora en calidad y tiempo a todas las versiones de CARA y a HIAC(10,10), mientras que HGA10000 y HGA25000 mejoran en ambas medidas a HIAC. HGA25000 y HGA100000 obtienen una calidad media superior que HIAC_look_ahead_Paralelo, con un tiempo medio inferior. Es decir, para cada versión de CARA e HIAC existe un número de secuencias de manera que HGA con ese límite mejora en calidad y tiempo al algoritmo en cuestión. Es por ello que se puede asegurar que HGA es mejor que CARA e HIAC en j120.

En j90 (j60) CARA e HIAC obtenían desv_CPM = 11.12 (11.45)% y 10.99 (10.44)% con un tiempo medio de 4.63 (2.76) y 2.53 (1.14) segundos respectivamente. HGA5000 mejora a CARA en ambos conjuntos y es similar a HIAC en ellos. En los dos conjuntos HGA emplea varias veces menos tiempo que ellos. En j30 HGA mejora a HIAC y es similar a CARA, pero éste último emplea 5 veces más tiempo medio.

Comparación con Hartmann + DJ

De los algoritmos desarrollados en el capítulo 3 con un límite de 5000 secuencias, Hartmann + DJ era el mejor, tanto en tiempo medio como en calidad. En la Tabla 5.6 se ofrecen las medidas usuales para Hartmann + DJ 5000 y 10000 sobre j120.

	Σ	desv_UB	desv_CPM	med_CPU
Hartmann + DJ 5000	75616	1.74	33.24	1.60
Hartmann + DJ 10000	75327	1.44	32.73	3.04

Tabla 5.6. Hartmann + DJ 5000 y 10000.

HGA5000 mejora a Hartmann + DJ 10000 tanto en calidad (diferencias pequeñas) como en tiempo. HGA2500 es similar en calidad (mejor desv_CPM y peor desv_UB) a Hartmann + DJ 5000, pero emplea menos tiempo medio. De hecho, se cumple en estos casos que Hartmann + DJ con 2k secuencias tarda aproximadamente un 50% más que HGA con k secuencias, que a su vez emplea aproximadamente un 33% más de tiempo que Hartmann + DJ con k soluciones. Dado que HGA con k secuencias obtiene una calidad al menos similar a Hartmann + DJ con 2k secuencias, se puede afirmar que HGA supera a Hartmann + DJ.

La rapidez de HGA

Al comparar HGA con el resto de algoritmos se puede observar que el primero es sensiblemente más rápido que la mayoría de ellos (salvo Hartmann (1) y, quizás, Hartmann (2) y Alcaraz y Maroto), incluso teniendo en cuenta los ordenadores. Para dar otro dato de su rapidez, hemos calculado el tiempo necesario en secuenciar 5000 soluciones aleatorias. Sin tener en cuenta el tiempo necesario para calcular las 5000 listas de actividades aleatorias, el tiempo medio ha sido de 2.32 segundos y, teniéndolo en cuenta, de 4.50. Estos tiempos son superiores al de HGA, especialmente el segundo. Esto corrobora la rapidez de los diversos procedimientos empleados en HGA, en particular del operador de cruce de los picos y de la justificación, las mayores diferencias con Hartmann (1).

Justificación por elegibles

En el capítulo 4 hemos conseguido mejorar la calidad de Hartmann + DJ 5000 al justificar por elegibles en lugar de por extremos. Esto también se puede conseguir con HGA5000. La nueva versión aplica la misma regla que Hartmann + DJ para la elección

de la actividad a justificar, pero no aplica DJ por elegibles a todas las secuencias sino, únicamente, hasta que se han calculado en total 1000 secuencias, además de a todos los individuos obtenidos por β-Biased al comienzo de la segunda mitad de HGA. El resto de secuencias se justifica doblemente mediante la justificación por extremos. Los parámetros han sido POPsize = 50 y π = 0.5. La Tabla 5.7 muestra los resultados del algoritmo en cuestión en j120.

HGA5000	Σ	desv_UB	desv_CPM	med_CPU
Justificación extremos	75222	1.36	32.54	2.03
Justificación elegibles	75065	1.17	32.27	4.74

Tabla 5.7. HGA5000 con justificación por extremos y por elegibles.

Como se puede observar, se experimenta una cierta mejora en las desviaciones medias. El tiempo medio empleado por la nueva versión es mucho mayor, hasta el punto de que HGA10000 (justificación por extremos) lo supera en calidad y tiempo, pero hay que tener en cuenta que no ha sido optimizada la programación de la justificación por elegibles (para admitir diversas reglas de elección de la actividad a justificar) y que, aun así, su tiempo medio no es excesivo. En cualquier caso, HGA5000_elegibles marca un nuevo mínimo en los algoritmos con un límite de 5000 secuencias, mejorando en más de un 3 (≈1.82)% la mejor desviación con respecto al CPM (UB) de la literatura. Teniendo en cuenta que se puede mejorar el tiempo medio y la regla de elección utilizada, la justificación por elegibles (aunque sea en combinación con la justificación por extremos) parece ser prometedora de cara al futuro.

Adaptación de HGA a problemas similares al RCPSP

Al igual que el algoritmo de Hartmann, HGA se puede adaptar fácilmente a otros problemas de secuenciación de proyectos con recursos limitados. Por ejemplo, al RCPSP_inter y al RCPSP con múltiples modos y recursos no renovables, cambiando adecuadamente la codificación. El operador de cruce de los picos no sería necesario modificarlo en el segundo caso y mínimamente en el primero, para que pudiera trabajar con partes de actividades además de con actividades. Resulta más sencillo incluso adaptar HGA cuando la única variación con respecto al RCPSP es en la función objetivo; en ese caso basta modificar la manera de calcular la calidad de cada individuo. Dado que el operador de los picos acorta las secuencias y mejora la utilización de recursos, HGA puede obtener buenos resultados en la minimización de

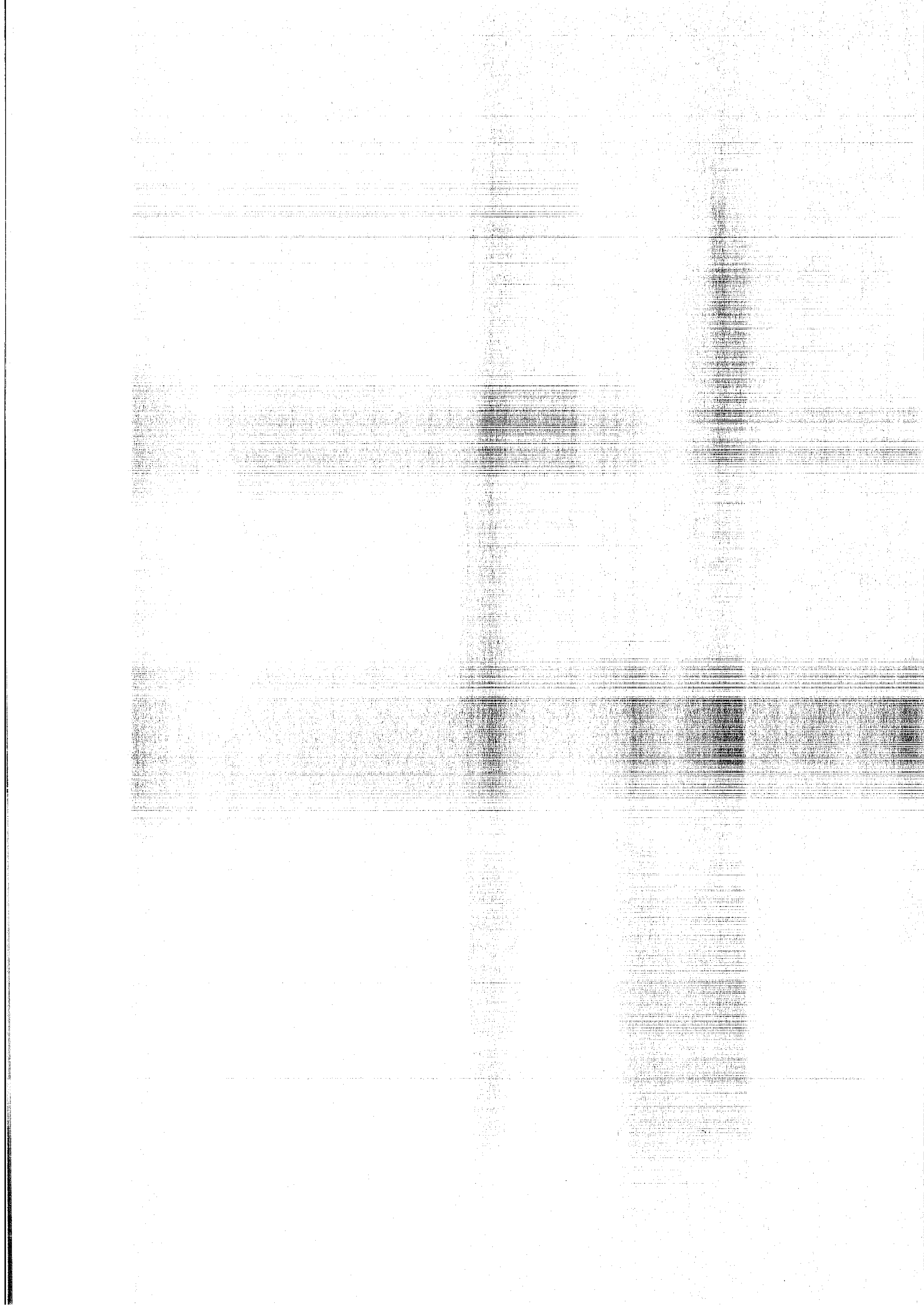
los tiempos de inicio, minimización de las fechas de entrega, etc., especialmente si se emplea una justificación adecuada (ver apartado 6 del capítulo 4).

Mejoras en HGA

Existen formas de mejorar la calidad y/o el tiempo de computación de HGA. Una de ellas ha sido comentada anteriormente y consiste en reemplazar la justificación por extremos por la justificación por elegibles, al menos parcialmente, una vez se haya mejorado la rapidez y eficacia de esta última. Otra vía de mejora pasa por adaptar el número de fases y/o el número de secuencias en cada fase a la instancia concreta que se esté resolviendo. El propio algoritmo podría detectar si la población actual está demasiado evolucionada (por ejemplo, si ha transcurrido un cierto número de iteraciones sin que se mejore el mejor individuo) y conviene comenzar una nueva fase con una nueva población. La forma de construir las nuevas poblaciones también puede modificarse, introduciendo soluciones cercanas a un conjunto de soluciones élite encontradas a lo largo del algoritmo, o calculando soluciones que contengan nuevas características no estudiadas hasta ese momento.

Un campo importante para la investigación reside en la posibilidad de combinar los picos de dos o más soluciones. Esto presenta diversos problemas que se tratan en el capítulo siguiente, así como algunas técnicas y algoritmos relacionados.

Capítulo 6
Resultados teóricos sobre la combinación de picos



1. Introducción

El concepto de pico ha resultado útil para construir un algoritmo (HGA) muy efectivo que combina la rapidez en obtener soluciones de buena calidad con la capacidad de producir soluciones de una calidad excelente si se permite que emplee más tiempo de computación. Además, los picos son elementos naturales de una secuencia, que pueden forman parte de muy diversos algoritmos.

Recordemos que un pico no es más que una lista ordenada de actividades que han demostrado poder secuenciarse juntas de una cierta manera aprovechando eficazmente los recursos. El operador de cruce de los picos obtiene la lista de picos disjuntos de una secuencia (el padre), los inserta en el hijo y completa la solución con el resto de actividades según el orden dictado por otra secuencia (la madre). Pero la madre tiene sus propios picos que podrían combinarse con los del padre para obtener nuevos individuos. Dando un paso más podemos pensar en generar nuevas soluciones a partir de un conjunto de picos obtenidos de las secuencias generadas hasta el momento. En este capítulo, primero se analizan las dificultades que plantea combinar picos provenientes de diferentes soluciones. A continuación, mediante la introducción de ciertos grafos dirigidos auxiliares, se caracterizan lo que hemos denominado cordilleras aceptables: conjuntos de picos que pueden dar lugar a nuevas soluciones. También se presenta un procedimiento para generar todas las cordilleras aceptables maximales. Finalmente, se proponen métodos para generar secuencias a partir de cordilleras aceptables y se indican diversos esquemas algorítmicos para el RCPSP que utilizan los conocimientos desarrollados en este capítulo.

2. Resultados teóricos

Vamos a suponer que disponemos de un conjunto $\mathbf{P}$ de picos obtenidos de diferentes soluciones. Nuestro objetivo es encontrar maneras de obtener nuevas secuencias que contengan algunos de esos picos.

Un pico es una sublista de una representación AL de una secuencia. Desde un punto de vista gráfico, un pico puede considerarse como un subconjunto de nodos del grafo del proyecto ordenados según un cierto orden topológico. Veamos un ejemplo. En el grafo de la Figura 6.1 tenemos marcados subconjuntos de nodos que corresponden a tres picos, {2,3,4}, {6,7,9,10} y {8,10,11}. En principio, no todas las actividades tienen

por qué estar en un pico (5 no pertenece a ninguno), pero una actividad sí puede estar en más de un pico (10 pertenece a los picos 2 y 3). Para completar la especificación de los picos es necesario indicar la ordenación de los nodos dentro de cada pico que podría ser, por ejemplo, la siguiente: P_1 = (2, 3, 4), P_2 = (6, 9, 10, 7) y P_3 = (10, 8, 11). Una ordenación distinta de los nodos correspondería a un conjunto de picos distintos: por ejemplo, P'_1 = (2, 4, 3), P'_2 = (9, 6, 10, 7) y P'_3 = (10, 11, 8).

Fijémonos en que si un pico contiene una actividad predecesora (no necesariamente inmediata) y una actividad sucesora (no necesariamente inmediata) de una actividad i entonces el pico también contiene la actividad i. A esta propiedad la denominaremos propiedad **sandwich**.

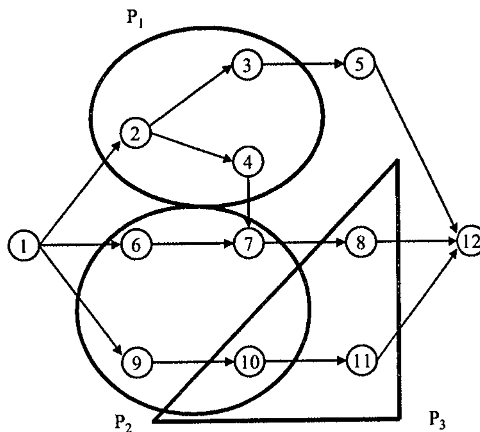


Figura 6.1

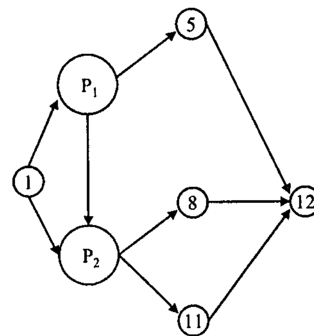


Figura 6.2

Para encontrar nuevas soluciones a partir de **P** vamos a desarrollar una técnica que generaliza la utilizada en HGA: seleccionaremos un subconjunto de picos de **P** y construiremos una lista de actividades que contenga los picos seleccionados como sublistas. No todos los subconjuntos de picos son apropiados para esta técnica. Por ejemplo, supongamos que hemos seleccionado dos picos que tienen actividades en común. En este caso, la nueva lista de actividades no podrá, generalmente, contener los dos picos como sublistas puesto que se repetirían las actividades comunes. Una posibilidad para resolver este problema sería eliminar de uno de los picos las actividades comunes, pero entonces podría ocurrir que la utilización de los recursos en el pico reducido no fuera suficientemente buena, por lo que no tendría ya sentido incluirlo en la nueva lista de actividades. Además está la complejidad añadida de, dado un subconjunto de picos que no son disjuntos dos a dos, determinar el orden en que

se reducen los picos. A continuación vamos a caracterizar por medio de un grafo auxiliar los subconjuntos de picos que son adecuados para la técnica que vamos a desarrollar.

2.1. Cordillera aceptable. El grafo de una cordillera

Definición: Cordillera

Denominaremos **cordillera** a un conjunto $C = \{P_1, ..., P_h\}$ de picos disjuntos dos a dos, i.e., no existe ninguna actividad que pertenezca a más de un pico.

Definición:

Diremos que una lista de actividades λ **contiene los picos de** $C = \{P_1, ..., P_h\}$ si las actividades de cada pico P_i ocupan posiciones consecutivas en λ en el mismo orden que en P_i.

Fijémonos en que una cordillera no es una lista ordenada de picos, por lo que las listas de actividades que contienen los picos de una cordillera los pueden contener en distinto orden.

Definición: Grafo de una cordillera

Definimos el **grafo de la cordillera C**, **G(C)**, como el grafo dirigido que se obtiene del grafo del proyecto G condensando cada conjunto de actividades que forma un pico en un nuevo nodo, al que denominaremos **supernodo**.

Así pues, el grafo de una cordillera C está formado por nodos **simples**, las actividades que no pertenecen a ningún pico de la cordillera, y por los supernodos, uno por cada pico, que representan todas las actividades que pertenecen a ellos. Como los picos son disjuntos dos a dos, G(C) está bien definido. Las aristas dirigidas de G(C) se definen de forma natural a partir de las relaciones de precedencia de G, es decir:

1) Sean i, j nodos simples, entonces $i \rightarrow j$ en G(C) si $i \rightarrow j$ en el grafo original.

2) Sean $i = I$ un supernodo y j un nodo simple, entonces $I \rightarrow j$ en G(C) si $\exists\, h \in I \,/\, h \rightarrow j$ en G.

3) Sean i un nodo simple y $j = J$ un supernodo, entonces $i \rightarrow J$ en G(C) si $\exists\, h \in J \,/\, i \rightarrow h$ en G.

4) Sean $i = I$ y $j = J$ supernodos, entonces $I \rightarrow J$ si $\exists\, r \in I,\ s \in J \,/\, r \rightarrow s$ en G.

Donde si i y j son dos nodos de un determinado grafo, la notación i ⇒ j (i → j) indica que el nodo i es predecesor (inmediato) del nodo j.

Si en el grafo de la Figura 6.1 consideramos la cordillera C = {P_1, P_2}, entonces el grafo de la Figura 6.2 muestra el grafo G(C).

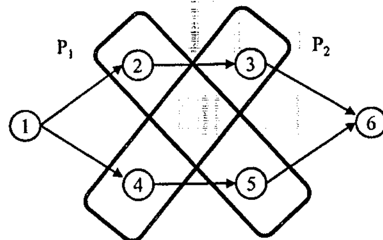


Figura 6.3 a): G; C = {P_1, P_2}

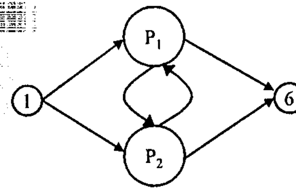


Figura 6.3 b): G(C)

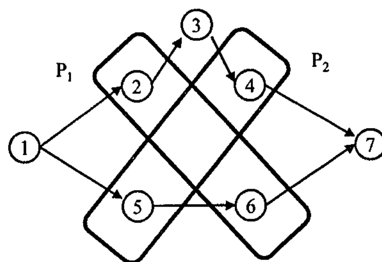


Figura 6.4 a): G; C = {P_1,P_2}

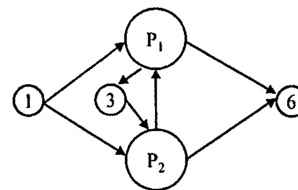


Figura 6.4 b): G(C)

Analicemos la existencia de circuitos o ciclos dirigidos en G(C). En G(C) no pueden existir circuitos que estén formados solamente por nodos simples pues G es acíclico. Por la propiedad sandwich de los picos tampoco pueden existir circuitos formados por uno o varios nodos simples y por un único supernodo. Sin embargo, sí que pueden existir circuitos que contengan dos o más supernodos como muestran los ejemplos de las Figuras 6.3 a) y b), 6.4 a) y b) y 6.5 a) y b).

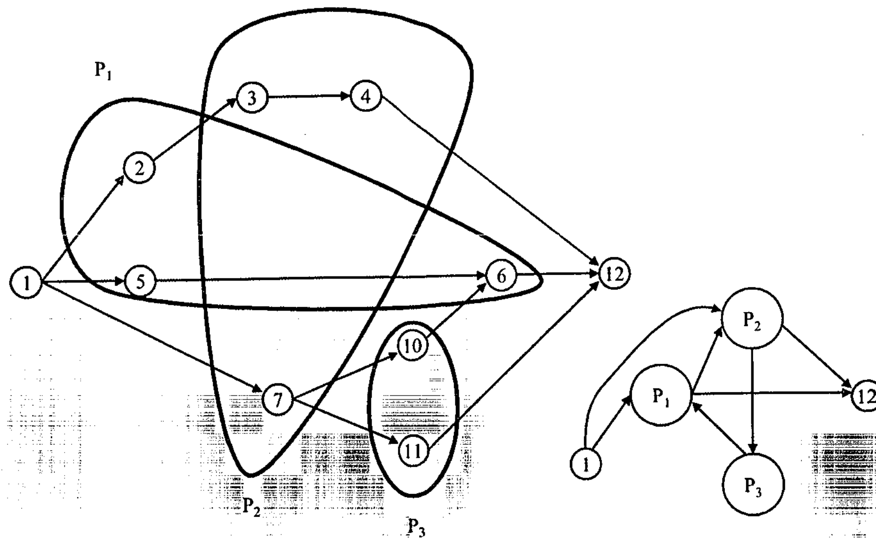


Figura 6.5 a): G; C = {P_1, P_2, P_3}   **Figura 6.5 b)**: G(C)

Ahora bien, si dos picos están en un circuito de G(C) no puede existir una lista de actividades λ que contenga ambos picos pues λ no sería compatible con las relaciones de precedencia. Así pues, es necesario que G(C) sea acíclico.

Definición: Cordillera aceptable

Denominaremos **cordillera aceptable** a toda cordillera C tal que el grafo G(C) es acíclico.

De la definición de cordillera aceptable se deduce que si la cordillera C = {P_1, P_2} no es aceptable, tampoco lo es cualquier cordillera que contenga los picos P_1 y P_2.

La siguiente proposición muestra que si C es una cordillera aceptable, entonces obtener listas de actividades de G que incluyan los picos de C es equivalente a encontrar listas de actividades de G(C).

PROPOSICIÓN 6.1

Sea C = {P_1, ..., P_h} una cordillera aceptable. Entonces, λ es una lista de actividades de G que contiene los picos de C $\Leftrightarrow$ λ' es una lista de actividades de G(C), donde λ' se obtiene de λ (y viceversa) reemplazando las sublistas de actividades que definen los picos por los supernodos.

Demostración

La demostración es sencilla si tenemos en cuenta que la posición relativa de un par de nodos simples es la misma en λ que en λ', que los picos son disjuntos y que las actividades de un pico ocupan posiciones consecutivas en λ.

Q.E.D.

Ejemplo

Las listas de actividades (1 P_1 P_2 8 5 11 12) y (1 P_1 5 P_2 11 8 12) del grafo G(C) de la Figura 6.2 se corresponden con las listas de actividades (1 2 3 4 6 9 10 7 8 5 11 12) y (1 2 3 4 5 6 9 10 7 11 8 12) del grafo G de la Figura 6.1, respectivamente.

2.2. Caracterización de las cordilleras aceptables. El grafo de los picos

Hemos visto que para generar nuevas secuencias que contengan subconjuntos de picos de $\mathbf{P}$ estos deben ser cordilleras aceptables. En esta sección vamos a presentar una caracterización de las cordilleras aceptables que posteriormente será utilizada para generarlas. Necesitamos introducir algunas definiciones.

Dados dos picos I y J de $\mathbf{P}$, diremos que:

- I es **predecesor** de J **respecto de G** (I ➔ J) si $\exists$ i $\in$ I, $\exists$ j $\in$ J / i $\Rightarrow$ j en G.
- I y J son **independientes respecto de G** si I no es predecesor respecto de G de J ni viceversa.

El grafo de los picos, GP, es un grafo dirigido que tiene un nodo por cada pico de $\mathbf{P}$ y tal que existe una arista del pico I al pico J (I $\rightarrow$ J) sii I $\cap$ J = $\varnothing$ y J no es predecesor de I respecto de G, es decir, no se cumple J ➔ I.

Así pues, dados dos picos I y J de $\mathbf{P}$ puede ocurrir uno de los siguientes casos:

- Si I $\cap$ J $\neq \varnothing$ o si [I ➔ J y J ➔ I] entonces no existirá ninguna arista entre ellos.
- Si I $\cap$ J = $\varnothing$, I ➔ J pero no es cierto que J ➔ I, entonces se cumplirá I $\rightarrow$ J, pero no J $\rightarrow$ I .
- Si I $\cap$ J = $\varnothing$ e I y J son independientes respecto de G, entonces se cumplirán I $\rightarrow$ J y J $\rightarrow$ I a la vez. Es decir, I y J estarán en un circuito de orden 2. Una arista que una dos picos independientes la denominaremos **especial**.

Las Figuras 6.6 a) y b) ilustran estos conceptos. En el grafo de la Figura 6.6 a) están marcados los cuatro picos que forman **P** mientras que el grafo de los picos **GP** aparece en la Figura 6.6 b). La única pareja de picos con intersección no vacía son P_2 y P_4, que, por tanto, no están relacionados en **GP**. Los picos P_1 y P_2 son independientes respecto del grafo original G porque ni 6 ni 7 son ni sucesores ni predecesores de 2 ó 3; esto se refleja en una doble arista entre ambos en **GP**. El pico P_1 es un predecesor respecto de G de P_3 y P_4, porque el nodo $2 \in P_1$ es predecesor del $5 \in P_3$ y del $8 \in P_4$; como, además, P_3 y P_4 no son predecesores de P_1 entonces existen sendas aristas de P_1 a P_3 y de P_1 a P_4. Por razones análogas, existen sendas aristas de P_3 a P_2 y de P_3 a P_4.

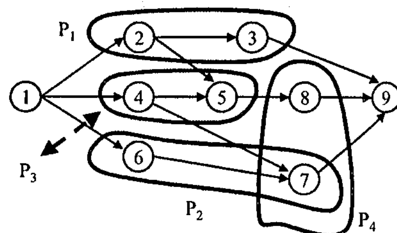


Figura 6.6 a): G

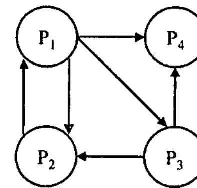


Figura 6.6 b): GP

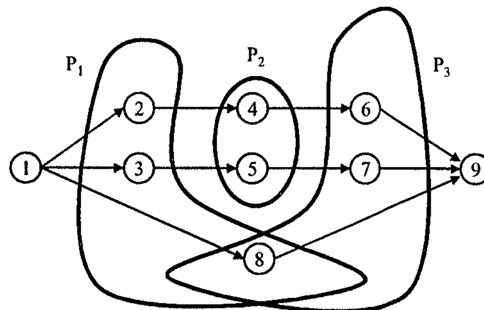


Figura 6.7 a): G; P = {P_1, P_2, P_3}

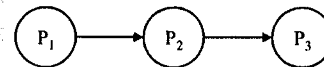


Figura 6.7 b): GP

CARACTERÍSTICAS Y PROPIEDADES DEL GRAFO DE LOS PICOS

1) **GP** puede ser disconexo. Si en la Figura 6.7 a) sólo existieran los picos P_1 y P_3, el grafo correspondiente lo sería.

2) Es evidente que existe en **GP** una arista entre dos picos sii la cordillera formada por esos dos picos es aceptable. Sin embargo no es cierto que si $I \rightarrow J$ y $J \rightarrow K$ en **GP**, entonces C = {I, J, K} sea una cordillera aceptable como muestra el ejemplo de la Figura 6.7.

3) Si la arista I → J de **GP** no es especial, entonces existe un camino que va de I a J en G(C) donde C es cualquier cordillera que contenga los picos I y J. La demostración se presenta en la proposición 6.2. Por lo tanto, si un conjunto de picos forman un circuito en **GP** que no utiliza aristas especiales entonces no puede formar parte de una cordillera aceptable.

4) Si no existe ninguna arista entre dos picos I y J es porque se cumple $I \cap J \neq \emptyset$ o (I ➔ J y J ➔ I) o ambas condiciones a la vez. En cualquier caso, los picos I y J no pueden formar parte de una cordillera aceptable. Por lo tanto, si una cordillera es aceptable, tiene que haber en **GP** al menos una arista entre cada par de picos de la cordillera. Así pues, el subgrafo generado por los picos de una cordillera aceptable tiene que ser un clique de **GP**. Las Figuras 6.8 a) y b) muestran que la afirmación contraria no es cierta. El subgrafo generado por P_1, P_2 y P_3 es un clique y, sin embargo, C = {P_1, P_2, P_3} no es una cordillera aceptable pues G(C) no es acíclico. Por otra parte, la cordillera C = {P_1, P_2, P_3} de la Figura 6.9 sí es aceptable.

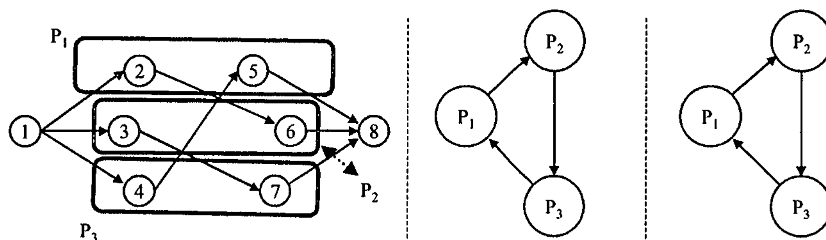


Figura 6.8 a): G, C = {P_1, P_2, P_3} **Figura 6.8 b):** **GP** **Figura 6.8 c):** G(C) sin aristas con 1 y 8

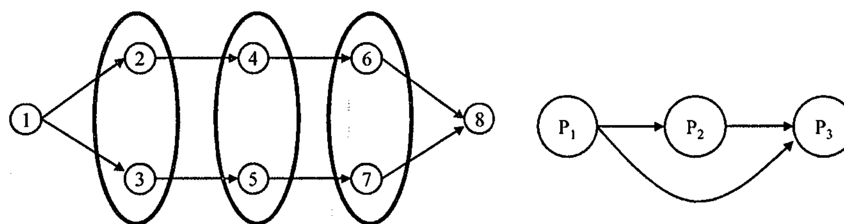


Figura 6.9 a): G **Figura 6.9 b):** **GP**

PROPOSICIÓN 6.2

Sea C una cordillera que contiene los picos P_1 y P_2 y supongamos que existe una arista no especial $P_1 \rightarrow P_2$ en **GP**. Entonces, existe un camino en G(C) que va de P_1 a P_2.

Demostración

C es una cordillera, por lo que sus picos son disjuntos dos a dos. Por hipótesis $P_1 \rightarrow P_2$ en **GP** con una relación no especial, por lo que se cumple $P_1 \Rightarrow P_2$, es decir, $\exists\, r \in P_1$, $s \in P_2 \,/\, r \Rightarrow s$ en G. Esto quiere decir que existe un camino L = $\{r\ i_1 \ldots i_h\ s\}$ en G que une r y s. A partir de L, vamos a construir un camino L' en G(C) que una P_1 y P_2. Para ello recorreremos en orden los nodos de L y cada uno de ellos dará lugar a como máximo un nodo de L'. Los nodos de L' mantendrán el orden relativo de los nodos de L que los originaron. Si un nodo de L es simple, dicho nodo formará parte de L'. Si por el contrario el nodo pertenece a un pico P, el supernodo P formará parte de L', a no ser que el anterior nodo de L' ya sea P. Como $r \in P_1$, $s \in P_2$ y $P_1 \cap P_2 = \varnothing$, L' es un camino en G(C) que une P_1 con P_2.

Q.E.D.

COROLARIO 6.3

Sea C = $\{P_1, \ldots, P_h\}$ una cordillera tal que L = $(P_1, \ldots, P_h)$ es un circuito en **GP** que no utiliza aristas especiales. Entonces, el conjunto $\{P_1, \ldots, P_h\}$ forma parte de un circuito en G(C).

Demostración

Evidente.

El camino en G(C) que une P_i con P_{i+1} en el corolario anterior puede contener alguno de los otros picos, como se puede observar en las Figuras 6.10 a), b) y c), en las que el camino que une P_2 y P_3 pasa por P_5, por lo que el circuito al que pertenece $\{P_1, P_2, P_3, P_4, P_5\}$ en G(C) es L' = $(P_1\ P_2\ P_5\ P_3\ P_4\ P_5)$.

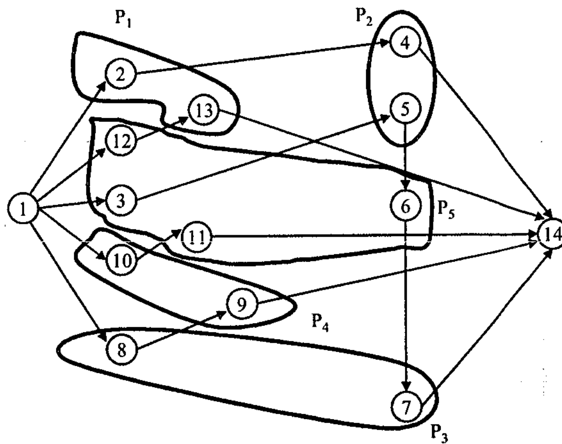


Figura 6.10 a): G

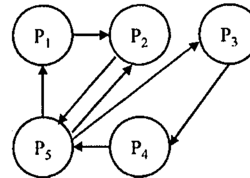


Figura 6.10 b):
G(C) (sin aristas con 1 y 14)

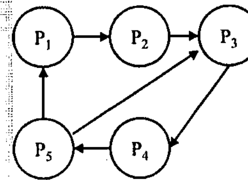


Figura 6.10 c): GP

Veamos ahora los resultados necesarios para caracterizar las cordilleras aceptables. Primero demostraremos una proposición en las que las hipótesis son relativamente restrictivas.

Si C = $\{P_1, ..., P_h\} \subseteq \mathbf{P}$, denotaremos por **G[C]** al subgrafo de **GP** generado por los picos de C. Diremos que G[C] es un clique si existe al menos una arista en **GP** entre cada pico de C.

PROPOSICIÓN 6.4

Sea C = $\{P_1, ..., P_h\} \subseteq \mathbf{P}$ tal que G[C] es un clique acíclico. Entonces, C es una cordillera aceptable.

Demostración

1) C cordillera

Sea P, Q $\in$ C, P $\neq$ Q. Por ser G[C] clique, existe en G[C] una relación entre P y Q. Entonces, por definición, P $\cap$ Q = $\varnothing$, por lo que todos los picos son disjuntos 2 a 2 y, por tanto, C es cordillera.

2) C aceptable

Por reducción al absurdo.

Supongamos que existe un ciclo dirigido en G(C). Podemos suponer, sin pérdida de generalidad, que el ciclo es de la forma $L = (P_1\ i_{11}\ i_{12} \ldots i_{1n_1}\ P_2 \ldots P_k\ i_{k1}\ i_{k2} \ldots i_{kn_k})$, con $k \geq 2$, $n_j \geq 0\ \forall j$, i_{rs} nodo simple $\forall$ r, s.

Sea $r \in \{1, \ldots, k-1\}$ tal que $n_r > 0$. Entonces, $P_r \rightarrow i_{r1}$ en G(C). Por definición de G(C), $\exists\ j \in P_r\ /\ j \rightarrow i_{r1}$ en G, por lo que, $j \Rightarrow i_{m_r}$ en G. Además, $i_{m_r} \rightarrow P_{r+1}$,en G(C), por lo que $\exists\ s \in P_{r+1}\ /\ i_{m_r} \rightarrow s$ en G .Si lo juntamos con lo anterior $j \Rightarrow s$ en G, i.e., $P_r \Rightarrow P_{r+1}$ (siempre que existe un camino formado por nodos simples entre dos picos P_i y P_j se cumple $P_i \Rightarrow P_j$). Si $n_r = 0$, entonces directamente $P_r \Rightarrow P_{r+1}$.

Pero G[C] clique, lo que implica que P_r y P_{r+1} están relacionados, por lo que, como $P_r \Rightarrow P_{r+1}$, necesariamente $P_r \rightarrow P_{r+1}$ en **GP**.

Análogamente $P_k \rightarrow P_1$ en **GP** por lo que $L' = (P_1\ P_2 \ldots P_k)$ es un ciclo dirigido en G[C], lo que es una contradicción.

Q.E.D.

Esta propiedad nos proporciona una manera de obtener cordilleras aceptables; basta con localizar cliques acíclicos en **GP**.

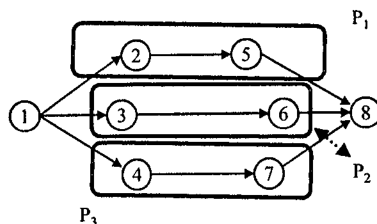


Figura 6.11 a): G

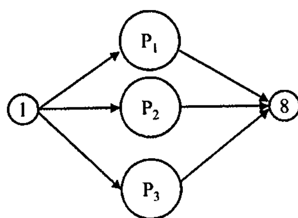


Figura 6.11 b): G(C)

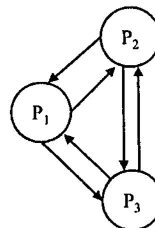


Figura 6.11 c): GP

Las Figuras 6.11 a), b) y c) muestran un ejemplo de que el recíproco no es cierto, dado que **GP** (Figura 6.11 c)) no es acíclico y C es aceptable. Este contraejemplo plantea la posibilidad de que la implicación 'C cordillera aceptable $\rightarrow$ G[C] acíclico' no sea cierta debido a las relaciones especiales. Veamos en la siguiente proposición que esta

afirmación es cierta. Como notación adicional definimos **G[C]'** = G[C] \ {aristas especiales}.

PROPOSICIÓN 6.5

Sea C = {P_1, ..., P_h} ⊆ **P** una cordillera aceptable. Entonces, G[C]' es acíclico.

Demostración

Por reducción al absurdo. Supongamos que existe un ciclo dirigido en G[C]'. Este ciclo sería un ciclo dirigido en **GP** que no contiene aristas especiales. Por el corolario 6.3, se tiene que L forma parte de un ciclo dirigido en G(C), lo que es una contradicción.

Q.E.D.

TEOREMA 6.6. CARACTERIZACIÓN DE LAS CORDILLERAS ACEPTABLES

Sea C = {P_1, ..., P_h} ⊆ **P**.

C es una cordillera aceptable ⇔ G[C] es un clique y G[C]' es acíclico.

Demostración

(→) Por la propiedad 4 y la proposición 6.5, G[C] es clique y G[C]' es acíclico.

(←) G[C] clique, lo que implica que los picos de C son disjuntos dos a dos, i.e., C es una cordillera.

Vamos a demostrar que G(C) es acíclico por reducción al absurdo. Sea L un ciclo dirigido de G(C).

Podemos escribir L como (P_1 i_{11} i_{12} ... i_{1n_1} P_2 ... P_k i_{k1} i_{k2} ... i_{kn_k}), con $k \geq 2$, $n_j \geq 0$ ∀j, i_{rs} nodo simple ∀ r,s. Bastará demostrar que L' = (P_1 P_2 ... P_k) es ciclo dirigido en G[C]', porque sería una contradicción con las hipótesis.

Sea i ∈ {1, ..., k-1}. Por lo visto en la demostración de la proposición 6.4, P_i ➔ P_{i+1}. Como, además, G[C] es un clique y teniendo en cuenta la definición de **GP**, se tiene que cumplir $P_i \to P_{i+1}$ en G[C] y no se puede cumplir la relación contraria. Así pues, la relación en **GP** de P_i a P_{i+1} no es especial. Análogamente, existe una relación no especial entre P_k y P_1 en G(C), por lo que queda demostrada la contradicción.

Q.E.D.

Con este teorema ya tenemos completamente caracterizadas las cordilleras aceptables. Para obtenerlas, basta aplicar algoritmos para encontrar cliques que al quitarle los 2-ciclos (≡ aristas especiales) sean acíclicos. Vamos a describir un algoritmo sencillo para ello, pero no queremos tener que distinguir entre aristas especiales y no especiales. Veamos a continuación unos resultados que nos permiten soslayar esta dificultad.

2.3. Generación de cordilleras aceptables

PROPOSICIÓN 6.7

Sea $C \subseteq \mathbf{P}$. C es una cordillera aceptable si y sólo si existe una manera de eliminar una arista entre cada par de picos independientes en G[C] de manera que el subgrafo resultante (:= G[C]") sea un clique acíclico.

Demostración

Los picos independientes tienen dos aristas especiales entre ellos sii tienen intersección vacía, en otro caso no tienen ninguna arista.

(←) Tenemos que demostrar que G[C] es clique y G[C]' es acíclico. Si G[C]" es clique, entonces G[C] = G[C]" ∪ {la arista especial que ha sido eliminada entre cada par de picos independientes} es clique.

Como G[C]" es clique, todos los picos de C, entre ellos los independientes, son disjuntos dos a dos. Consideramos P y Q dos picos independientes de C. Como son disjuntos, existen dos aristas especiales entre ellos en G[C] y una en G[C]". Por lo tanto, G[C]' = G[C]" \ {la arista especial que queda entre cada par de picos independientes} es obviamente acíclico al serlo G[C]".

(→) Por el teorema 6.6, G[C] es clique y G[C]' es acíclico. Como C es cordillera todos sus picos son disjuntos dos a dos, por lo que existen en G[C] dos aristas especiales entre cada par de picos independientes.

Puesto que G[C] es clique, cualquier G[C]" que construyamos a partir de eliminar una arista entre cada par de picos independientes lo será también, porque sólo se elimina una de las dos existentes. Falta demostrar que existe una manera de crear G[C]" para que sea acíclico.

Puesto que G[C]' es acíclico, existe al menos una ordenación de los picos C = $\{P_{i_1}, ..., P_{i_k}\}$ de manera que no hay aristas en G[C]' entre P_{i_r} y P_{i_s} si $i_r > i_s$. Escogemos una de esas ordenaciones. Podemos renombrar los picos de manera que el orden sea $P_1, ..., P_k$, por lo que no existirán aristas en G[C]' entre P_i y P_j si $i > j$.

Sean P_i y P_j, $i < j$, dos picos independientes de C. Entonces, no existe en G[C]' ninguna arista entre ellos. Añadimos a G[C]' la arista $P_i \rightarrow P_j$. Esta arista existe en G[C] y es especial, porque lo picos son disjuntos. Obviamente G[C]'' así construido, G[C]'' = G[C]' $\cup$ $\{(P_i,P_j) / i < j$ y P_i y P_j independientes$\}$ es acíclico, ya que no existe ninguna arista entre dos picos P_i y P_j con $i > j$.

Q.E.D.

Ejemplo

En la Figura 6.12 se puede visualizar una forma de construir G[C]'' a partir de G[C] clique cíclico. Para cada posible ordenación en G[C]' tenemos un G[C]'' clique acíclico distinto.

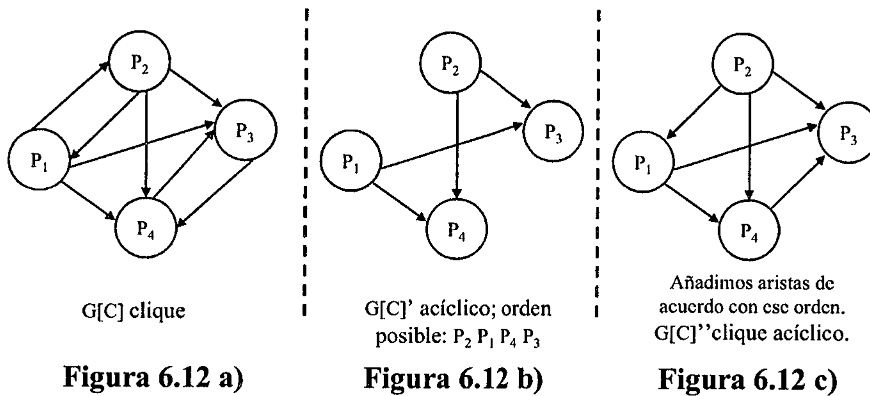


Figura 6.12 a) **Figura 6.12 b)** **Figura 6.12 c)**

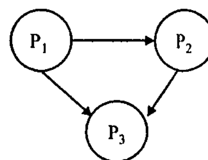


Figura 6.13: GP

COROLARIO 6.8

Sea C $\subseteq$ $\mathbf{P}$. C es una cordillera aceptable $\Leftrightarrow$ Existe un subgrafo de G[C] que es un clique acíclico.

Demostración

($\rightarrow$) Por el corolario anterior G[C]" $\subseteq$ G[C] es clique acíclico .

($\leftarrow$) Sea A este subgrafo. Por el corolario anterior basta demostrar que A es de la forma G[C]", i.e., tiene todas las aristas no especiales de G[C] y una única arista especial entre cada par de picos independientes.

1) Sea $P_i \rightarrow P_j$ una arista no especial de G[C]. Por definición, P_i y P_j no son independientes, por lo que sólo existe una arista entre esos dos picos en $\mathbf{GP}$ y en G[C]. Como A es clique, existe en A una arista entre P_i y P_j. Como A $\subseteq$ G[C], $P_i \rightarrow P_j$ pertenece a A.

2) Sean P_i, P_j de C independientes. Como A es clique acíclico sólo existe una arista en A entre ellos que, por ser independientes, debe ser especial.

Q.E.D.

Introducimos la siguiente notación: $\Gamma^+(P) = \{Q \in \mathbf{P} \,/\, P \rightarrow Q \text{ en } \mathbf{GP}\}$.

ALGORITMO 6.9

Sea $\mathbf{P} = \{P_1, ..., P_s\}$. Consideramos el siguiente algoritmo

1. Hacer C = $\varnothing$, D = $\mathbf{P}$.
2. Mientras (D $\neq \varnothing$)
   2.1. Elegir un pico P de D. Hacer C = C $\cup$ {P}.
   2.2. Construir D = D $\cap$ $\Gamma^+(P)$.
3. Devolver C.

COROLARIO 6.10

Cada vez que finaliza un paso 2.1 del algoritmo 6.9, C es una cordillera aceptable; en particular, el conjunto C que devuelve el procedimiento lo es. Además, este proceso es capaz de proporcionar cualquier cordillera aceptable maximal (i.e., no incluida en otra cordillera aceptable de mayor cardinal).

Demostración

1) Supongamos que acabamos de finalizar una etapa 2.1. Podemos expresar C como $\{P_1, ..., P_k\}$ en el orden en que han sido escogidos. Por el paso 2.2 se cumple que si $i < j$, entonces $P_j \in \Gamma^+(P_i)$, por lo que la arista $P_i \rightarrow P_j$ existe en **GP**. Consideramos A como el subgrafo formado por estos picos y todas estas aristas. Obviamente A es un clique pero como, además, no contiene aristas entre P_i y P_j si $i > j$, entonces A es acíclico. Además, $A \subseteq G[C]$, por lo que podemos aplicar el corolario 6.8 para demostrar que C es una cordillera aceptable.

2) Sea C una cordillera aceptable maximal de cardinal k. Por el corolario 6.8 existe un subgrafo A, $A \subseteq G[C]$ con A clique acíclico. De acuerdo con este subgrafo podemos ordenar C de una única manera $C = \{P_1, ..., P_k\}$ de forma que no haya aristas entre P_i y P_j con $i > j$, y que entre cada par de picos P_i y P_j con $i < j$ exista la arista $P_i \rightarrow P_j$ en A. O, equivalentemente, $P_j \notin \Gamma^+(P_i)$ si $i > j$ y $P_j \in \Gamma^+(P_i)$ si $i < j$. Así pues, el algoritmo 6.9 puede elegir los picos $P_1, ..., P_k$ en ese orden en el paso 2.1. Después de escoger P_k, $D = \varnothing$, pues sino existiría $Q \in D$. Pero por 1) $C \cup \{Q\}$ es una cordillera aceptable, lo que es una contradicción porque C es cordillera aceptable maximal. Por lo tanto, el algoritmo 6.9 devuelve la cordillera C.

Q. E. D.

Comentarios

1) Por extensión, podemos obtener todas las cordilleras aceptables, porque cada una de ellas está incluida en una cordillera aceptable maximal que puede ser obtenida por el algoritmo 6.9. Bastaría entonces con extraer de ella el subconjunto buscado.

2) No todas las cordilleras aceptables que proporciona el algoritmo son maximales. Si consideramos el grafo de los picos de la Figura 6.13, $C = \{P_1, P_3\}$ es una salida posible del procedimiento (en la 1ª iteración escogemos P_1 y en la 2ª iteración $D = \{P_2, P_3\}$ y escogemos P_3; en este caso en la 3ª iteración $D = \varnothing$). C no es una cordillera aceptable maximal porque $C^* = \mathbf{P}$ también es aceptable.

3) Las cordilleras que se pueden obtener mediante el algoritmo son aquellas que no son ampliables 'por el final', i.e., $C = \{P_1, ..., P_k\}$ / no existe $P \in \mathbf{P}$ de manera que todas las aristas $P_1 \rightarrow P, ..., P_k \rightarrow P$ pertenezcan a **GP**. Es decir, no existe ningún pico que pueda estar detrás de la ordenación $P_1, ..., P_k$ (aunque sí puede existir alguno que pueda estar en medio, como lo demuestra el ejemplo anterior). Como las cordilleras aceptables maximales no son ampliables por el final, en particular se pueden obtener con el algoritmo.

4) Lo más destacable de este procedimiento es que no es necesario distinguir entre aristas especiales y no especiales. $\Gamma^{+}(P)$ contiene todas las aristas que parten de P, especiales o no. Tampoco es necesario chequear si existen ciclos entre los picos que van formando C.

5) Cuando se utilice esta técnica en un algoritmo no se pretenderá, en general, generar todas las cordilleras aceptables, sino sólo un número prefijado de ellas. Dependiendo de la estrategia adoptada, las cordilleras se pueden generar de un modo aleatorio o atendiendo a un criterio. En este último caso, se plantean nuevos problemas de optimización combinatoria: dado un grafo de los picos generar las k mejores cordilleras aceptables con respecto a diversos criterios. Posibles criterios a maximizar podrían ser los siguientes: utilización de recursos de la cordillera, número de actividades en la cordillera, número de picos en la cordillera. Fijémonos que es muy fácil construir algoritmos greedy o GRASP para los dos primeros criterios. La optimización del tercer criterio estaría relacionada con el problema del camino más largo.

3. POSIBLES FORMAS DE CONSTRUIR UNA LISTA DE ACTIVIDADES QUE CONTENGA UNA CORDILLERA ACEPTABLE

Una cordillera aceptable no define por sí misma una lista de actividades. Para construir una lista de actividades a partir de una cordillera aceptable hay que tomar dos tipos de decisiones: el orden en que se colocan los picos en la lista de actividades y el orden en que se colocan las actividades que no pertenecen a ningún pico. Recordemos que las actividades de cada pico están ya ordenadas. Estas decisiones se pueden tomar simultánea o secuencialmente.

A continuación se esbozan algunas posibilidades para construir listas de actividades que contengan los picos de la cordillera aceptable C que se ha escogido previamente. Recordemos que esto es equivalente a generar listas de actividades de G(C).

(i) Aleatoriamente

Dado que G(C) es un grafo acíclico se pueden construir listas de actividades aleatorias.

(ii) Reglas de prioridad

Para utilizar reglas de prioridad en G(C) podemos definir la prioridad de un pico como el máximo (o el mínimo, dependiendo de la regla) de las prioridades de las actividades que lo forman. Al tener definidas prioridades sobre todos los elementos de G(C) es fácil construir listas de actividades a partir de ellas aplicando un SGS, bien de forma determinista o mediante muestreos. Se pueden utilizar tanto reglas estáticas como dinámicas.

(iii) Soluciones de calidad

De un modo análogo al visto en el algoritmo HGA, una solución marcaría el orden en que se secuencian los picos de la cordillera C y otra, las posiciones del resto de actividades.

4. Posibles técnicas y algoritmos relacionados con los picos

En este apartado se comentan algunas posibles técnicas y esquemas algorítmicos basados en la teoría de los picos desarrollada.

Cambios en HGA

Cada pareja de individuos A y B elegida para combinarse da lugar a más de dos hijos; los dos primeros se calculan como en el algoritmo original y los demás se obtienen escogiendo cordilleras aceptables del grafo de los picos obtenido de juntar los picos de A y B y calculando listas de actividades que las contengan. Otra posibilidad es que, además de las combinaciones del algoritmo original, se elijan k soluciones de la población en cada iteración y con sus picos se forme el grafo de los picos. Extrayendo cordilleras aceptables y construyendo listas de actividades que las contengan se pueden obtener nuevos individuos que difícilmente podrían aparecer de la manera usual.

Algoritmo genético I

En lugar de una población de individuos podríamos tener una población de picos que fuera evolucionando. En cada iteración se escogerían cordilleras aceptables para generar nuevas soluciones mediante alguna de las formas vistas en el apartado anterior. Las soluciones podrían aportar nuevos picos a la población si éstos resultaban lo suficientemente atractivos. La calidad o fitness de un pico se podría calcular de acuerdo a la utilización de recursos en él, a la calidad de la solución de la que procede o a ambos.

Algoritmo genético II

Este algoritmo codifica las secuencias por medio de las representaciones AL, cada una de las cuales se divide en k partes iguales. El procedimiento trabaja con k poblaciones en paralelo, una para cada parte. Así, la primera población contiene a lo largo del algoritmo primeras partes de soluciones, las n/k primeras actividades de diferentes representaciones AL. Las partes de una representación AL desempeñan el papel de los picos; a cada parte le asignamos un valor que representa su calidad, que es una medida ponderada de la utilización de recursos y de la duración de la secuencia. Cuando una secuencia es creada, una o varias de las partes de su representación AL se introducen en las poblaciones correspondientes si su valor es suficientemente alto. El umbral puede ser diferente para cada parte.

Al disponer de 'picos' de todas las partes de una secuencia eludimos la posibilidad de que la mayoría de ellos estén situados aproximadamente en el medio de las secuencias. Esto puede ser así porque, en general, la utilización de recursos al principio y al final de las secuencias no puede ser muy alta debido a la limitación de actividades disponibles para secuenciarse. Además, de este modo se asegura que la intersección entre picos sea mucho menor, puesto que cada población se dedica a una parte de la secuencia. A pesar de esto, a veces puede ser necesario trabajar con picos no disjuntos. Otra ventaja de este método es que se evita el problema de trabajar un número excesivo de veces con picos muy similares, que se diferencian entre sí en 2 ó 3 actividades o en cómo se combinan sus elementos.

Reencadenamiento de picos

Dadas dos secuencias S y S' podemos construir un camino entre S y S' introduciendo en S los picos de S' uno a uno. Para ello, en la etapa k se construiría una lista de actividades con k picos de S' y el resto de actividades según el orden relativo de S.

Esta idea resulta atrayente, como también lo es si en lugar de S' se trabaja con 2 ó 3 soluciones élite, se van escogiendo cada vez cordilleras aceptables C que contengan más picos de esas secuencias y se van introduciendo en S.

Intensificación / diversificación mediante picos

Conectado con la anterior técnica, ésta consistiría en mantener un 'pool' de los mejores picos encontrados a lo largo del algoritmo (cualquier tipo de algoritmo). Para provocar un efecto intensificador bastaría calcular soluciones que contuvieran

obligatoriamente algunos de estos picos, o introducir cordilleras aceptables en soluciones calculadas mediante otros procedimientos.

La diversificación mediante picos sería menos sencilla, pero podría consistir en impedir que determinadas actividades, las pertenecientes a los picos del 'pool', se secuenciaran juntas. Para ello, no se permitiría que sus posiciones en las listas de actividades fueran demasiado similares.

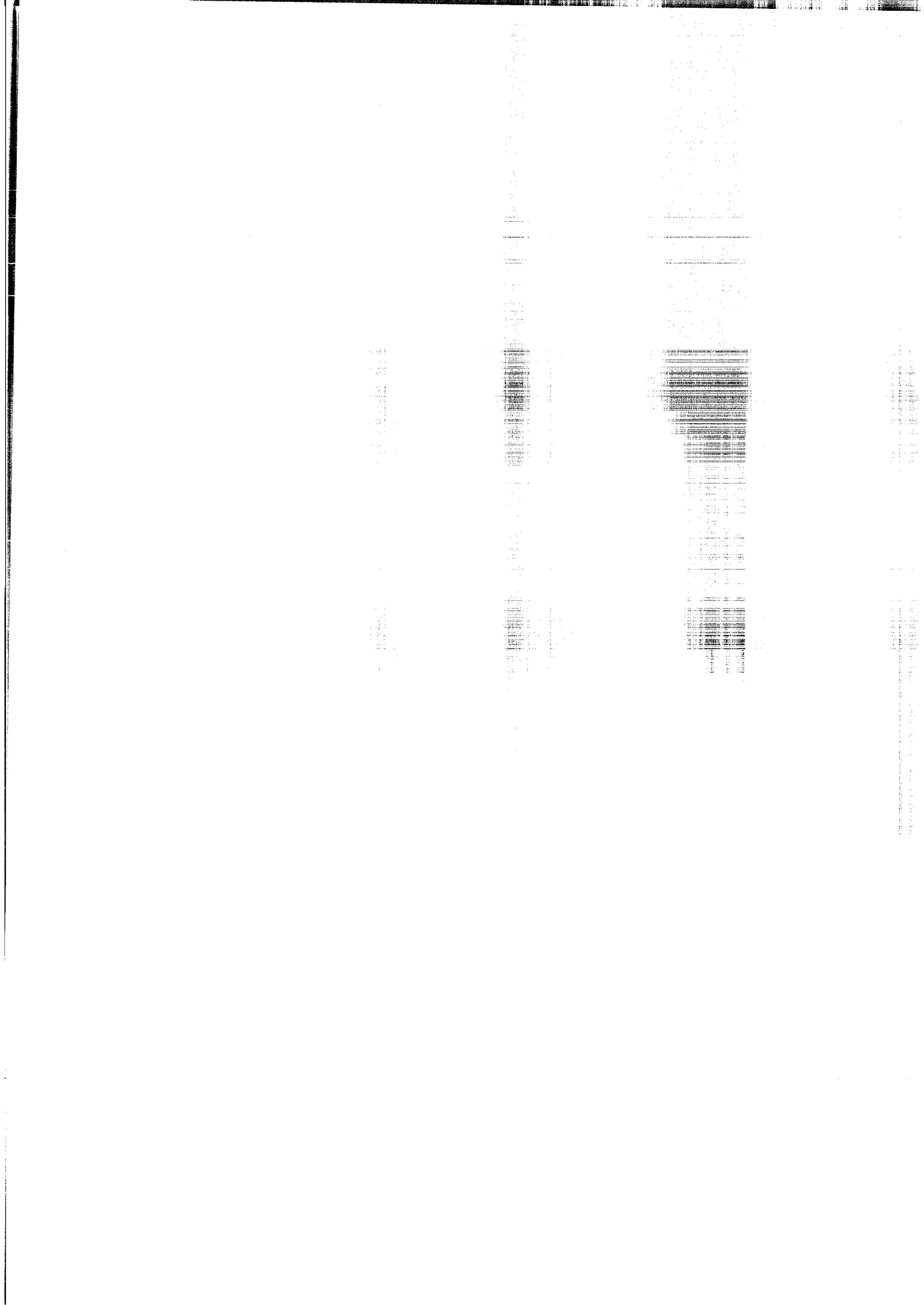
Combinar picos del grafo original y del inverso

En principio todas las técnicas comentadas se aplican sobre el grafo original, pero también se pueden aplicar sobre el grafo inverso, i.e., con todas las secuencias y los picos justificados a la derecha.

Los picos de las soluciones activas y de sus correspondientes soluciones justificadas a la derecha son presumiblemente bastante diferentes – como lo son las soluciones. Una posibilidad añadida a lo anterior es tratar de combinarlos, enlazando soluciones justificadas a la izquierda con otras a la derecha.

Capítulo 7

Conclusiones y líneas futuras de investigación



1. CONCLUSIONES

En este apartado vamos a resumir los resultados más importantes de la tesis, así como algunas de las conclusiones que se pueden extraer de los mismos.

En primer lugar hemos realizado una revisión bibliográfica de los distintos enfoques heurísticos para el RCPSP, entre los que se encuentran aproximaciones muy recientes.

Hemos desarrollado fundamentalmente tres algoritmos heurísticos nuevos para el RCPSP. El primero, CARA, trata de mejorar las secuencias concentrándose alternativamente en la cola y la cabeza. Su misión principal es la de adelantar las actividades que necesariamente deben secuenciarse antes si se quiere mejorar la duración del proyecto. Para ello utiliza información proporcionada por la regla de prioridad LFT y por la holgura dinámica. En una segunda fase se crean secuencias de una calidad similar a la de la mejor solución obtenida en la primera fase mediante un método de muestreo modificado llamado β-Biased. A las mejores secuencias se les aplica un nuevo método de muestreo que restringe considerablemente el espacio de búsqueda entorno a cada solución.

El segundo algoritmo, HIAC, es un método basado en poblaciones que incorpora diferentes estrategias para generar y mejorar una población de secuencias. HIAC consta de tres fases. En la primera se construye una población inicial y se aplica una función de mejora, HIA, a cada uno de sus elementos. HIA intenta mejorar una secuencia aumentando localmente la utilización de recursos. En la segunda fase se aplica alternativamente un procedimiento de combinar secuencias e HIA. La tercera fase comienza como la segunda de CARA, calculando mediante β-Biased soluciones cercanas a la mejor secuencia global. A las mejores se les aplica HIA, repitiendo el proceso si se produce una mejora global.

El último algoritmo, HGA, es un algoritmo genético híbrido que emplea un nuevo operador de cruce basado en el concepto de pico, una parte de una secuencia con una utilización alta de recursos. Al aplicar el operador a dos secuencias, una madre y un padre, se obtienen una hija y un hijo. La hija (hijo) hereda los picos de la madre (padre) y el resto de actividades se posiciona de acuerdo con el padre (la madre), teniendo en cuenta que el resultado debe ser una lista de actividades. HGA posee una segunda fase en la que el genético básico se aplica a una población obtenida aplicando β-Biased a la mejor solución obtenida en la primera fase. De esta manera, el

algoritmo se concentra en una región prometedora y es capaz de obtener mejores soluciones con un número menor de secuencias.

En el desarrollo de estos algoritmos hemos introducido una serie de conceptos nuevos, muchos de ellos intrínsecamente relacionados con las secuencias, y hemos descrito algunas de sus propiedades. Estos elementos son importantes por sí mismos y pueden emplearse de diversas formas en algoritmos diferentes a los nuestros. También hemos definido diferentes procedimientos con distintos objetivos, muestreos alrededor de una solución, métodos para combinar secuencias, funciones de mejora, etc., que pueden ser empleados en otros heurísticos. En particular, hemos construido un esquema algorítmico para el RCPSP, MetaRCPSP, capaz de producir heurísticos diversos para este problema con solo especificar y/o modificar algunos elementos.

Uno de los conceptos que más posibilidades ofrece es el de los picos, especialmente si se permite combinar picos de varias soluciones. Esta combinación más general conlleva una problemática que hemos detectado y analizado en el capítulo 6. También hemos desarrollamos una teoría sobre los picos que proporciona maneras sencillas de escoger picos adecuados para la combinación y de construir listas de actividades que los contengan. Además, proponemos varias técnicas y algoritmos que emplean esos procedimientos.

Hemos comparado CARA, HIAC y HGA con los mejores heurísticos de la literatura que conocemos. Según los resultados, CARA es competitivo con los mejores algoritmos y tanto HIAC como HGA los mejoran de forma clara, al menos en j120, el conjunto más difícil de instancias. Tanto HIAC como HGA obtienen una mejor calidad media que cualquier algoritmo de la literatura, empleando para ello 5 (la versión más rápida) y 2 segundos respectivamente. Además, con sus versiones de mayor calidad son capaces de alcanzar cotas de calidad muy superiores a las reportadas en la literatura, con unos tiempos medios no excesivos. En concreto, las desviaciones respecto del CPM (UB) de esas versiones de HIAC y HGA son de 31.58 (0.81)% y 30.95 (0.37)% respectivamente, mientras que la mejor de la literatura es 32.97 (≈1.62)%. Especialmente significativo es el hecho de que HGA lidera con amplitud la lista de algoritmos con una limitación de 5000 secuencias. Este campo es donde se han centrado durante los últimos años los esfuerzos de la gran mayoría de investigadores de heurísticos para el RCPSP.

Un dato global de la calidad de los algoritmos desarrollados en la tesis es que, a fecha del 10 de septiembre del 2001, 289 de las 600 mejores soluciones en j120 habían sido obtenidas por esos algoritmos o alguna de sus versiones. Esta cifra resulta más significativa si se tiene en cuenta que, cuando se comenzó a mandar soluciones, un

número importante de ellas no se podían mejorar porque eran óptimas, como mínimo 156 (su optimalidad se asegura con las cotas inferiores existentes, que presumiblemente no son muy ajustadas en muchos casos, por lo que el número real de óptimos puede ser mayor).

Un aspecto más teórico de la tesis ha sido la definición por primera vez de distancias para el RCPSP, que podrán ser empleadas en el futuro para estudiar el comportamiento de los heurísticos y para su mejora, así como para un mayor conocimiento del problema.

Un aspecto clave de la tesis es la demostración de que la secuenciación hacia atrás y la justificación son técnicas que pueden ser muy importantes en el RCPSP. Por una parte hemos descrito un mecanismo oscilatorio sencillo y eficaz de alternar la búsqueda en las regiones de secuencias activas a la derecha e izquierda (activas para la red original y la inversa). Este mecanismo lo hemos empleado en CARA e HIAC, produciendo excelentes resultados, pero puede ser empleado con cualquier otra función de mejora. Por otra parte, hemos demostrado que la justificación produce mejoras sensibles al añadirla a una variedad amplia de algoritmos. De los resultados obtenidos se puede llegar a afirmar que, hoy por hoy, los heurísticos que no la emplean no son capaces de igualar en calidad a los que sí lo hacen, al menos si se limita el número máximo de secuencias calculadas a 5000.

Dadas las prestaciones de la justificación, hemos profundizado en su estudio teórico, definiendo diferentes extensiones de la justificación (por extremos) y describiendo las relaciones de inclusión entre ellas. También hemos comprobado con un par de ejemplos que puede ser interesante y provechoso seguir investigando en este campo. Además, hemos descrito cómo emplear la justificación en generalizaciones del RCPSP, en particular en el RCPSP con interrupción, el RCPSP con múltiples modos y recursos no renovables y el RCPSP con fechas de entrega en la función objetivo. En el primero hemos realizado pruebas computacionales que demuestran la utilidad de la justificación. Los resultados obtenidos al añadirla nos permiten afirmar que la interrupción es útil en la secuenciación de proyectos con recursos limitados, al menos heurísticamente.

Nuestra opinión es que en años sucesivos se generalizará el uso de la justificación (y de la secuenciación hacia atrás) tanto en el RCPSP como en algunas de sus generalizaciones; ayudar a cimentar las bases para que así sea es una de las aportaciones más importantes de la tesis.

2. LÍNEAS FUTURAS DE INVESTIGACIÓN

Los resultados comentados en el apartado anterior abren un número importante de líneas de investigación para mejorar los algoritmos heurísticos descritos, encontrar nuevos algoritmos para el RCPSP o extender los resultados a otros problemas de secuenciación de proyectos con recursos limitados.

Con respecto a modificaciones de los algoritmos descritos, una posibilidad es combinar los movimientos de CARA y, especialmente, los de HIA con la justificación. Otra posibilidad es estudiar formas rápidas y eficaces de justificación por elegibles y general, intentando superar en calidad – tiempo a la justificación por extremos. Esta justificación se podría emplear en HGA. De igual modo, la forma en que se ha añadido la justificación a los algoritmos (mediante la doble justificación) es muy simple. Parece prometedor estudiar la combinación de la justificación de actividades con otras técnicas, es decir, estudiar otras formas de integración de la justificación en los algoritmos.

Una línea de investigación paralela consiste en programar y comparar los diversos procedimientos descritos en el capítulo 6 relacionados con combinar los picos de varias soluciones. Un comienzo mínimo pero interesante consiste en modificar ligeramente HGA. En cada iteración de HGA, además de las secuencias que se obtienen actualmente, se pueden crear otras soluciones combinando picos de varias soluciones. Esto puede aportar tanto diversidad como soluciones de mayor calidad.

Con respecto a las distancias, puede ser muy provechoso emplearlas para construir heurísticos más efectivos. Un ejemplo de esto es la posibilidad de controlar la diversidad existente en las poblaciones de HGA y actuar en consecuencia. Son conocidos los problemas de los genéticos para mantener la diversidad y que la población continúe evolucionando. Si se detecta poca diversidad en la población, (1) se puede reemplazar parte de la población por nuevas soluciones, (2) durante un cierto número de iteraciones se puede modificar el fitness de las secuencias penalizando el parecido con los elementos de la población o (3) se puede decidir cortar la fase actual y comenzar una nueva con una población obtenida a partir de la mejor solución.

Desde el punto de vista teórico, sería interesante utilizar las distancias para profundizar en el conocimiento del espacio de soluciones posibles para el RCPSP y en cómo se distribuyen las soluciones de calidad. Esto podría servir, por ejemplo, para caracterizar la dificultad de las instancias en función de los parámetros que las

definen, independientemente de los procedimientos exactos o heurísticos que se empleen para su resolución.

Quizás la línea de investigación de resultados más inmediatos sea la de extender algunos de los procedimientos estudiados a otros problemas de secuenciación de proyectos con recursos limitados. Por ejemplo, HGA puede generalizarse fácilmente entre otros al RCPSP con otras funciones objetivo, al RCPSP con interrupción o con múltiple modos y recursos no renovables.

También el esquema algorítmico MetaRCPSP y el mecanismo oscilatorio pueden utilizarse para otros problemas, especificando cada uno de sus elementos de acuerdo con las particularidades del problema. Así como la justificación y la secuenciación hacia atrás, que debe demostrarse que con modificaciones pueden tan útiles en esos problemas como lo son en el RCPSP.

Algunos ejemplos de problemas donde se pueden intentar aplicar estas técnicas son:

- el RCPSP con interrupción
- el RCPSP con múltiples modos y recursos no renovables
- el RCPSP con otras funciones objetivo temporales como la minimización de los tiempos de paso o la minimización de los retrasos si existen fechas de entrega
- el RCPSP con fechas de entrega (en las restricciones)
- el problema de secuenciación de proyectos (con recursos limitados) con restricciones generalizadas
- el problema de maximizar el valor neto actualizado (NPV) cuando existen restricciones de recursos
- el problema de nivelación de recursos
- el problema donde las disponibilidades de los recursos o las demandas de recursos de las actividades varían con el tiempo
- los problemas de empaquetamiento, muy relacionados con los problemas de secuenciación (en algunos casos se pueden modelizar como problemas de secuenciación)

Bibliografía

Alcaraz, J. (2001), Algoritmos Genéticos para Programación de Proyectos con Recursos Limitados, Tesis Doctoral, Universidad Politécnica de Valencia, España.

Alcaraz, J. y Maroto, C. (2001), A Robust Genetic Algorithm for Resource Allocation in Project Scheduling, Annals of Operations Research 102, pp. 83-109.

Alvarez-Valdés, R. y Tamarit, J.M. (1989a), Algoritmos Heurísticos Deterministas y Aleatorios en Secuenciación de Proyectos con Recursos Limitados, Qüestiió 13, pp. 173-191.

Alvarez-Valdés, R. y Tamarit, J.M. (1989b), Heuristic Algorithms for Resource-Constrained Project Scheduling: a Review and an Empirical Analysis, en: R. Slowinski y J. Weglarz (Eds.), *Advances in project scheduling*, Elsevier, Amsterdam, pp. 113-134.

Artigues, C., Michelon, P. y Reusser, S. (2000), Insertion Techniques for Static and Dynamic Resource Constrained Project Scheduling, LIA report 152, Avignon, Francia.

Baar, T., Brucker, P. y Knust, S. (1997), Tabu-Search Algorithms for the Resource-Constrained Project Scheduling Problem, Technical report, Osnabrücker Schriften zur Mathematik, Fachbereich Mathematik/Informatik, Osnabrück.

Baar, T., Brucker, P. y Knust, S. (1998), Tabu Search Algorithms and Lower Bound for the Resource-Constrained Project Scheduling Problem, en: S. Voss, S. Martello, I. Osman y C. Roucairol (Eds.), Meta-Heuristics: *Advances and Trends in Local Search Paradigms for Optimization*, Kluwer Academic Publishers, Boston, pp. 1-18.

Ballestín, F. (1999), Nuevos Procedimientos de Búsqueda en el Espacio de Soluciones del Problema de Secuenciación de Proyectos con Recursos Limitados, Trabajo de investigación, Universidad de Valencia, España.

Bandelloni, M., Tucci, M. y Rinaldi, R. (1994), Optimal Resource Leveling Using Non-Serial Dynamic Programming, European Journal of Operational Research 78, pp. 162-177.

Bell, C. E. y Han, J. (1991), A New Heuristic Solution Method in Resource-Constrained Project Scheduling, Naval Research Logistics 38, pp. 315-331.

Bey, R. B., Doersch, R. H. y Patterson, J. H. (1981), The Net Present Value Criterion: Its Impact on Project Scheduling, Project Management Quaterly 12, pp. 35-45.

Blazewicz, J., Cellary, W., Slowinski, R. y Weglarz, J. (1986), *Scheduling under Resource Constraints – Deterministic Models*, Baltzer, Basel.

Blazewicz, J., Lenstra, J. K. y Rinooy Kan, A. H. G. (1983), Scheduling Subject to Resource Constraints: Classification and Complexity, Discrete Applied Mathematics 5, pp. 11-24.

Boctor, F. F. (1990), Some Efficient Multi-Heuristic Procedures for Resource-Constrained Project Scheduling, European Journal of Operational Research 49, pp. 3-13.

Böttcher, J., Drexl, A., Kolisch, R. y Salewski, F. (1996), Project Scheduling under Partially Renewable Resource Constraints, Technical report, Service de Robotique et Automatisation, Uinversité de Liegè, publicado en Management Science 45, pp. 543-559, (1999).

Bouleimen, K. y Lecocq, H. (1998), A New Efficient Simulated Annealing Algorithm for the Resource-Constrained Project Scheduling Problem, Technical report, Service de Robotique et Automatisation, Université de Liegè, (de próxima aparición en European Journal of Operational Research).

Brinkmann, K. y Neumann, K. (1996), Heuristic Procedures for Resource-Constrained Project Scheduling with Minimal and Maximal Time Lags: the Resource Leveling and the Minimum Project-Duration Problem, Journal of Decision Systems 5, pp. 129-155.

Brucker, P., Drexl, A., Möhring, R., Neumann K. y Pesch, E. (1999), Resource-Constrained Project Scheduling: Notation, Classification, Models, and Methods, European Journal of Operational Research 112, pp. 3-41.

Brucker, P., Knust, S., Schoo, A. y Thiele, O. (1998), A Branch & Bound Algorithm for the Resource-Constrained Project Scheduling Problem, European Journal of Operational Research 107, pp. 272-288

Cho, J. –H. y Kim, Y. –D. (1997), A Simulated Annealing Algorithm for Resource Constrained Project Scheduling Problems, Journal of Operational Research Society 48, pp. 736-744.

Christofides, N., Alvarez-Valdés, R. y Tamarit, J. M. (1987), Project Scheduling with Resource Constraints: a Branch and Bound Approach, European Journal of Operational Research 29, pp. 262-273.

Cooper, D.F. (1976), Heuristics for Scheduling Resource-Constrained Projects: an Experimental Investigation, Management Science 22, pp. 1186-1194.

Davis, E. W. (1966), Resource Allocation in Project Network Models – a Survey, The Journal of Industrial Engineering 17, pp. 177-188.

Davis, E. W. (1973), Project Scheduling under Resource Constraints – Historical Review and Categorization of Procedures, AIIE Transactions 5, pp. 297-313.

Davis, E. W. y Patterson, J. H. (1975), A Comparison of Heuristic and Optimum Solutions in Resource-Constrained Project Scheduling, Management Science 21, pp. 944-955.

Deckro, R. F. y Hebert, J. E. (1989), Resource Constrained Project Crashing, OMEGA International Journal of Management Science 17, pp. 69-79.

Deckro, R. F. y Hebert, J. E. (1990), A Multiple Objective Programming Framework for Tradeoffs in Project Scheduling, Engineering Costs and Production Economics 18, pp. 255-264.

Demeulemeester, E. (1992), 'Optimal Algorithms for Various Classes of Multiple Resource-Constrained Project Scheduling Problems', PhD Dissertation, Katholieke Universiteit Leuven, Belgium.

Demeulemeester, E. (1995), Minimizing Resource Availability Costs in Time-Limited Project Networks, Management Science, 41, pp. 1590-1598.

Demeulemeester, E. y Herroelen, W. (1992), A Branch-and-Bound Procedure for the Multiple Resource-Constrained Project Scheduling Problem, Management Science 38, pp. 1803-1818.

Demeulemeester, E. y Herroelen, W. (1997), New Benchmark Results for the Resource-Constrained Project Scheduling Problem, Management Science 43, pp. 1485-1492.

Demeulemeester, E., Herroelen, W. y Elmaghraby, S. E. (1996), Optimal Procedures for the Discrete Time/Cost Trade-Off Problem in Project Networks, European Journal of Operational Research 88, pp. 50-68.

Dorndorf, U., Pesch, E. y Phan-Huy, T. (2000), A Branch and Bound for the Resource Constrained Project Scheduling Problem, Mathematical Methods in OR 52, pp. 413-439.

Elmaghraby, S. E. (1977), *Activity Networks: Project Planning and Control by Network Models*, Wiley, Nueva York.

Elmaghraby, S. E. (1995), Activity Nets: a Guided Tour through Some Recent Developments, European Journal of Operational Research 82, pp. 383-408.

Eglese, R. W. (1990), Simulated Annealing: A Tool for Operational Research, European Journal of Operational Research 46, pp. 271-281.

Glover, F. (1989a), Tabu Search – Part I, ORSA Journal on Computing 1, pp. 190-206.

Glover, F. (1989b), Tabu Search – Part II, ORSA Journal on Computing 2, pp. 4-32.

Glover, F. (1998), A Template for Scatter Search and Path Relinking, en: J. –K. Hao, E. Lutton, E. Ronald, M. Schoenauer y D. Snyers (Eds.), *Artificial Evolution*, Lecture Notes in Computer Science 1363, Springer-Verlag, Berlin, pp. 13-54.

Hartmann, S. (1997), A Competitive Genetic Algorithm for Resource-Constrained Project Scheduling, Technical report 451, Manuskripte aus den Instituten für Betriebswirtschaftslehre der Universität Kiel.

Hartmann, S. (1998), A Competitive Genetic Algorithm for Resource-Constrained Project Scheduling, Naval Research Logistics 45, pp. 733-750.

Hartmann, S. (1999), *Project Scheduling under Limited Resources: Models, Methods, and Applications*, número 478 en Lecture Notes in Economics and Mathematical Systems. Springer, Berlin, Alemania.

Hartmann, S. (2000), A Self-Adapting Genetic Algorithm for Project Scheduling under Resource Constraints, Workpaper.

Hartmann, S. (2001), Project Scheduling with Multiple Modes: a Genetic Algorithm, Annals of Operations Research 102, pp. 111-135.

Hartmann, S. y Kolisch, R. (2000), Experimental Evaluation of State-of-the-Art Heuristics for the Resource-Constrained Project Scheduling Problem, European Journal of Operational Research 127, pp. 394-407.

Herroelen, W. (1972), Resource-Constrained Project Scheduling – the State of the Art, Operational Research Quaterly, 23, 261-275.

Herroelen, W., Demeulemeester, E. y De Reyck, B. (1998), Resource-Constrained Project Scheduling: a Survey of Recent Developments, Computers and Operations Research 25, pp. 279-302.

Herroelen, W., Demeulemeester, E. y De Reyck, B. (1999), A Classification Scheme for Project Scheduling, en: J. Weglarz (Ed.), *Project Scheduling - Recent Models, Algorithms and Applications*, Kluwer Academic Publishers, Boston, pp. 1-26.

Holland, H. J. (1975), *Adaptation in Natural and Artificial Systems*, University of Michigan Press, Ann Arbor.

Icmeli, O. y Rom, W. (1997), Ensuring Quality in Resource Constrained Project Scheduling, European Journal of Operational Research 103, pp. 483-496.

Icmeli, O. y Rom, W. (1998), Analysis of the Characteristics of Projects in Diverse Industries, Journal of Operations Management 16, pp. 43-61.

Kelley, J. E., Jr. (1961), Critical–Path Planning and Scheduling. Mathematical Basis, Operations Research 9, pp. 296-320.

Kirkpatrick, S., Gelatt, C. D. y Vecchi, M. P. (1983), Optimization by Simulated Annealing, Science 220, pp. 671-680.

Klein, R. (2000), *Scheduling of Resource-Constrained Projects*, Kluwer Academic Publishers, Boston.

Klein, R. y Scholl, A. (1998), Scattered Branch and Bound – an Adaptive Search Strategy Applied to Resource-Constrained Project Scheduling, Schriften zur Quantitativen Betriebswirtschaftslehre 6/98, Technische Universität Darmstadt, Alemania.

Kohlmorgen, U., Scmeck, H. y Haase, K. (1999), Experiences with Fine-Grained Parallel Genetic Algorithms, Annals of Operations Research 90, pp. 203-219.

Kolisch, R. (1995), *Project Scheduling under Resource Constraints – Efficient Heuristics for Several Problem Classes*, Physica, Heidelberg.

Kolisch, R. (1996a), Efficient Priority Rules for the Resource-Constrained Project Scheduling Problem, Journal of Operations Management 14, pp. 179-192.

Kolisch, R. (1996b), Serial and Parallel Resource-Constrained Project Scheduling Methods Revisited: Theory and Computation, European Journal of Operational Research 90, pp. 320-333.

R. Kolisch y A. Drexl, Adaptive Search for Solving Hard Project Scheduling Problems, Naval Research Logistics 43 (1996) 23-40.

Kolisch, R. y Hartmann, S. (1999), Heuristic Algorithms for the Resource-Constrained Project Scheduling Problem: Classification and computational analysis, en: J. Weglarz (Ed.), *Project Scheduling - Recent Models, Algorithms and Applications*, Kluwer Academic Publishers, Boston, pp. 147-178.

Kolisch, R. y Padman, R. (2000), An Integrated Survey of Deterministic Project Scheduling, Omega 29 249-272.

Kolisch, R., Schwindt, C. y Sprecher, A. (1999), Benchmark Instances for Project Scheduling Problems, en: J. Weglarz (Ed.), Project Scheduling - Recent Models, Algorithms and Applications, Kluwer Academc Publishers, Boston, pp. 197-212.

Kolisch, R. y Sprecher, A. (1996), PSPLIB – a Project Scheduling Problem Library, European Journal of Operational Research, 96, pp. 205-216.

Kolisch, R., Sprecher, A. y Drexl, A. (1992), Characterization and Generation of a General Class of Resource-Constrained Project Scheduling Problems: Easy and Hard Instances, Technical report 301, Manuskripte aus den Instituten für Betriebswirtschaftslehre der Universität Kiel.

Kolisch, R., Sprecher, A. y Drexl, A. (1995), Characterization and Generation of a General Class of Resource-Constrained Project Scheduling Problems, Management Science 41, pp. 1693-1703.

Krämer, A. (1995), Scheduling Multiprocessor Tasks on Dedicated Processors, PhD Thesis, Department of Mathematics/Informatics, Universität Osnabrück.

Lawrence, S. R. (1985), Resource Constrained Project Scheduling – a Computational Comparison of Heuristic Scheduling Techniques, Technical report, Graduate School of Industrial Administration, Carnegie-Mellon University, Pittsburgh.

Lee, J. –K. y Kim, Y. –D. (1996), Search Heuristics for Resource Constrained Project Scheduling, Journal of the Operational Research Society 47, pp. 678-689.

Leon, V. J. y Ramamoorthy, B. (1995), Strength and Adaptability of Problem-Space Based Neighbourhoods for Resource-Constrained Scheduling, OR Spektrum 17, pp. 173-182.

Li, R. K. –Y. y Willis, J. (1992), An Iterative Scheduling Technique for Resource-Constrained Project Scheduling, European Journal of Operational Research 56, pp. 370-379.

Lino, P. (1997), Planificación de Proyectos en Diagramas de Precedencias, Tesis Doctoral, Universidad de Valencia, España.

Lova, A., Maroto, C. y Tormos, P. (2000), A Multicriteria Heuristic Method to Improve Resource Allocation in Multiproject Scheduling, European Journal of Operational Research 127, pp. 408-424.

Malcolm, D. G., Roseboom, J. H., Clark, C. E. y Fazar, W. (1959), Applications of a Technique for Research and Development Program Evaluation, Operations Research 7, pp. 646-669.

Martello, S. y Toth, P. (1990), Knapsack Problems: Algorithms and Computer Implementations, John Wiley & Sons.

Martin, O., Otto, S. W. y Felten, E. W. (1992), Large-Step Markov Chains for TSP Incorporating Local Search Heuristics, Operations Research Letters 11, pp. 219-224.

Mausser, H. E. y Lawrence, S. R. (1995), Exploiting Block Structure to Improve Resource-Constrained Project Schedules, en: F. Glover, I. Osman y J. Kelley (Eds.), *Metaheuristics 1995:State of the Art*, Kluwer, Maryland, pp. 203-218.

Merkle, D., Middendorf, M. y Schmeck, H. (1999), Ant Colony Optimization for Resource-Constrained Project Scheduling, Workpaper Institute for Applied Computer Science and Formal Description Methods, Universität of Karlsruhe, Karlsruhe, Alemania.

Merkle, D., Middendorf, M. y Schmeck, H. (2001), Ant Colony Optimization Techniques for the Resource-Constrained Project Scheduling Problem, Workpaper Institute for Applied Computer Science and Formal Description Methods, Universität of Karlsruhe, Karlsruhe, Alemania.

Mingozzi, A., Maniezzo, V., Ricciardelli, S. y Bianco, L. (1998), An Exact Algorithm for Project Scheduling with Resource Constraints Based on a New Mathematical Formulation, Management Science 44, pp. 714-729.

Moder, J. J., Phillips, C. R. y Davis, E. W. (1983), *Project Management with CPM, PERT and Precedence Diagramming*, Van Nostrand Reinhold, Nueva York.

Möhring, R. H. (1984), Minimizing Costs of Resource Requirements in Project Networks subject to a Fixed Completion Time, Operations Research 32, pp. 89-120.

Möhring, R. H., Schulz, A., Stork, F. y Uetz, M. (2000), Solving Project Scheduling Problems by Minimum Cut Computations, Workpaper.

Naphade, K. S., Wu, S. D. y Storer, R. H. (1997), Problem Space Search Algorithms for Resource-Constrained Project Scheduling, Annals of Operations Research 70, pp. 307-326.

Neumann, K. y Zimmermann, J. (1999), Methods for Resource-Constrained Project Scheduling with Regular and Nonregular Objective Functions Schedule-Dependent Time Windows, en: J. Weglarz (Ed.), *Project Scheduling - Recent Models, Algorithms and Applications*, Kluwer Academic Publishers, Boston.

Nonobe, K. e Ibaraki, T. (1999), Formulation and Tabu Search Algorithm for the Resource Constrained Project Scheduling Problem (RCPSP), Technical report 99010, de próxima aparición en Essays and Surveys in Metahuristics (MIC'99).

Oguz O. y Bala, H. (1994), A Comparative Study of Computational Procedures for the Resource Constrained Project Scheduling Problem, European Journal of Operational Research 72, pp. 406-416.

Özdamar, L. (1999), A Genetic Algorithm Approach to a General Category Project Scheduling Problem, IEEE Transactions on Systems, Man, and Cybernetics, Part C: Applications and Reviews 29, pp. 44-59.

Özdamar, L. y Ulusoy, G. (1995), A Survey on the Resource-Constrained Project Scheduling Problem, AIIE Transactions 27, pp. 574-586.

Özdamar, L. y Ulusoy, G. (1996a), An Iterative Local Constraint Based Analysis for Solving the Resource Constrained Project Scheduling Problem, Journal of Operations Management, 14, pp. 193-208.

Özdamar, L. y Ulusoy, G. (1996b), A Note on an Iterative Forward/Backward Scheduling Technique with Reference to a Procedure by Li and Willis, European Journal of Operational Research 89, pp. 400-407.

Patterson, J. H. (1976), Project Scheduling: The Effects of Problem Structure on Heuristic Performance, Naval Research Logistics Quaterly 23, pp. 95-123.

Patterson, J. H. (1984), A Comparison of Exact Approaches for Solving the Multiple Constrained Resource, Project Scheduling Problem, Management Science 30, pp. 854-867.

Patterson, J. H. y Roth, G. W. (1976), Scheduling a Project under Multiple Resource Constraints: a Zero-One Programming Approach, AIIE Transactions 8, pp. 449-455.

Pinson, E., Prins, C. y Rullier, F. (1994), Using Tabu Search for Solving the Resource-Constrained Project Scheduling Problem, *Proceedings of the Fourth International Workshop on Project Management and Scheduling*, Leuven, pp. 102-106.

Pollack-Johnson, B. (1995), Hybrid Structures and Improving Forecasting and Scheduling in Project Management, Journal of Operations Management 12, pp. 101-117.

Pritsker, A. A. B., Watters, L. J. y Wolfe, P. M. (1969), Multiproject Scheduling with Limited Resources: a Zero-One Programming Approach, Management Science 16, pp. 93-107.

Pritsker, A. A. B. y Happ, W. W. (1966), GERT: Graphical Evaluation and Review Techique – Part I: Fundamentals, Journal of Industrial Engineering 17, pp. 267-274.

Sampson, S. E. y Weiss, E. N. (1993), Local Search Techniques for the Generalized Resource Constrained Project Scheduling Problem, Naval Research Logistics 40, pp. 665-675.

Schirmer, A. y Riesenberg, S. (1997), Parameterized Heuristics for Project Scheduling – Biased Random Sampling Methods, Technical report 456, Manuskripte aus den Instituten für Betriebswirtschaftslehre der Universität Kiel.

Schirmer, A. y Riesenberg, S. (1998), Case-Based Reasoning and Parameterized Random Sampling for Project Scheduling, Technical report 472, Manuskripte aus den Instituten für Betriebswirtschaftslehre der Universität Kiel.

Slowinski, R. (1989), Multiobjective Project Scheduling under Multiple-Category Resource Constraints, en R. Slowinski y J. Weglarz (Eds.), *Advances in Project Scheduling*, Elsevier, Amsterdam, pp. 151-167.

Slowinski, R., Soniewicki, B. y Weglarz. J. (1994), DSS for Multiobjective Project Scheduling, European Journal of Operational Research 79, pp. 220-229.

Shaffer, L. R., Ritter, J. B. y Meyer, W. L. (1965), *The Critical-Path Method*, McGraw Hill, Nueva York.

Smith-Daniels, D. E. y Aquilano, N. J. (1987), Using a Late Start Resource Constrained Project Schedule to Improve Project Net Present Value, Decision Sciences 18, pp. 617-630.

Speranza, M. G. y Vercellis, C. (1993), Hierarchical Models for Multi-Project Planning and Scheduling, European Journal of Operational Research 64, pp. 312-325.

Sprecher, A. (1994), *Resource-Constrained Project Scheduling: Exact Methods for the Multi-Mode Case*, número 409 en Lecture Notes in Economics and Mathematical System. Springer, Berlin, Alemania.

Sprecher, A. (1996), Solving the RCPSP Efficiently at Modest Memory Requirements, Technical report 425, Manuskripte aus den Instituten für Betriebswirtschaftslehre der Universität Kiel.

Sprecher, A. y Drexl, A. (1998), Multi-Mode Resource-Constrained Project Scheduling by a Simple, General and Powerful Sequencing Algorithm, European Journal of Operational Research 107, pp. 431-450.

Sprecher, A., Hartmann, S. y Drexl, A. (1997), An Exact Algorithm for Project Scheduling with Multiple Modes, OR Spektrum 19, pp. 195-203.

Sprecher, A., Kolisch, R. y Drexl, A. (1995), Semi-Active, Active, and Non-Delay Schedules for the Resource-Constrained Project Scheduling Problem, European Journal of Operational Research 80, 94-102.

Talbot, F. B. (1982), Resource-Constrained Project Scheduling with Time-Resource Tradeoffs: the Nonpreemptive Case, Management Science 28, pp. 1197-1210.

Thomas, P. R. y Salhi, S. (1997), An Investigation into the Relationship of Heuristic Performance with Network-Resource Characteristics, Journal of the Operational Research Society, 48, pp. 34-43.

Thomas, P. R. y Salhi, S. (1998), A Tabu Search Approach for the Resource Constrained Project Scheduling Problem, Journal of Heuristics 4, pp. 123-139.

Tormos, P. y Lova, A. (2001), A Competitive Heuristic Solution Technique for Resource-Constrained Project Scheduling, Annals of Operations Research 102, pp. 65-81.

Ulusoy, G. y Özdamar, L. (1989), Heuristic Performance and Network/Resource Characteristics in Resource-Constrained Project Scheduling, Journal of the Operational Research Society 40, pp. 1145-1152.

Ulusoy, G. y Özdamar, L. (1994), A Constraint-Based Perspective in Resource Constrained Project Scheduling, International Journal of Production Research 32, pp. 693-705.

Valls, V., Pérez, A. y Quintanilla, S. (1998), Pre-Processing Techniques for Resource Allocation in the Heterogeneous Case, European Journal of Operational Research 107, pp. 470-491.