UNIVERSIDAD DE BUENOS AIRES
Facultad de Ciencias Exactas y Naturales
Departamento de Matemática

Tesis de Licenciatura

Algoritmos para el problema de Job-Shop Scheduling basados en Programación Dinámica

Agustín Alejandro Nogueira

Director: Ignacio Ojea.

28 de Agosto de 2015

Agradecimientos

A Ignacio:

Por dirigir esta tesis y trabajar junto conmigo para su realización. Por el esfuerzo y excesiva cantidad de tiempo dedicados a superar las dificultades, y por siempre impulsarme a seguir adelante.

A Jelke van Hoorn:

Por la amabilidad y predisposición al intercambio de ideas, y por el tiempo dedicado a resolver problemas de este trabajo.

A mi vieja:

Por dar todo por mí, y por siempre cuidarme y amarme solo como una madre sabe hacerlo.

A mi viejo:

Por ayudarme y enseñarme los valores de la vida. Por el apoyo incondicional, y por siempre guiarme para ser una mejor persona.

A mi abuela:

Por ser mi segunda madre y abuela al mismo tiempo. Por todo lo que hiciste por mí y todo lo que aprendí de vos.

A Samanta:

Por acompañarme hace ya ocho años en este viaje. Por el amor que me das todos los días, y la paciencia que me tenés. Definitivamente este trabajo no podría haberse realizado sin vos.

A Herman:

Porque sos como un hermano. Por las anécdotas inolvidables y por todos los momentos compartidos. Y, obviamente, por ayudarme en el código y en detalles técnicos de este trabajo.

A Ramón:

Porque sos como otro hermano. Porque, a pesar de las continuas lesiones y momentos bochornosos que me haces pasar, hablamos el mismo idioma.

A Luciano y Gabriel:

Por ser mis amigos desde que tengo memoria. Por los lindos momentos que pasamos juntos, y todas las noches de vicio que compartimos.

A Adrián, Roberto, Alejandro y Lucio:
Por los grandes momentos vividos dentro de la facultad y fuera de ella. Por el tiempo, las cervezas y las risas compartidas.

A Gisela (*La Enana*):
Por siempre estar predispuesta a ayudarme y prestarme tus apuntes, y, principalmente, por ser una gran amiga y persona (exceptuando la vez que me abandonaste en la parada del 160).

A Noelia y Paula:
Por los momentos compartidos y las experiencias vividas dentro y fuera de la facultad, y por soportar todas mis bromas.

A Santiago:
Por todos los momentos que compartidos en el laburo y fuera de él, y (tengo que admitirlo) por todas las veces que me acercaste a casa a altas horas de la madrugada.

A Luciana:
Porque me soportas diariamente en el laburo, y porque sé que siempre tenes un oído disponible para lo que sea.

A Mariu (*La Rubia*):
Porque, más alla de las distancias, fuiste y sos una gran amiga.

A mis amigos:
Alán, Alejandra, Gisela, Gabriela, Hernán, Estefanía, Mariela B., Mariela, Irina, Lucas, Javi.

Índice

1. Introducción y nociones básicas 1
- 1.1. Problemas de optimización y optimización combinatoria 1
  - 1.1.1. Problemas de optimización e instancias del problema . 1
  - 1.1.2. Problemas de optimización combinatorios 2
  - 1.1.3. Grafos 3
  - 1.1.4. Cotas 5
- 1.2. Algoritmos 6
  - 1.2.1. Definiciones y nociones básicas 6
  - 1.2.2. Complejidad algorítmica 6
  - 1.2.3. Teoría NP 8
  - 1.2.4. Algoritmos exactos y heurísticas 12

2. Programación Dinámica 15
- 2.1. Descripción del método 15
  - 2.1.1. Procesos de Decision en Múltiples Pasos 15
  - 2.1.2. El principio de optimalidad 15
  - 2.1.3. El método general 19
  - 2.1.4. Ejemplo práctico: el problema de la mochila 22
- 2.2. La programación dinámica en problemas de secuenciamiento . 25
  - 2.2.1. Definiciones y nociones básicas 25
  - 2.2.2. Ejemplo práctico: un problema de scheduling 27
- 2.3. Limitaciones de la programación dinámica 29
- 2.4. Conclusiones y comentarios finales 31

3. Problema del Job-shop scheduling 33
- 3.1. Descripción del problema 33
- 3.2. Definiciones y nociones básicas 34
- 3.3. Schedules 35
  - 3.3.1. Definición 35
  - 3.3.2. Representación de las soluciones 37
  - 3.3.3. Orden entre operaciones 38
  - 3.3.4. Schedules activos 43
- 3.4. Complejidad del problema 45

4. Aplicación de programación dinámica al JSSP 47
- 4.1. Preliminares 47
- 4.2. El JSSP como problema de secuenciamiento 47
  - 4.2.1. Representación del problema mediante secuencias . . . 48
  - 4.2.2. Expansión de secuencias y generación de soluciones . . 54

4.3. El principio de optimalidad para el JSSP 61
4.3.1. Analogía con el TSP en un primer acercamiento 61
4.3.2. Secuencias comparables y dominadas 65
4.4. Dominancia directa, indirecta y cadenas de dominancia 77
4.4.1. Construcción de la solución óptima 82
4.5. Secuencias con operaciones retrasadas 87
4.5.1. Eliminación de secuencias parciales con operaciones retrasadas 88
4.5.2. Cotas inferiores y superiores 95

5. Implementación 109
5.1. Construcción del algoritmo 109
5.2. Complejidad del algoritmo 114
5.2.1. Supuestos previos 114
5.2.2. Parámetros y variables 115
5.2.3. Análisis de complejidad 116

6. Heurística para el JSSP 121
6.1. Introducción 121
6.2. Reglas de prioridad sobre operaciones 122
6.3. Beam Search 123
6.4. Implementación heurística para el JSSP 124
6.4.1. Método general 124
6.4.2. Construcción del algoritmo 125

7. Resultados y análisis 129
7.1. Algoritmo Exacto 129
7.2. Heurística 133

8. Conclusiones y comentarios finales 139

A. Contraejemplo 141

Referencias 145

Resumen

El objetivo principal de esta tesis es estudiar un algoritmo exacto para el problema de job-shop scheduling, basado en programación dinámica, que fue propuesto en [6]. El algoritmo es de complejidad exponencial, lo cual resulta natural dado que el problema de job-shop scheduling es NP-Hard [3]. Sin embargo, es también exponencialmente mejor que la fuerza bruta. De este modo, representa un pequeño avance en un sentido poco frecuente, dado que la mayor parte de la literatura dedicada a la resolución de problemas NP-Hard apunta al desarrollo de nuevas heurísticas.

El algoritmo construye soluciones iterativamente y por etapas. En cada etapa se cuenta con un conjunto de soluciones parciales cada una de las cuales es expandida en la etapa siguiente a través del agregado de una nueva operación. La naturaleza exponencial del algoritmo radica en el crecimiento exponencial de estos conjuntos. Al seguir la línea de un planteo por programación dinámica, la formulación del problema propuesta en [6] permite comparar soluciones parciales con el objetivo de descartar algunas de ellas. Esto contribuye a mejorar la eficiencia práctica del algoritmo. Nuestra implementación mejora la original, dado que incorpora nuevos criterios de comparación que permiten reducir considerable el número de soluciones parciales construidas por el algoritmo.

Al tratarse de un algoritmo exacto, éste permite resolver en tiempos aceptables instancias relativamente pequeñas del problema, pero no instancias medianas o grandes. Para estos casos, implementamos una variante heurística que consiste esencialmente en acotar arbitrariamente el número de soluciones parciales generadas en cada etapa. Si bien esta heurística permite encontrar soluciones cercanas al óptimo en instancias que resultaban inabordables con el algoritmo exacto, su eficiencia dista de la de otras técnicas heurísticas estudiadas en la literatura.

El primer capítulo está dedicado a la introducción de nociones básicas de problemas de optimización combinatoria, mientras que el segundo presenta los rudimentos de la programación dinámica. En el Capítulo 3 se plantea el problema de job-shop scheduling. El cuarto capítulo representa el corazón de la tesis: allí se realiza la formulación del problema, siguiendo la línea de [6], se presenta el esquema del algoritmo y se demuestra que efectivamente encuentra una solución óptima. La demostración de este hecho dada en el artículo original contenía errores que aquí son salvados. En el Apéndice se muestra, con un contraejemplo, la falsedad de alguna de las afirmaciones de [6]. En el Capítulo 5 se detalla nuestra implementación y se estima su complejidad. La técnica heurística es presentada en el sexto capítulo, mientras que el séptimo está dedicado a la exposición de los resultados obtenidos.

1. Introducción y nociones básicas

Introducción

En este capítulo introductorio explicaremos conceptos fundamentales y nociones básicas sobre problemas de optimización. Estas ideas nos servirán en los futuros capítulos para poder atacar el problema principal de este trabajo desde un punto de vista estructurado, lo que facilitará la exposición y comprensión.

Dado que la intención no es agobiar al lector con infinidad de definiciones formales, la explicación de los conceptos en esta introducción se hará de forma breve, concisa e informal. Por otra parte, entendemos que una explicación exhaustiva de estos conceptos excede largamente el ámbito de esta tesis, por lo cual nos limitaremos a exhibir las nociones mínimas e indispensables.

1.1. Problemas de optimización y optimización combinatoria

1.1.1. Problemas de optimización e instancias del problema

Empezaremos definiendo qué es una instancia del problema:

Definición 1.1. *Una* instancia *de un problema de optimización será un par (S, f) donde:*

- *S es un conjunto de puntos.*
- *$f : S \to \mathbb{R}$ una función que se desea minimizar.*

De este modo, resolver la instancia (S, f) de un cierto problema consistirá en encontrar $s_{opt} \in S$ tal que

$$f(s_{opt}) = \min_{s \in S}\{f(s)\}$$

El el conjunto de soluciones S se denomina conjunto de soluciones *factibles* y será el espacio de búsqueda de la solución óptima s_{opt}. Las solución factibles serán aquellas que cumplan una serie de reglas propias o *restricciones* del problema.

A la función $f : S \to \mathbb{R}$ la llamaremos *funcional* o *función de costo.* f representar cuantitativamente la cantidad a optimizar.

Cabe destacar que cualquier problema de maximización cuyo funcional sea $f(s)$, puede ser visto como un problema de minimización definiendo un

nuevo funcional $g(s) = -f(s)$. En este sentido, la representación en la Definición 1.1 abarca ambos casos.

Observemos que cualquier instancia puede ser vista como un caso particular de un problema de optimización. Así, podremos definir un problema de optimización de la siguiente forma:

Definición 1.2. *Un problema de optimización será el conjunto $\mathcal{I}$ de todas las instancias asociadas a dicho problema. Por lo tanto, consideraremos que un determinado método* resuelve *un problema si es capaz de resolver cualquier instancia $i \in \mathcal{I}$ dada.*

1.1.2. Problemas de optimización combinatorios

Básicamente, los problemas de optimización combinatorios son problemas de optimización en donde el conjunto de soluciones factibles es finito. Más formalmente:

Definición 1.3. *Una* instancia *de un problema de optimización combinatoria será un par (S, f) tal que:*

- $S = \{s_1, \cdots, s_n\}$ *es el conjunto de puntos factibles.*
- $f : S \to \mathbb{R}$ *una función que servirá para evaluar cada punto factible.*

Resolver esta instancia del problema consiste en encontrar $s_{opt} \in S$ tal que

$$f(s_{opt}) = \min_{i=1,\cdots,n} \{f(s_i)\}$$

Proposicion 1.1. *Cualquier instancia (S, f) de un problema de optimización combinatoria, con $S \neq \emptyset$, tiene, al menos, una solución óptima.*

Demostración. Como $S \neq \emptyset$, $\#(S) = n$ es finito y $f : S \to R$, entonces existe un único mínimo del conjunto $\{f(s) : s \in S\}$. □

La anterior proposición pone de manifiesto la principal característica de los problemas de optimización combinatoria: admiten solución óptima. Cabe resaltar que en la Proposición 1.1, la unicidad se corresponde con el valor mínimo de las soluciones factibles, pero no con la cantidad de soluciones óptimas. De hecho, en muchos problemas existe más de una solución óptima.

1.1.3. Grafos

Como el conjunto de soluciones factibles es finito, muchos problemas de optimización combinatoria pueden ser representados gráficamente utilizando *grafos finitos.*

Definición 1.4. *Un grafo dirigido G se define como un par (V, E), donde V es un conjunto cuyos elementos son denominados vértices ó nodos, y E es un conjunto de pares no ordenados de vértices que reciben el nombre de aristas ó arcos.*

Si $V = \{v_1, \cdots, v_n\}$, los elementos de E se representan de la forma (v_i, v_j), donde $i \neq j$. Además, diremos que v_i y v_j son adyacentes si $(v_i, v_j) \in E$.

Por último, G será ponderado *si para cada arista $(v_i, v_j) \in E$ existe una función $f : E \to \mathbb{R}$ que le asigne un número real, que llamaremos* costo *o* peso *de la arista.*

Haremos uso de las siguientes definiciones:

Definición 1.5. *Dado $G = (V, E)$ un grafo,*

1. *Un* camino *en G será una sucesión de nodos $v_1, v_2, \cdots, v_k$, tales que v_i y v_{i+1} son adyacentes para $i = 1, \cdots, k-1$. Notaremos este camino como $v_1 \to v_2 \to \cdots \to v_k$.*
2. *Un* ciclo *en G será una sucesión de nodos que comienza y termina en el mismo nodo: $v_1, v_2, \cdots, v_k, v_1$, tal que $(v_i, v_i + 1) \in E$ para $i = 1, \cdots, k-1$ y $(v_k, v_1) \in E$.*
3. *Un grafo $G = (V, E)$ se dirá* conexo *si $\forall v, w \in V : \exists$ un camino entre v y w.*
4. *Un grafo $G = (V, E)$ será un* arbol *si es conexo y no contiene ciclos. Los nodos $v \in V$ en los que incida una única arista serán las* hojas *del arbol.*

Para ilustrar cómo un problema de optimización combinatoria puede ser representado mediante un grafo, expondremos el siguiente ejemplo.

Problema 1. Camino mínimo

Dado un grafo dirigido, acíclico y ponderado $G = (V, E, f)$, un nodo inicial v y un nodo final w, queremos hallar el camino de mínimo costo *entre v y w, donde el costo de un camino $u_0 \to u_1 \to u_2 \to \cdots \to u_n$ se define como la suma de los pesos de sus aristas:*

$$f(u_0 \to u_1 \to u_2 \to \cdots \to u_n) = \sum_{i=0,\cdots,n-1} f((u_i, u_{i+1}))$$

En la Figura 1 podemos ver una instancia particular de este problema, compuesta de 9 nodos y 12 aristas

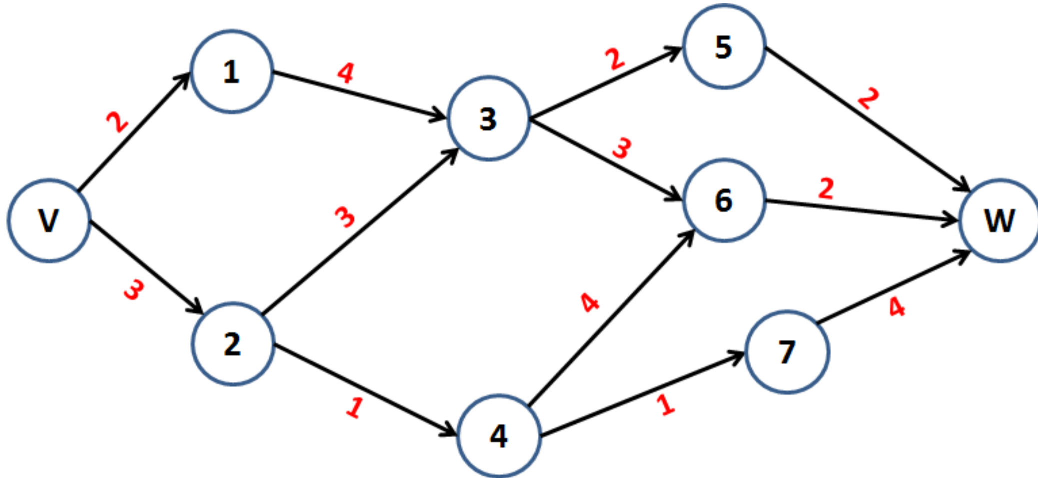


Figura 1: Instancia de un problema de camino mínimo

En esta instancia, una posible solución podría ser la exploración de todos los caminos que salen de v y llegan a w. Este procedimiento, conocido en la literatura como *búsqueda exhaustiva* o *fuerza bruta*, genera el grafo de la figura 2 para la instancia anterior, al que llamaremos *árbol de búsqueda*.

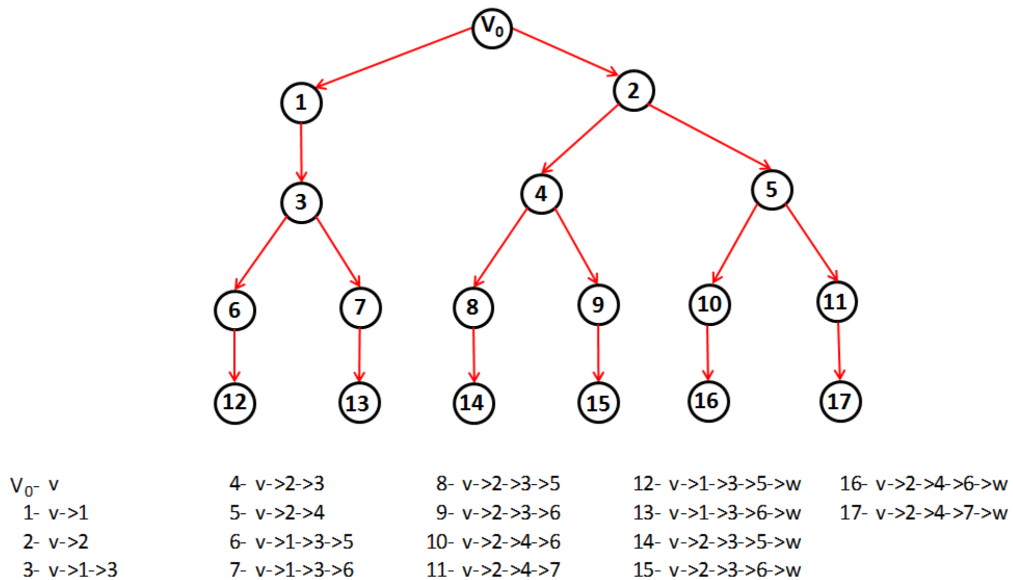


Figura 2: Árbol de búsqueda correspondiente a la instancia de la Figura 1

Podemos identificar los siguientes elementos en el árbol de búsqueda:

- Un nodo raíz v_0 que representará el estado inicial del problema.
- Los vértices de la última etapa u *hojas* del árbol. Cada hoja representa una solución factible diferente.
- Cada nodo salvo el nodo raíz y las hojas representan estados intermedios entre el estado inicial y las soluciones factibles. Estos estados intermedios serán llamados soluciones parciales, dado que constituyen potenciales soluciones factibles que aún se encuentran *incompletas*.
- Una serie de caminos que llevan desde el nodo raíz a una solución factible (hoja), que denominaremos *ramas*. Así, cada solución parcial se *ramificará* en más soluciones parciales dependiendo de las restricciones del problema.
- Cada solución parcial se encuentra a una determinada cantidad de vértices del estado inicial (o de los estados finales). Esta cantidad será denominada como *nivel* o *etapa*. Así, la k-ésima etapa tendrá nodos que estén a k nodos del nodo raíz (o de las hojas).

En determinadas ocasiones es posible descartar una determinada ramificación dentro del árbol de búsqueda. Por ejemplo: si al realizar el recorrido exhaustivo del árbol del búsqueda del ejemplo se ha considerado la solución $v \to 2 \to 4 \to 7 \to w$, cuyo costo es 9, puede descartarse la solución parcial $v \to 2 \to 3 \to 6$ cuyo costo también es 9, pero deberá inevitablemente aumentar para alcanzar el nodo w. Esta acción de descarte de soluciones parciales se denomina *poda*.

Como los problemas de optimización combinatoria tienen finitas soluciones factibles, en general, las ramificaciones serán también finitas, permitiéndonos representar gráficamente la forma en que resolveremos los problemas mediante árboles de búsqueda. Esta forma de representación es útil no sólo porque facilita el entendimiento, sino también porque nos permite aplicar técnicas conocidas o desarrollar métodos alternativos dependiendo de la estructura propia del problema.

1.1.4. Cotas

Para cualquier problema de optimización podemos definir genéricamente dos tipos de cotas:

Definición 1.6.
- *Cotas inferiores de soluciones parciales: $CI(x)$ es una cota inferior de la solución parcial x si $\forall s \in S$ tal que $\exists$ un camino desde x hasta s se cumple que $f(s) \geq CI(x)$.*

- *Cotas superiores: CS es una cota superior del problema si $f(s_{opt}) \leq CS$.*

Observemos que de la definición anterior se pueden concluir los siguientes resultados:

1. Toda solución factible s es cota superior del problema, pues $f(s) \geq \text{mín}_{s \in S}\{f(s)\} = f(s_{opt})$.

2. Si $CS = f(s)$ y $CI(x) \geq CS$, entonces $\forall s'$ en alguna rama de x: $f(s) \leq f(s')$. Luego, s será una *mejor* solución que todas las soluciones factibles en ramas de x, por lo que podremos podar dichas ramas.

1.2. Algoritmos

1.2.1. Definiciones y nociones básicas

En general, para poder resolver problemas no triviales se debe desarrollar una metodología que pueda computar la solución a dicho problema. Para ello, se implementa en una computadora lo que se conoce como *algoritmo*.

Definición 1.7. *Un* algoritmo *consiste en un conjunto ordenado de operaciones sistemáticas, que permite hacer un cálculo y hallar la solución de un problema. Dados un estado inicial y unos parámetros de entrada, siguiendo los pasos sucesivos se llega a un estado final, obteniéndose así dicha solución.*

A esta transición desde el estado inicial al estado final la llamaremos *ejecución* del algoritmo.

Para que un algoritmo pueda resolver una instancia de un problema de optimización, debe tener las siguientes propiedades.

- *Validez*: el algoritmo es válido si efectivamente el estado final es una solución óptima s_{opt}.

- *Finitud*: el algoritmo debe terminar su ejecución en finitos pasos.

1.2.2. Complejidad algorítmica

Supongamos que tenemos un problema, y dos algoritmos que lo resuelven. En este caso, ¿Cuál resulta la mejor elección? La respuesta es: el algoritmo más *eficiente*, es decir, el algoritmo que menos recursos en tiempo y espacio (de memoria) consuma.

La *complejidad* o *costo de un algoritmo* es una medida de los recursos (tiempo, memoria) que requiere su ejecución en función del *tamaño* de la

instancia del problema. El tamaño de una instancia viene dado por la cantidad de información que es necesaria para especificarla. Por ejemplo, en el problema de camino de mínimo costo, una instancia queda determinada por la cantidad de nodos en V, las ramas en E y los valores de los costos de estas ramas, y el tamaño de la instancia es esencialemente $|V| + 2|E|$ (cantidad de nodos más cantidad de arcos, más cantidad de costos). Dado que los costos sólo aumentan el tamaño real de la instancia en un factor fijo (2 veces el número de aristas), suele asumirse que el tamaño de una instancia viene dado simplemente por $|V|+|E|$. En general, cuanto mayor es el tamaño de la instancia puntual que se desea resolver, mayor es la dificultad para resolverla.

La complejidad de un determinado algoritmo estará dada por el número de operaciones que éste debe realizar para hallar la solución óptima, en función del tamaño de cada instancia particular. Para la realización de este cómputo se toman en cuenta las operaciones consideradas *elementales*. Es decir, típicamente: la suma y el producto de escalares y la asignación de valores a variables. Es importante observar que en muchos casos, operaciones en apariencia simples (como por ejemplo evaluar la función de costo) pueden resultar, en realidad, sumamente costosas, en la medida en que acumulan gran cantidad de operaciones elementales.

La principal característica de la complejidad, definida como una función del tamaño de la instancia, es que no depende de la capacidad de cada máquina para ejecutar el algoritmo, sino que resulta una medida *intrínseca* de la bondad del algoritmo.

En general, calcular el número exacto de operaciones que realizará un algorimo para resolver una determinada instancia puede resultar sumamente engorroso y poco práctico. Por lo tanto, expresaremos la complejidad no con una fórmula exacta, sino a través de un criterio de comparación:

Definición 1.8. *Sean $f, g : \mathbb{N} \to \mathbb{N}$ dos funciones. Diremos que el orden de crecimiento de g es* del orden *del de f y notaremos $g = O(f)$, si $\exists K \in \mathbb{R}_{>0}$ tal que $\forall n \in \mathbb{N} : g(n) \leq Kf(n)$, (excepto a lo sumo un número finito de valores de n).*

Típicamente, dado un cierto algoritmo, y asumiendo que se tiene una instancia de tamaño n, calcularemos la cantidad $c(n)$ de operaciones que el algoritmo deberá realizar en el peor de los casos y diremos que la complejidad es $O(c(n))$[1].

[1] A veces esta noción de complejidad recibe el nombre particular de *complejidad en el peor caso*. También puede calcularse la complejidad en el mejor caso y la complejidad en el caso promedio. La combinación de estas nociones permiten afinar el análisis de los algoritmos. Aquí seguiremos el enfoque clásico que se limita a la complejidad del peor caso.

1.2.3. Teoría NP

En esta subsección veremos cómo pueden clasificarse la mayoría de los problemas de optimización desde el punto de vista de la complejidad. Esta clasificación es sumamente importante en la práctica, ya que nos dará una idea *a priori* de la dificultad del problema.

Debido a que la teoría es muy extensa, abordaremos los conceptos fundamentales a título informativo, con el solo objeto de realizar un encuadre general del problema de Job Shop Scheduling.

Como la teoría NP se enfoca desde problemas de decisión, comenzaremos por definir que significa que un problema sea *de decisión*.

Definición 1.9. *Diremos que un problema es de decisión si la respuesta a dicho problema es SI ó NO.*

Por ejemplo, el siguiente problema es un problema de decisión:

Definición 1.10. *Dado un grafo $G = (V, E)$, entonces, un* camino hamiltoniano *de G será un camino que visita todos los vértices $v \in V$ una sola vez. Si además el último vértice visitado es adyacente al primero, el camino es un* ciclo hamiltoniano *o* circuito hamiltoniano*.*

Problema 2. Ciclo hamiltoniano

Dado un grafo $G = (V, E)$, ¿Existe un ciclo hamiltoniano en G?

Para ser más específicos, supongamos que tenemos una instancia del problema anterior dada por el grafo de la Figura 3. En rojo se puede ver un circuito hamiltoniano dentro del grafo. La numeración de los nodos es simplemente a modo de definir un comienzo y un final.

Si bien la instancia es pequeña (12 vértices y 22 aristas), no parece ser sencillo definir si hay un circuito hamiltoniano en G. No obstante, si de algún modo obtuviéramos el camino detallado en rojo, no nos sería díficil validar si es un circuito hamiltoniano.

Sobre este concepto se basa la teoría NP. Un problema de decisión será NP si nos es posible responder en forma eficiente si una posible solución es factible ó no. Más formalmente:

Definición 1.11. *Diremos que un algoritmo es polinomial si su complejidad es $O(n^k)$, donde n es el tamaño de la instancia y $k \in \mathbb{N}$.*

Definición 1.12. *Un problema de decisión pertenece a la clase NP (polinomial no-determinísticamente) si dada una instancia de SI y un candidato a solución s, existe un algoritmo polinomial que verifique si s es efectivamente solución o no.*

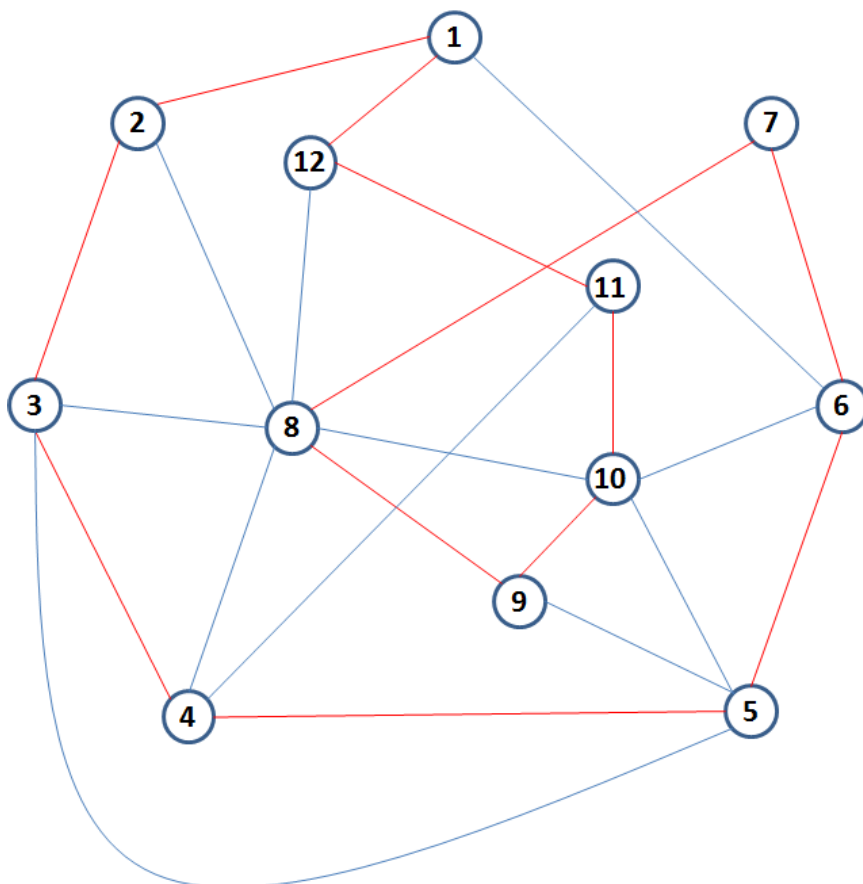


Figura 3: Grafo y circuito hamiltoniano (rojo)

Se deja entrever de la anterior definición que un algoritmo será eficiente si es polinomial. No obstante, un algoritmo polinomial no tiene por qué ser necesariamente eficiente, puesto que está claro que un algoritmo con complejidad $O(n^{1000})$ resulta inútil en la práctica pese a ser polinomial. Sin embargo, siguiendo la teoría clásica de problemas NP, asumiremos que un algoritmo polinomial es *bueno* en oposición a los algoritmos con complejida exponencial (por ejemplo, complejidad $O(2^n)$).

Otra observación importante es que verificar la validez de una solución siempre será más fácil que encontrar dicha solución. Aún así, en muchos problemas de decisión se puede hallar la solución que nos permite responder afirmativamente el problema en forma eficiente (es decir, mediante un algoritmo polinomial).

Definición 1.13. *Un problema de decisión pertenece a la clase P (polinomial) si existe un algoritmo polinomial para resolverlo. Es importante re-*

marcar que resolver *un problema de decisión implica, de ser posible, hallar una solución s que muestre que la respuesta al problema es SI y, en caso contrario, garantizar que la respuesta es NO.*

De las Definiciones 1.12 y 1.13 resulta sencillo observar que $P \subseteq NP$. En este sentido, se podría decir que los problemas en P son los problemas más *fáciles* de NP. Por este motivo, diremos que los problemas de la clase P son *tratables*.

En relación con la dificultad de cada problema, podemos definir un criterio mediante el cual se puedan comparar dos problemas de decisión y establecer cuál de ellos es el más díficil de resolver.

Definición 1.14. Π *y* Π' *problemas de decisión, que poseen respectivamente conjuntos* $I_\Pi, I_{\Pi'}$ *de instancias y conjuntos* $Y_\Pi \subset I_\Pi, Y_{\Pi'} \subset I_{\Pi'}$ *de instancias cuya respuesta es afirmativa.*

Diremos que $f : I_{\Pi'} \to I_\Pi$ *es una* reducción polinomial *de* Π' *en* Π *si* f *se computa por un algoritmo polinomial y* $\forall i \in I_{\Pi'} : i \in Y_{\Pi'} \Leftrightarrow f(i) \in Y_\Pi$.

Notaremos dicha reducción como $\Pi' \propto \Pi$.

Está claro que si $\Pi' \propto \Pi$ y contamos con un algoritmo g que resuelve Π polinomialmente, entonces podremos resolver Π' en forma eficiente utilizando el algoritmo polinomial $g \circ f$. Diremos, por lo tanto, que Π resulta al menos tan difícil de resolver como Π'.

Definición 1.15. *Un problema* Π *es* $NP - completo$ *si se cumplen:*

1. $\Pi \in NP$

2. $\forall \Pi' \in NP : \Pi' \propto \Pi$

Observemos que resolver eficientemente un problema $NP-c$ implicaría resolver eficientemente *todos* los problemas NP. Por esto, los problemas $NP-c$ son considerados como los problema NP más *díficiles*.

A priori, no está claro que existan problemas en la clase $NP - c$, ni cómo hallarlos.

Observemos que si f, g son reducciones polinomiales, entonces la composición de ambas $f \circ g$ $(g \circ f)$ también lo es. De aquí no es díficil deducir que vale la transitividad entre reducciones polinómicas:

Proposicion 1.2. *Si* Π, Π', Π'' *problemas de decisión tales que* $\Pi'' \propto \Pi'$ *y* $\Pi' \propto \Pi$*, entonces* $\Pi'' \propto \Pi$.

Ahora, supongamos que nuestro problema Π es NP, y sabemos que Π' es $NP - c$. Luego, si $\Pi' \propto \Pi$, entonces Π será $NP - c$. En efecto:

- $\Pi \in NP$
- $\forall \Pi'' \in NP, \Pi'' \propto \Pi' \wedge \Pi' \propto \Pi \Rightarrow \Pi'' \propto \Pi$ por la Proposición 1.2

Luego, para poder establecer si un problema es $NP-c$, necesitaremos hallar una reducción polinomial de un problema $NP-c$ en el anterior. Este hecho, hace que todos los problemas $NP-c$ sean equivalentes en dificultad.

En 1971, Cook demostró que el problema llamado $3-SAT$ es $NP-c$. A partir de este primer problema, utilizando reducciones polinomiales se ha probado la pertenencia a $NP-c$ de un gran número de problemas. Por ejemplo, el problema del camino hamiltoniano es $NP-c$.

Observemos que las definiciones de P y de $NP-c$ son cualitativamente diferentes. $NP-c$ se define a través de una propiedad que vincula a un problema en NP con los demás, mientras que la clase P es caracterizada de manera empírica a través de una propiedad (la existencia de un algoritmo polinomial) que en principio sólo depende del problema considerado. Si pudiésemos encontrar un problema Π en $P \cap NP-c$, entonces resolviendo Π y reduciendo todos los problemas de NP en Π, obtendríamos que $NP = P$. Dada que hasta el momento ha sido imposible tanto reducir un problema que se sabe en $NP-c$ a otro en P, como hallar un algoritmo polinomial para un algoritmo en $NP-c$, se sospecha fuertemente que la $NP-c \cap P = \emptyset$, y por lo tanto $NP \neq P$. Sin embargo, el problema de la relación entre P y $NP-c$ sigue abierto. Ha habido muchos intentos fallidos de demostraciones de que $P \cap NP-c = \emptyset$, y otros más tantos de que $P = NP$.

Ahora bien, dado que nos interesa trabajar con problemas de optimización buscaremos aplicar la teoría NP a este tipo de problemas. Nos preguntamos, entonces: ¿Cómo se relacionan los problemas de optimización con los problemas de decisión?

Todo problema de optimización admite una versión de decisión:

Definición 1.16. *Sea Π un problema de optimización. Dado k un número, podemos asociar a Π un problema de decisión dado por la pregunta: ¿existe una solución factible s de Π para I_Π tal que $f(s) \leq k$ si el problema es de minimización (o $f(s) \geq k$ si el problema es de maximización)?*

Para clarificar el tema, introduciremos el siguiente ejemplo:

Problema 3. Viajante de comercio

Dada una lista de ciudades, las distancias entre cada par de ellas y una ciudad de origen ¿cuál es la ruta más corta posible que visita cada ciudad exactamente una vez y regresa a la ciudad origen?

El *problema del viajante de comercio* (también conocido como *Traveling Salesman Problem* o TSP) es el ejemplo más estudiado y famoso dentro de la optimización combinatoria.

En su versión de decisión, el problema podría ser enunciado así:

Problema 4. Viajante de comercio, versión de decisión

Dado un número k, una lista de ciudades, las distancias entre cada par de ellas y una ciudad de origen ¿existe una ruta que sea más corta que k que visita cada ciudad exactamente una vez y regresa a la ciudad de origen?

Esencialmente, el TSP en su versión de decisión es equivalente a decidir si existe un circuito hamiltoniano entre las ciudades que tenga un costo final menor que k. Este problema resulta aún más díficil que el problema de decisión 2, ya que se ha incorporado una restricción sobre los pesos de las aristas.

A este tipo de problemas de optimización cuyos problemas de decisión son $NP{-}c$ se los llama $NP{-}HARD$. Las instancias medianas de estos problemas son realmente díficiles de resolver en tiempos razonables, mientras que las grandes resultan prácticamente imposibles de resolver en forma óptima. Por esta razón, este tipo de problemas son llamados *intratables*.

1.2.4. Algoritmos exactos y heurísticas

Si aceptamos la conjetura de que $NP \neq P$, debemos resignarnos a aceptar que no existen algoritmos polinomiales capaces de resolver problemas $NP - HARD$. Es decir: todo algoritmo que resuelva un problema $NP - HARD$ resultará al menos exponencial y su ejecución para instancias medianas demandará un tiempo inaceptable.

A modo de ejemplo, supongamos que tenemos una instancia del problema del viajante con 10 ciudades, todas interconectadas entre sí. El problema del viajante es $NP - HARD$, de modo que asumiremos que no existen algoritmos polinomiales que lo resuelvan. Si pretendiésemos aplicar, por ejemplo, un algoritmo de fuerza bruta, deberíamos analizar $15! \sim 10^{12}$ posibles recorridos del viajante. Si asumimos que una computadora moderna es capaz de computar el costo de un recorrido en un milisegundo segundos, el tiempo de resolución de la instancia sería de $10^9 s \sim 31$años.

En casos como este, la eficiencia del algoritmo resulta más importante que su validez, por lo que se prioriza el tiempo de ejecución por sobre la obtención de la solución óptima.

La práctica común es desarrollar algoritmos que intenten encontrar una solución factible lo más cercana posible al óptimo, pero que no necesariamente cumpla la condición de optimalidad.

De esta forma, distinguiremos dos grandes grupos de algoritmos:

- Algoritmos exactos: son aquellos para los cuales se tiene la certeza de que la solución devuelta es la solución óptima del problema. No tienen restricciones sobre su complejidad.
- Heurísticas: son algoritmos que encuentran una *buena* solución (es decir, una solución que podamos considerar aceptable según parámetros establecidos) y cuya complejidad es polinomial, por lo que son considerados eficientes. No obstante, la solución devuelta puede no ser óptima.

Para ejemplificar estas categorías, centrémonos en el problema 1.

Una posibilidad sería hacer un algoritmo que haga una lista de todos los posibles caminos entre el nodo inicial y el final, y luego, calcule en cuál de ellos se da el mínimo de la función de costo. Este algoritmo explorará todo el árbol de búsqueda descrito en la Figura 2 para la instancia propuesta. Esta clase de algoritmos son denominados *algoritmos de fuerza bruta* o *búsqueda exhaustiva* y, si bien son algoritmos exactos, resultan muy ineficientes. Su utilización se reduce a problemas de tamaño muy pequeño.

Por otro lado, podríamos ir construyendo un camino iterativamente, eligiendo las aristas con menor peso en cada paso, hasta llegar al final. Este tipo de algoritmos llamados *algoritmos golosos* son heurísticas. En la instancia propuesta del problema, este algoritmo elegirá el camino $s = v \to 1 \to 3 \to 5 \to w$ comparando sólo los costos de las opciones en cada solución parcial. El valor final es $f(s) = 10$ para este caso. No obstante, el óptimo de dicha instancia se da en el camino $s_{opt} = v \to 2 \to 4 \to 7 \to w$, con una función de costo final de $f(s_{opt}) = 9$.

2. Programación Dinámica

2.1. Descripción del método

2.1.1. Procesos de Decision en Múltiples Pasos

La Programación Dinámica (en adelante *PD*), fue desarrollada para el tratamiento y resolución de *Procesos de Decision en Múltiples Pasos*. Para dar una idea de estos procesos muy informalmente, podemos describirlos de la siguiente manera:

Un Proceso de Decision en Múltiples Pasos consta de un sistema físico que para cada instante de tiempo t se encuentra en un determinado *estado*, descripto mediante variables cuantificables. En determinados momentos, hay que tomar decisiones sobre este estado que tendrán un costo definido. Cada decisión produce una transformación del estado original en un nuevo estado, a tiempo $t + 1$. Así, se llega a un *estado final* o *terminal*, que es producto de una cadena de transformaciones anteriores. En este estado final se hace una valuación de las variables mediante una función objetivo. Idealmente, se quiere conocer la cadena de decisiones para que el costo de llegar a este estado final sea mínimo (o máximo).

Hay una gran cantidad de problemas que pueden ser formulados como procesos de decisión en múltiples pasos. Mencionaremos algunos rápidamente:

- Planificación de la producción
- Procesos de Markov
- Programación de pacientes en medicina clínica
- Políticas de inversión y asignación de presupuestos

2.1.2. El principio de optimalidad

Supongamos ahora que debemos resolver el problema del camino mínimo. Recordamos que el objetivo final del problema es hallar una camino mínimo entre un nodo inicial u y un nodo final v dentro del grafo pesado $G = (V, E, c)$. Idealmente, consideraremos que el grafo es conexo, por lo que el problema resulta factible.

Para resolver el problema, definiremos conjuntos de nodos V_i de la siguiente forma.

1. En principio, al nodo $V_0 = \{v\}$.

2. Definiremos V_1 como el conjunto de nodos adyacentes a los nodos de V_0, o, en este caso, adyacentes al nodo final v.

3. Así, podremos definir V_i como el conjunto de nodos adyacentes a los nodos de V_{i-1}

Siguiendo este procedimiento, como el grafo tiene finitos nodos y aristas, eventualmente llegaremos al nodo inicial u, el cual constituirá el conjunto V_n ($V_n = \{u\}$). n será, pues, la cantidad de nodos en el camino más largo entre u y v.

De esta forma, tendremos que $V = \bigcup_{i=0}^{n} V_i$. Además es claro que $V_i \cap V_j = \emptyset$ si $i \neq j$. Luego, el conjunto $\{V_i\}_{i=0}^{n}$ es una partición del conjunto de vértices V.

Supongamos ahora que tenemos un camino óptimo entre u y v: $u \to w_1 \to w_2 \to \cdots \to w_k \to v$. Observemos que, dado que el camino de u a v es óptimo, entonces el camino $w_j \to w_{j+1} \to \cdots \to v$ resulta un camino óptimo entre w_j y v. En efecto:

Demostración. Supongamos que no. Definiremos la función f que da el costo final de un camino dado de la siguiente forma:

$$f(x_1 \to x_2 \to \cdots x_n) = \sum_{k=1}^{n-1} c(x_i, x_{i+1})$$

Luego, existe un camino óptimo $w'_j \to w'_{j+1} \to \cdots \to v, w'_j = w_j$, diferente de $w_j \to w_{j+1} \to v$ tal que

$$f(w'_j \to w'_{j+1} \to \cdots \to v) < f(w_j \to w_{j+1} \to \cdots \to v)$$

Ahora, podremos construir un camino entre u y v como $u \to w_1 \to \cdots \to w_{j-1} \to w_j \to w'_{j+1} \to \cdots \to v$. Entonces

$$f(u \to \cdots \to w_{j-1} \to w_j \to w'_{j+1} \to \cdots \to v) =$$
$$= f(u \to w_1 \to \cdots \to w_{j-1}) + f(w_j \to w'_{j+1} \to \cdots \to v) <$$
$$< f(u \to w_1 \to \cdots \to w_{j-1}) + f(w_j + w_{j+1} \to \cdots \to v) =$$
$$f(u \to \cdots \to w_{j-1} \to w_j \to w_{j+1} \to \cdots \to v)$$

por lo que el camino inicial no es óptimo, lo que conlleva a un absurdo. □

Esta observación, en apariencia sencilla ó trivial, se denomina *Principio de optimalidad*, y constituye el pilar fundamental de la PD. Veamos cómo podemos aprovechar dicha propiedad para resolver el problema.

Definiremos $f : V \to \mathbb{R}$ la función que asigna a cada vértice $u \in V$ el costo del camino óptimo entre u y v. Luego, el principio de optimalidad puede expresarse de la siguiente forma:

$$f(u) = c_{uw} + f(w)$$

donde w es el nodo que sigue a u dentro del camino óptimo, y $c_{ww'}$ denota el peso de la arista (w, w').

De la anterior expresión podremos independizar el hecho de que w sea el siguiente nodo a u en el camino mínimo, de la siguiente forma:

$$f(u) = \min_{w:(u,w)\in E} c_{uw} + f(w)$$

Esta expresión se deduce básicamente de la anterior: si sabemos el camino óptimo entre el nodo w y v para cada w tal que $(u, w) \in E$, entonces el camino óptimo entre un nodo anterior u y v será el que agregue el mínimo costo a $f(w)$. A esta ecuación se la llama *ecuación funcional* ó *ecuación de Bellman*.

Ahora, definiremos $p : V\backslash V_0 \to V\backslash V_n$ a la función correspondiente de asignar a cada vértice w el camino óptimo entre w y v, es decir, $p(w)$ será el camino óptimo de una instancia del problema que toma a w como nodo inicial. Observemos que esta función en principio podría no estar bien definida, en el sentido de que puede haber más de un camino óptimo partiendo de w. No obstante, hay formas de resolver esta ambigüedad de forma adecuada, por lo que no nos preocuparemos por este problema.

Dentro de este contexto, nuestro objetivo final es hallar $f(u)$ y $p(u)$. Así, la función f nos permitirá obtener el costo del camino mínimo, mientras que p irá guardando como se construye dicho camino. Así, iremos calculando la solución iterativamente para cada $i = 0, 1, \cdots n-1$, de *atrás para adelante*, de la siguiente forma:

$$f(w) = \min_{w\in V_i, w'\in V_{i+1}, (w,w')\in E} c(w, w') + f(w')$$

$$p(w) = \arg\min_{w\in V_i, w'\in V_{i+1}, (w,w')\in E} c(w, w') + f(w')$$

El método planteado calcula primero $f(w)$ y $p(w)$ para cada $w \in V_1$. Luego, utiliza esta información para calcular $f(w)$, para cada $w \in V_2$. El proceso continua hasta llegar a $w \in V_{n-1}$, para luego obtener la solución óptima $f(u)$.

No obstante, hay que agregar que para calcular $f(w)$ y $p(w)$ para $w \in V_1$, necesitaremos saber $f(w)$ para $w \in V_0$, o, equivalentemente, $f(v)$. Esto no tiene sentido, ya que no existe un camino entre v y v. Podríamos pensar que el costo de llegar de v a v es nulo, por lo que definiendo $f(v) = 0$ queda bien determinados el resto de los cálculos.

Este tipo de condiciones que resultan necesarias para poder comenzar con las iteraciones se llaman *condiciones de borde*. Dichas condiciones se definen en términos de cada problema, y se aplican tanto a los nodos iniciales ó finales dependiendo de como se desarrolle el método.

Para que la exposición del método resulte más clara, resolveremos la instancia expuesta en la Figura 1.

En principio, hagamos la partición correspondiente a la instancia:

$$V_0 = \{w\},\ V_1 = \{5, 6, 7\},\ V_2 = \{3, 4\},\ V_3 = \{1, 2\},\ V_4 = \{v\}$$

Una vez hecha la partición, procederemos a calcular los valores de f y p iterativamente:

- $f(w) = 0$
- $f(5) = \text{mín}\{f(w) + c(5, w)\} = 0 + 2 = 2;\ \ p(5) = w$
- $f(6) = \text{mín}\{f(w) + c(6, w)\} = 0 + 2 = 2;\ \ p(6) = w$
- $f(7) = \text{mín}\{f(w) + c(7, w)\} = 0 + 4 = 4;\ \ p(7) = w$
- $f(3) = \text{mín}\{f(5) + c(3, 5), f(6) + c(3, 6)\} = \text{mín}\{2 + 2, 2 + 3\} = 4;\ \ p(3) = 5$
- $f(4) = \text{mín}\{f(6) + c(4, 6), f(7) + c(4, 7)\} = \text{mín}\{2 + 4, 4 + 1\} = 5;\ \ p(4) = 7$
- $f(1) = \text{mín}\{f(3) + c(1, 3)\} = 4 + 4 = 8;\ \ p(1) = 3$
- $f(2) = \text{mín}\{f(3) + c(2, 3), f(4) + c(2, 4)\} = \text{mín}\{4 + 3, 5 + 1\} = 6;\ \ p(2) = 4$
- $f(v) = \text{mín}\{f(1) + c(v, 1), f(2) + c(v, 2)\} = \text{mín}\{8 + 2, 6 + 3\} = 9;\ \ p(v) = 2$

Así, hemos hallado camino mínimo, que es $v \to 2 \to 4 \to 7 \to w$, y cuyo valor de retorno es 9.

Observemos que si hubiéramos utilizado un algoritmo de fuerza bruta, en este caso, tendremos que construir los 17 nodos del árbol de búsqueda

de la Figura 2. Luego, para construir los caminos y sus valores de retorno, tendremos que realizar en total 17 sumas. Finalmente, tendremos que hallar el mínimo de las 6 soluciones completas, lo que se puede hacer ordenando los valores de retorno de cada una de ellas. La mayoría de los algoritmos hacen esto en 5 operaciones aproximadamente, por lo que tendríamos que realizar como mínimo 22 operaciones. Utilizando PD hemos hecho 12 sumas y 4 comparaciones, lo que equivale a 16 operaciones básicas.

Se puede apreciar como se han reducido la cantidad de operaciones necesarias, aún cuando el tamaño de la instancia es muy chico. La potencia de la aplicación de PD se hace más notoria a medida que las instancias crecen en cantidad de nodos y de aristas. Por ejemplo, una arista entre el nodo $(6, 7)$ con peso 1 hubiera aumentado la cantidad de sumas en 3 y la cantidad de hojas en 2, por lo que necesitaríamos 5 operaciones adicionales para la fuerza bruta; mientras que para PD sólo necesitaremos 2 operaciones adicionales (1 suma y 1 comparación).

2.1.3. El método general

Para ilustrar la formulación mediante PD, supongamos que tenemos un Proceso discreto determinístico. Podemos describir este tipo de procesos como procesos de decisión en múltiples pasos en los que el sistema esta descrito a cualquier tiempo t por $x_t = (x_t^1, \cdots, x_t^m) \in \Omega \subseteq \mathbb{R}^m$, donde Ω representa una región del espacio restringida por la naturaleza del problema. A diferencia de estos procesos, existen problemas que tienen un cáracter estocástico, y, por lo tanto, el sistema es descrito con cierta *incertidumbre*, representada generalmente por funciones de distribución de probabilidad.

Introduciremos notación para facilitar la comprensión de los ejemplos y formalizar la noción de procesos de decisión.

Definición 2.1. *Definiremos los siguientes items:*

- $t \in \mathbb{N}$ *representará la variable temporal.*
- $x(t) = (x_1(t), \cdots, x_n(t))$ *serán las variables cuantificables.*
- *El par* $u = (t, x(t))$ *representará a cada estado. Llamaremos* $\mathcal{U}$ *al conjunto de estados.*
- *A su vez, sobre cada estado* u *habrá que tomar una decisión* d *de un conjunto de posibles decisiones que notaremos como* $D(u)$*.*
- *Al elegir* $d \in D(u)$*, el estado* u *se transformará a un nuevo estado* v*. Notaremos como* $T(u, d)$ *a la transformación así definida. Luego,* $v = (t + 1, x') = T(u, d)$*.*

La anterior notación sirve para definir la noción de estado, como también la forma en que éstos se transformarán en un nuevo estado. Es necesario saber cómo relacionar a cada estado con los costos y la función objetivo:

Definición 2.2. *Introducimos la siguiente notación:*

- *Llamaremos* $c(u, d)$ *al costo de transformar un estado* u *en un nuevo estado* $v = T(u, d)$.
- *Llamaremos* política *o* estrategia *a una función que para cada estado* u *asigna una decisión* $d \in D(u)$. *Naturalmente, el objetivo es obtener una política que retorne el mínimo (o máximo) costo posible. A una política con estas características la llamaremos* política óptima.
- *Notaremos* $f : \mathcal{U} \rightarrow \mathbb{R}$ *a la función de costo del proceso. Así,* $f(u)$ *será el costo óptimo del estado* u*, es decir: el mínimo (o máximo) costo en que puede incurrirse para alcanzar un estado terminal a partir del estado* u.

Naturalmente, el costo acumulado por el proceso en el estado u, no depende sólo de u, sino de la secuencia de decisiones posteriores para llegar a un estado terminal v. Así, $f(u)$ corresponde al costo de la *política óptima parcial*, desde el estado inicial u hasta v. En particular, si u es un estado inicial, $f(u)$ representa el valor de una política óptima.

El modo en que se *acumulen* los costos de las decisiones (c_{uv}) dependerá del problema. Típicamente, el costo acumulado es la suma de los costos de todas las decisiones tomadas. Sin embargo, la PD admite también otro tipo de funcionales, como por ejemplo: el máximo de los costos de cada decisión. No entraremos en detalles respecto de este tipo de cambios en el funcional.

Analizando la mecánica de un proceso de decisión en múltiples pasos notaremos que dado un estado particular a tiempo t, no resulta necesaria la información de los estados anteriores cuyas transformaciones nos condujeron al estado actual. Es decir, una vez transformado un estado u en un nuevo estado $v = T(u, d)$, u puede ser descartado: sea cual sea el costo acumulado para alcanzar v, este costo ya fue pagado y no puede deshacerse.

Lo dicho anteriormente da una nueva concepción a la noción de estado. El estado no será el resultado de una evaluación de todas las variables que intervienen en el problema a un determinado tiempo, sino que estará definido por la mínima cantidad de variables relevantes para inferir las decisiones posteriores. Mediante esta representación requerimos la cantidad minimal de información a cada tiempo t.

Notaremos como u_0 el estado inicial, u_t un estado a tiempo t. A su vez, asumiremos que estamos considerando un proceso de n etapas, es decir, $t \in \{0, \cdots, n\}$.

En este contexto, una política p quedará definida por la selección de las sucesivas transformacion $T_1, T_2, \cdots, T_n$, que representarán las decisiones tomadas sobre cada estado de cada etapa. Así, tendremos:

$$\begin{aligned} u_1 &= T(u_0, d(u_0)) = T_1(u_0) \\ u_2 &= T(u_1, d(u_1)) = T_2(u_1) \\ &\vdots \\ u_{n-1} &= T(u_{n-2}, d(u_{n-2})) = T_{n-1}(u_{n-2}) \\ u_n &= T(u_{n-1}, d(u_{n-1})) = T_n(u_{n-1}) \end{aligned}$$

Es decir, cada política estará definida por las decisiones $d \in D(u)$ tomadas en cada etapa.

En muchos casos se presenta más de un estado terminal posible. En estos casos, se suele definir un estado *ficticio* u_{n+1}, que será adyacente a todos los estados terminales (i.e $T_{n+1}(u_n) = u_{n+1}$), tal que $\forall u_n$, $c_{u_n u_{n+1}} = 0$. De esta forma, el estado terminal queda definido como u_{n+1} y $f(u_{n+1}) = f(u_n)$.

Una vez entendido lo anterior, el principio de optimalidad nos permite obtener una formulación análoga a la del problema del camino mínimo, en donde el peso de la arista (u, v) se corresponderá con el costo $c(u, d)$ de la transformación del estado u en el estado $v = T(u, d)$. Así, obtendremos las siguientes expresiones para la función de costo $f : \mathcal{U} \to \mathbb{R}$:

$$f(u_t) = c(u_t, d_{opt}(u_t)) + f(T(u_t, d_{opt}(u_t)),$$

donde $d_{opt}(u_t)$ representa la decisión tomada sobre el estado u_t correspondiente a la política óptima, y $f(u)$ es el mínimo (o máximo) costo desde u hasta u_{n+1}. Asumimos aquí por convención, que el costo total del proceso es la suma de los costos parciales.

De este modo, el principio de optimalidad nos permite hallar una expresión recurrente para la función f. Luego, la resolución sigue el mismo esquema iterativo que en el caso del camino mínimo.

Suponiendo que el problema trate de hallar el costo mínimo, la decisión óptima $d_{opt}(u_t)$ será el mínimo sobre todas las posibles decisiones $d \in D(u_t)$. Luego:

$$\begin{aligned} f(u_t) &= \min_{d \in D(u_t)} c(u_t, d) + f(T(u_t, d)) \\ T(u_t) &= \arg\min_{d \in D(u_t)} c(u_t, d) + f(T(u_t, d)) \end{aligned}$$

En el caso de que el problema sea de maximización, entonces simplemente con tomar $d_{opt}(u_t) = \max_{d \in D(u_t)} c(u_t, d)$, la ecuación funcional quedaría:

$$f(u_t) = \max_{d \in D(u_t)} c(u_t, d) + f(T(u_t, d))$$

$$T(u_t) = \arg\max_{d \in D(u_t)} c(u_t, d) + f(T(u_t, d))$$

2.1.4. Ejemplo práctico: el problema de la mochila

Problema 5. Problema de la mochila 0-1

Supongamos que tenemos una mochila, con capacidad para llevar un peso determinado $P \in \mathbb{N}$. Debemos elegir de un conjunto finito de objetos $O = \{o_1, \cdots, o_n\}$, algunos de ellos para llevarlos en la mochila. A su vez, cada objeto $o_i \in O$ tiene un determinado valor $v_i \in \mathbb{N}$, así como un peso $p_i \in \mathbb{N}$ conocido.

El problema consistirá en decidir cuáles objetos debemos poner en la mochila para que la suma de sus valores sea la máxima posible.

Observemos que una solución factible puede ser representada por un subconjunto $S \subseteq O$, que cumpla las restricciones de peso. Como O es un conjunto finito, $P(O)$ también lo será, por lo que el conjunto de soluciones factibles también será finito. Luego, el problema es efectivamente un problema de optimización combinatoria, por lo que tiene una solución óptima.

Dicho esto, modelaremos el problema así:

Definición 2.3. *Definiremos:*

- $\forall i = 1, \cdots, N: \ x_i \in \{0, 1\}$ *variables tales que:*

$$x_i = \begin{cases} 1 \textit{ si decido meter } o_i \textit{ en la mochila,} \\ 0 \textit{ caso contrario} \end{cases}$$

- $\forall S \in P(O) : V(S) = \sum_{o \in S} v_o = \sum_{i=1}^{N} x_i.v_i$
- $\forall S \in P(O) : P(S) = \sum_{o \in S} p_o = \sum_{i=1}^{N} x_i.p_i$

Una vez introducida esta notación, podemos plantear el problema como un problema de programación lineal entera:

$$max \sum_{i=1}^{N} x_i.v_i$$

Sujeto a:

$$\sum_{i=1}^{N} x_i.p_i \leq P$$

$$\forall i = 1, \cdots, N: \; x_i \in \{0,1\}$$

Si bien puede parecer *inocente*, el problema de la mochila pertenece a la clase $NP - HARD$.

El planteo con programación dinámica Aunque no se reconoce una variable t que mida el transcurso del tiempo, sí podemos pensar el problema de forma cronológica: es decir, ordenar los objetos bajo un determinado criterio (por ejemplo, el peso de cada objeto o el valor), para luego tomar una decisión sobre cada objeto, yendo desde el primer objeto hasta el último.

De acuerdo con esto, podremos dividir al problema en N etapas, en donde para cada etapa V_t, con $t = 0, 1, \cdots, N$, ya hemos tomado una decisión sobre los primero t objetos.

La noción de estado debe tener la información relevante para describir de manera precisa una etapa dada, que este caso viene dada por las decisiones tomadas respecto de los objetos ya considerados, y la capacidad restante en la mochila. Luego, un estado será representado simbólicamente por un par $u = (t, z), t \in \{0, 1, \cdots, N\}, z \in \{0, \cdots, P\}$, en donde t indica la cantidad de objetos sobre los que se ha tomado una decisión, y z es el peso que resiste la mochila.

Dada esta noción de estado, el conjunto de decisiones a tomar $D(u) = D(t, z)$ reflejará la elección sobre el objeto t-ésimo. En general, tendremos dos opciones:

1. Metemos o_t en la mochila, lo que implicará que $x_t = 1$. De esta forma $T(u) = (t + 1, z - p_t)$, con $z - p_t \geq 0$, y su costo será v_t.
2. Lo dejamos fuera, lo que implicará que $x_t = 0$ y que $T(u) = (t + 1, z)$, con un costo nulo.

Supongamos ahora que nos encontramos en una situación descrita por un estado inicial $u_0 = (0, P)$. Por lo tanto, resta decidir qué hacer con todos los objetos $\{o_1, \cdots, o_N\}$.

Definiremos $f(t, z)$ como el máximo valor que se puede obtener cuando aún falta decidir sobre los objetos $o_t, \cdots, o_N$ y la mochila puede soportar un peso z. Con este planteo, resulta sencillo ver que la solución del problema es $f(0, P)$, ó, lo que es lo mismo, f evaluada en el estado inicial u_0.

Para que el principio de optimalidad resulte válido, debemos comprobar que dada una asignación óptima $[x_1, \cdots, x_t, \cdots, x_N]$, entonces $[x_t, \cdots, x_N]$ resulta también óptima para el estado $u_t = (t, P - \sum_{i=1}^{t-1} x_i p_i)$, con $t = 1, \cdots, N$. En efecto:

Demostración. Sea $[x_1, x_2, \cdots, x_N]$ una solución óptima para el problema de la mochila partiendo de un estado inicial u_0.

Supongamos ahora que la conclusión es falsa. Luego, $[x_t, \cdots, x_N]$ no es una solución óptima partiendo de u_t, por lo que $\exists\ [x'_t, \cdots, x'_N]$ que si lo es, con $x_i \neq x'_i$ para algún $i = t, \cdots, N$. Entonces:

$$\sum_{i=t}^{N} x'_i v_i > \sum_{i=t}^{N} x_i v_i$$

Luego, se deduce que

$$\sum_{i=1}^{t-1} x_i v_i + \sum_{i=t}^{N} x'_i v_i > \sum_{i=1}^{t-1} x_i v_i + \sum_{i=t}^{N} x_i v_i$$

El término a la derecha se corresponde con el valor de retorno de la política $[x_1, x_2, \cdots, x_t, x_{t+1}, \cdots, x_N]$, mientras que el término a la izquierda es el valor de retorno de una política $[x_1, x_2, \cdots, x'_t, x'_{t+1}, \cdots, x'_N]$. Por lo tanto, $[x_1, x_2, \cdots, x_t, x_{t+1}, \cdots, x_N]$ no puede ser una asignación óptima partiendo de u_0, lo cual conlleva a una contradicción.

□

La ecuación funcional En base a lo dicho, la ecuación funcional será:

$$f(t, z) = \max\{f(t+1, z), f(t+1, z - p_t) + v_t\}$$

donde el término de la derecha hace referencia a las posibles decisiones a tomar sobre el objeto t.

Para fundamentar la ecuación antes definida, podemos argumentar que el máximo beneficio que puede ser obtenido cuando aún falta decidir qué hacer con los objetos $o_t, \cdots, o_N$, y teniendo la mochila una capacidad para soportar un peso z, es igual al máximo entre las posibles decisiones sobre el objeto o_t más el beneficio de la asignación óptima de los objetos $o_{t+1}, \cdots, o_N$, como indica el principio de optimalidad.

Establecida la recurrencia, debemos imponer condiciones de borde, que, dada la forma de la ecuación funcional, serán restricciones sobre los estados terminales. Dentro de este contexto, las condiciones de borde estarán definidas por dos sucesos:

- La asignación de la totalidad de los objetos, que se representará mediante un estado adicional $(N, z), z = 0, \cdots, P$. También es posible agregar un estado adicional $(N+1, z)$, en donde los costos de transformar cualquier estado (N, z) en $(N+1, z)$ serán nulos. Esta práctica se hace mucho para poder estandarizar el procedimiento de PD de forma análoga al de camino mínimo.

- La saturación del peso que soporta la mochila, que se representará mediante un estado $(t, 0), t = 1, \cdots, N$.

En ambos casos resulta imposible meter algún objeto en la mochila, por lo que no podremos sumar valor a ésta. De aquí que el máximo valor será nulo, lo que se traduce en las siguientes dos condiciones de borde:

$$f(N+1, z) = 0, z = 0, \cdots, P$$

$$f(t, 0) = 0, t = 1, \cdots, N$$

Así, el esquema de PD se podría representar de la siguiente forma:

$$\begin{cases} Hallar\ f(1, P) \\ f(t, z) = max\{f(t+1, z), f(t+1, z-p_t) + v_t\} \\ f(N+1, z) = 0,\ con\ z = 0, \cdots, P \\ f(t, 0) = 0,\ con\ t = 1, \cdots, N \end{cases}$$

Una vez establecido este esquema, podremos calcular la solución óptima iterativamente la ecuación funcional de hasta llegar a $f(1, P)$. Luego, de ser necesario, se reconstruye la política óptima desde el estado inicial hasta el estado final, o se va guardando a cada paso cuál es dicha política.

2.2. La programación dinámica en problemas de secuenciamiento

En esta subsección presentamos a modo de ejemplo los problemas de secuenciamiento, cuya formulación por PD fue presentada por Held y Karp en [8].

2.2.1. Definiciones y nociones básicas

Un problema de secuenciamiento es un problema dado por un conjunto de elementos A que debe ser ordenado de manera óptima según algún criterio.

Definición 2.4. *Sea un conjunto A finito, con $\#(A) = n$. Una* secuencia *s será una selección ordenada de algunos elementos de A. Así:*

$$s = [s_1, \cdots, s_m], \forall i = 1, \cdots, m : \; s_i \in A.$$

Diremos que la secuencia es parcial *si $m < n$ y que es* completa *si $m = n$.*

Notemos que cualquier secuencia s tiene asociado un subconjunto $B(s) \in P(A)$. Introduciremos la siguiente notación:

Definición 2.5. *Sea un conjunto A finito, y $s = [s_1, \cdots, s_m]$ una secuencia de A. Luego, definiremos $B_s = \{s_1, \cdots, s_m\}$ al conjunto de elementos que se encuentran en la secuencia s.*

Los problemas de optimización asociados a secuenciamiento establecen sus funciones de costos en relación con el orden de los elementos dentro de la secuencia. En general, se busca la secuencia óptima $s_{opt} = [s_1, \cdots, s_m]$, donde $B_{s_{opt}} = A$. Esta secuencia óptima constituirá una permutación de todos los elementos de A. Así, el espacio de búsqueda será de a lo sumo $(\#(A))!$ elementos. Probablemente, sea mucho menor que esta cantidad dado que hay que tomar en cuenta las restricciones propias del problema.

Este tipo de representación nos permite ir construyendo la solución óptima en forma iterativa, a partir de secuencias que irán creciendo en el número de elementos. Para ello, hagamos la siguiente observación:

Observación 2.1. *Sea $s = [s_1, \cdots, s_m]$, con $m < \#(A)$. Luego, podemos asociar un costo al agregar un elemento $x \notin B_s$, que genere una nueva secuencia s' de la siguiente forma:*

$$s' = [s_1, \cdots, s_m, x]$$

A esta operación binaria de encolamiento *se la denomina* expansión*, y se la suele denotar como $+$. Así, si expandimos s con x generando s', entonces escribiremos:*

$$s' = s + x$$

Repitiendo el anterior proceso iterativamente una cantidad finita de pasos, y teniendo en cuenta las restricciones propias del problema, podremos generar el espacio de soluciones factibles. Haremos, pues, la siguiente diferenciación:

Luego, en términos de la anterior definición, cualquier secuencia parcial representa una solución parcial, y puede ser expandida a una secuencia completa, obteniéndose así una solución factible del problema. Dentro de este

contexto, la noción de etapa resulta natural. Cada etapa del algoritmo estará marcada por las longitudes de las secuencias que interectuarán dentro de ella.

Todo parece indicar que cada estado estará representado por una secuencia, y que su transformación en un nuevo estado se realiza mediante la expansión. Sin embargo, no resulta claro cómo asociar a esta representación de estado una ecuación funcional. El problema radica en hallar un funcional que relacione todas las secuencias de longitud t.

Si reflexionamos sobre la noción de secuencia, observaremos que dada una secuencia se puede obtener la siguiente información:

- Los elementos incluidos en la secuencia.
- Los elementos que restan por incluir en la secuencia.
- La función de costo asociado a la secuencia.

Luego, notemos que si tuviésemos dos secuencias con los mismos elementos $s_1, s_2 : B_{s_1} = B_{s_2}$, el conjunto de elementos para expandir ambas secuencias resulta ser el mismo ($A \setminus B_{s_1}$).

Si, además, pudiésemos asegurar que cualquier forma de completar s_1 a una secuencia completa es también una forma de completar s_2, entonces un camino óptimo para s_1 será un camino óptimo para s_2. Bastará con tomar la secuencia mínima entre s_1 y s_2.

De esta forma, nuestros estados estarán descritos por los posibles subconjuntos de elementos B_s. A su vez, a cada estado podremos asociar una secuencia mínima, que consiste en tomar la secuencia con la mínima función de retorno dentro del conjunto de secuencias asociadas a B_s.

En este contexto, el principio de optimalidad establece que dada cualquier secuencia óptima $s = [s_1, s_2, \cdots, s_n]$, entonces para cualquier subsecuencia parcial $s_x = [s_1, s_2, \cdots, s_x]$, la expansión óptima será $[s_{x+1}, s_{x+2}, \cdots, s_n]$.

En la próxima subsección exhibiremos un ejemplo práctico de aplicación de PD a un problema de secuenciamiento con el objetivo de clarificar los conceptos de este apartado.

2.2.2. Ejemplo práctico: un problema de scheduling

En [8], los autores presentan tres problemas de secuenciamiento, y su planteo mediante PD. Además, exponen una heurística muy interesante.

El primero de dichos problemas es un problema de *scheduling*, o programación de tareas. A continuación, describiremos brevemente el problema y mostraremos de qué forma verlo como en un problema de secuenciamiento.

Deduciremos la ecuación funcional y desarrollaremos un algoritmo de PD para resolver el problema.

Problema 6. Problema de scheduling *Supongamos que tenemos n tareas $j_1, j_2, \cdots, j_n$, que deben ser ejecutados en una única máquina. Cada tarea j_k consume un tiempo determinado p_k, y tiene un costo asociado $c_k(t)$ que depende del tiempo t en que se completa su ejecución. Asumiremos que la máquina no puede ser interrumpida durante la realización de una tarea, y que puede trabajar en forma continua.*

Dadas estas condiciones, el problema es hallar una forma de programar todas las tareas de forma tal de obtener el menor costo posible.

Secuencias Intrínsecamente, el problema nos *induce* a una representación mediante secuencias. En este caso, una secuencia completa será una tira o vector $i_1, i_2, \cdots, i_n$, donde la tarea j_{i_k} debe ser la k-ésima en ejecutarse. De esta forma, a través de $i_1, \cdots, i_n$ se puede representar una permutación de las tareas. Una solución factible o *schedule* consistirá en $j_{i_1}, \cdots, j_{i_n}$.

Así, podemos redefinir el problema en estos términos:

- El costo asociado a toda solución factible $i_1, \cdots, i_n$ será $\sum_{k=1}^{n} c_{i_k}(t_{i_k})$.
- El tiempo total de terminación t_{i_n} será igual a $\sum_{k=1}^{n} p_{i_k}$

Luego, debemos hallar la solución que minimice el costo $\sum_{k=1}^{n} c_{i_k}(t_{i_k})$.

Ecuación funcional En su artículo, Held y Karp plantean un esquema de PD para resolver el problema. Para esto, introducen la siguiente notación:

- $B = \{k_1, k_2, \cdots, k_{|B|}\}$ será un subconjunto de $\{1, 2, \cdots, n\}$.
- $t_B = \sum_{k \in B} p_k$ será el tiempo de terminación de las tareas $j_{k_1}, \cdots, j_{k_{|B|}}$.
- $C(B)$ será el mínimo costo incurrido por las tareas $j_{k_1}, \cdots, j_{k_{|B|}}$ en el intervalo $[0, t_B]$.

Con estas definiciones, podremos escribir la ecuación funcional utilizando la misma idea que expresamos en el apartado de secuenciamiento:

$$\begin{cases} \#(B) = 1 \Rightarrow \forall k : C(\{k\}) = c_k(p_k) \\ \\ \#(B) > 1 \Rightarrow C(B) = \text{mín}_{k \in B}[C(B \backslash \{k\}) + c_k(t_B)] \end{cases}$$

Así, las secuencias de tareas que estarán relacionadas serán aquellas que tengan las mismas operaciones.

La ecuación funcional anterior puede justificarse de la siguiente forma: dado un orden óptimo $[i_1, \cdots, i_{|B|}]$ para el conjunto de tareas $\{j_{i_1}, \cdots, j_{i_{|B|}}\} = B$, la última tarea en ejecutarse es $j_{i_{|B|}}$. Luego, la ejecución de las anteriores tareas $\{j_{i_1}, \cdots, j_{i_{|B|-1}}\}$ debe resultar óptima para $B \backslash \{i_{|B|}\} = \{i_1, \cdots, i_{|B|-1}\}$. Demostremoslo:

Demostración. Supongamos que no resultará óptima. Entonces, existiría un orden óptimo para $B \backslash \{i_{|B|}\}$, que denotaremos como $\{i'_1, \cdots, i'_{|B|-1}\}$. Por lo tanto, tendremos que

$$\sum_{k=1}^{n} c_{i'_k}(t_{i'_k}) < \sum_{k=1}^{n} c_{i_k}(t_{i_k})$$

Como la tarea $j_{i_{|B|}}$ será la última también para la secuencia asociada a $\{i'_1, \cdots, i'_{|B|-1}\}$, entonces, se deduce que:

$$\sum_{k=1}^{n} c_{i'_k}(t_{i'_k}) + c_{i_{|B|}}(t_{i_{|B|}}) < \sum_{k=1}^{n} c_{i_k}(t_{i_k}) + c_{i_{|B|}}(t_{i_{|B|}}),$$

de donde no puede ser que $j_{i_1}, \cdots, j_{i_{|B|}}$ tenga costo mínimo para el conjunto B.

□

Así, el costo mínimo asociado a $B = \{i_1, \cdots, i_{|B|}\}$ resulta ser $C(B \setminus \{i_{|B|}\}) + c_{i_{|B|}}(t_B)$, de donde se deduce la recurrencia.

Finalmente, para resolver el problema mediante el esquema de PD propuesto, computaremos primero la condición de borde para cada $B : \#(B) = 1$, para luego iterar sobre los subconjuntos B con $\#(B) > 1$. A cada paso, iremos guardando la secuencia parcial óptima asociada a cada conjunto B.

2.3. Limitaciones de la programación dinámica

Desde el punto de vista formal, para poder aplicar PD es necesario asegurar la validez del principio de optimalidad. Esto implica dos problemas a priori.

El primero, naturalmente, es que no siempre se puede comprobar la validez del principio. Habrá problemas en donde no es posible establecer esta relación recursiva de forma que el problema se pueda dividir en subproblemas con la misma estructura.

El segundo problema consiste en la dificultad para establecer la recursión. Como hemos visto a través de este capítulo, la noción de estado y el funcional f estan intímamente relacionados. En este sentido, la noción de estado no

sólo debe contener la información relevante para describir una política óptima desde cualquier estado hasta los estados terminales, sino también, debe asegurar que la ecuación funcional sea válida. Dicha definición no resulta tan evidente en muchos casos.

Superados los problemas teóricos, aún tenemos graves limitaciones prácticas, que son producto de lo que se conoce como la *dimensionalidad* del problema. Si bien se ha dicho que la PD optimiza el uso de la información, ajustando en el mayor grado posible la cantidad utilizada, suele suceder que esta cantidad todavía resulta excesiva.

Si comparamos la PD con un algoritmo de fuerza bruta, nos daremos cuenta que ambos necesitan de dos ingredientes fundamentales:

- Una forma de construir las soluciones factibles.
- La búsqueda del mínimo sobre este universo de soluciones factibles.

La gran ventaja del primero sobre el segundo se debe al principio de optimalidad, que nos permite evitar el derroche de recursos en una gran cantidad de soluciones parciales que no conducen al óptimo. De esta manera, mediante la definición de cada estado, los caminos a analizar se reducen en gran medida. Visto desde este punto de vista, podríamos decir que la PD es una búsqueda exahustiva *mejorada*.

No obstante, en esencia no deja de ser una búsqueda exahustiva, hecho por el cual también adquiere su cáracter de algoritmo exacto. Por más óptima que resulte la reducción de estados, en muchos ejemplos prácticos el tamaño de las instancias terminará agotando los recursos disponibles.

Este problema dimensional impactará a nivel computacional, saturando la memoria física de la computadora y/o aumentando excesivamente los tiempos de ejecución. Si bien existen técnicas de programación que mejoran ambos problemas, con el aumento dimensional las instancias se tornan problemas intratables.

Para poner un ejemplo, observemos el problema presentado en 5. Según lo visto en este apartado, la complejidad computacional del algoritmo de PD resulta ser igual a $O(NP)$. Esto nos indica que crece linealmente en P, lo cual claramente es un incoveniente, ya que dos problema con la misma cantidad de variables pero con restricciones de peso $P_1, P_2, P_1 << P_2$ tendrán tiempos de ejecución totalmente diferentes.

Del análisis realizado de la complejidad se infiere que este aumento en el tiempo se debe a que la generación de estados aumenta con la capacidad P. De esta manera, si ponemos una capacidad P muy grande, lo que sucederá es que la tabla $T(t, z)$ sature la memoria puede suceder que el algoritmo sature

la memoria de la máquina en alguna etapa dada. Por poner un ejemplo, si en una etapa.

Esto aún empeora en el caso que se agregue una restricción espacial al problema:

Problema de la mochila 2 Supongamos que tenemos una mochila, con capacidad para llevar un peso determinado $P \in \mathbb{N}$, y que posee un determinado espacio físico $W \in \mathbb{N}$. Debemos elegir de un conjunto finito de objetos $O = \{o_1, \cdots, o_n\}$, algunos de ellos para llevarlos en la mochila. A su vez, cada objeto $o_i \in O$ tiene un determinado valor $v_i \in \mathbb{N}$, así como un peso $p_i \in \mathbb{N}$ y un espacio $w_i \in \mathbb{N}$ conocidos.

El problema consistirá en decidir cuáles objetos debemos poner en la mochila para que la suma de sus valores sea la máxima posible.

Como puede intuirse, la formulación mediante PD es totalmente análoga, salvo por el hecho de que los estados serán de la forma (t, z, y), donde y representará el espacio remanente en la mochila. De esta manera, observemos que la tabla ahora tendrá una dimensión adicional, por lo que el algoritmo tendrá una complejidad de $O(NPW)$. Esto aumenta en gran número la cantidad de estados.

Este comportamiento patológico hace que tanto la dimensionalidad del problema como la cantidad de restricciones influyan en forma crítica en la complejidad de los algoritmos de PD. En estos casos, el problema suele tratarse mediante la aplicación de heurísticas, aunque también se practica la reducción de estados mediante métodos alternativos.

2.4. Conclusiones y comentarios finales

Hemos introducido las nociones básicas de Programación Dinámica, junto con algunos ejemplos prácticos de aplicación. El objetivo de ha sido familiarizar al lector con algunos conceptos y nociones que reutilizaremos más adelante. Entre éstos, se destaca la analogía entre el problema práctico 6 y el esquema teórico propuesto por Gromicho y compañía [6] para resolver el Job Shop Scheduling.

3. Problema del Job-shop scheduling

3.1. Descripción del problema

El problema de *job shop scheduling* es una de las variantes de los conocidos *problemas de scheduling*. Básicamente, los problemas de scheduling son problemas de asignación, en donde se debe decidir cómo programar una serie de tareas a fin de optimizar el tiempo de finalización o los costos.

Estos problemas son conocidos dentro del ámbito de la optimización por su complejidad y dificultad. Han sido largamente estudiados, y se han propuesto muchísimas opciones de resolución a cada variante. No obstante, no se han hallado métodos satisfactorios para la resolución de instancias medianas en la gran mayoría de los casos. Aún con la capacidad de cómputo actual, las instancias grandes resultan prácticamente intratables.

Problema 7. Job Shop Scheduling
Supongamos que tenemos una determinada cantidad de tareas o jobs *a realizar. Cada tarea consta de operaciones que deben ser ejecutadas en un orden pre-establecido. A su vez, cada una de estas operaciones se realizará en una única estación de procesamiento (que denominaremos* máquina*) encargada de dicha parte del proceso, y tardará un tiempo definido.*

Todos los jobs tienen la misma cantidad de operaciones. No obstante, tanto el orden de las operaciones como sus tiempos de ejecución pueden variar de un job a otro.

Asumiremos además que las máquinas no pueden ser interrumpidas en medio del procesamiento de una operación. Tampoco, podremos sobrecargar la máquina ejecutando más de una operación a la vez.

El objetivo final del problema es encontrar una manera de programar las tareas en cada máquina, de forma de terminar lo antes posible la totalidad de las tareas asignadas.

El problema de job-shop scheduling aparece en numerosas ramas de la industria y sirve de modelo a diversos problemas prácticos: desde la producción de diferentes bienes en una línea de montaje, que inspira el modelo de problema, hasta la programación de horarios de servicios de transporte (e.g.: [4]). Por otra parte, existen numerosas variantes del problema que admiten ligeras diferencias en el planteo (varias máquinas que pueden realizar un mismo tipo de operación en paralelo, productos que pueden obtenerse a través de distinta combinación de operaciones, etc.).

A simple vista el problema parece bastante estructurado: la serie de restricciones expuestas anteriormente parecen acotar el conjunto de decisiones sobre cómo programar la totalidad de los jobs.

Pese a esto, si bien las restricciones acotan las posibilidades en que puede ser programada cada operación, la cantidad de soluciones factibles del problema excede por mucho lo que uno imaginaría, creciendo exponencialmente en el número de jobs y máquinas. Así, la complejidad se agudiza muy rápidamente, haciendo que muchos de los algoritmos exactos conocidos actualmente sean ineficientes aún en instancias de diez jobs y diez máquinas.

Es por esto que la investigación actual sobre el Job Shop Scheduling está orientada al desarrollo y mejora continua de heurísticas que permitan hallar soluciones buenas, pero que pueden resultar no ser óptimas.

Para simplificar notación, de aquí en adelante nos referiremos al problema por su abreviatura *JSSP* (Job Shop Scheduling Problem).

El resto de esta tesis está dedicada a la exposición detallada de un algoritmo exacto para el JSSP basado en ideas de Programación Dinámica, propuesto por Gromicho, van Hoorn, Saldanha-da-Gama y Timmer en [6] y, complementariamente, al desarrollo de algunas heurísticas obtenidas a partir de modificaciones de este algoritmo.

3.2. Definiciones y nociones básicas

Introduciremos notación básica para tratar al JSSP.

Se deben *programar* una cantidad n de jobs en m máquinas. Notaremos como $\mathcal{J} = \{j_1, j_2, ..., j_n\}$ al conjunto de jobs y como $\mathcal{M} = \{m_1, m_2, \cdots, m_m\}$ al conjunto de máquinas.

Así, cada job estará constituido por m operaciones, cada una de las cuales se ejecutará en una máquina diferente. De este modo, tendremos $n \times m$ operaciones. Notaremos como $\mathcal{O} = \{o_1, o_2, ..., o_{nm}\}$ al conjunto de las operaciones. La numeración será elegida de manera que las operaciones correspondientes al job j_i serán denotadas como o_{i+kn}, donde:

- i, con $i = 1, 2, \cdots, n$, representa el job correspondiente, siendo n la cantidad total de jobs.
- k, con $k = 0, 1, \cdots, m-1$, representa el orden de la operación relativo al job.

Por ejemplo, si se define una instancia de 3 jobs ($n = 3$) y 3 máquinas ($m = 3$), las operaciones correspondientes al primer job ($i = 1$) serán:

$$O_1 = \{o_p \in \mathcal{O} :\ p = 1 + 3k,\ 0 \leq k \leq m-1\} = \{o_1, o_4, o_7\}$$

Por lo tanto, las operaciones $o_1, \cdots, o_n$ se corresponderán con las primeras operaciones de cada job, las operaciones $o_{n+1}, \cdots, o_{2n}$, con las segundas operaciones de cada job, y así sucesivamente.

Luego, queda definida unívocamente la función que a cada operación le asigna el job al cual pertenece. Llamaremos a esta función j:

$$j : \mathcal{O} \to \mathcal{J}, \qquad j(o_i) = i \text{ mód } n,$$

donde mód representa a la función resto de la división entre enteros.

A su vez, consideraremos también las funciones:

- $p(o)$ = tiempo de procesamiento de la operación o.
- $m(o)$ = máquina donde debe ser procesada la operación o.
- $indice(o)$ = índice correspondiente a la operación o. Así $indice(o_i) = i \; \forall i$.

Por último, supondremos que los tiempos de procesamiento de cada operación son números naturales, por lo que

$$\forall o \in \mathcal{O} : \; p(o) \in \mathbb{N}$$

3.3. Schedules

3.3.1. Definición

Llamaremos *schedule* o *planificación* a una función $\psi : \mathcal{O} \to \mathbb{N}_{\geq 0}$, tal que dada una operación $o \in \mathcal{O}$, devuelve el tiempo $\psi(o)$ al que comienza a ejecutarse dicha operación. Diremos que un schedule es factible si respeta las restricciones del problema. Más formalmente:

Definición 3.1. *Sea $\psi : \mathcal{O} \to \mathbb{N}_{\geq 0}$ un schedule. Diremos que ψ es* factible *si se cumplen las siguientes condiciones:*

1. $\forall o_l, o_r \in \mathcal{O} : (j(o_l) = j(o_r) \; \wedge \; r < l) \Rightarrow \psi(o_r) + p(o_r) \leq \psi(o_l)$
2. $\forall o_l, o_r \in \mathcal{O} : (o_l \neq o_r \wedge \; m(o_r) = m(o_l)) \Rightarrow (\psi(o_r) + p(o_r) \leq \psi(o_l) \vee \psi(o_l) + p(o_l) \leq \psi(o_r))$

La primera condición de la Definición 3.1 indica que una operación no puede comenzar a ejecutarse en una máquina antes que se hayan ejecutado las anteriores operaciones correspondientes al mismo job; mientras que la segunda dice que dos operaciones diferentes no pueden tener solapamientos al ejecutarse en una misma máquina.

Naturalmente, cada schedule factible tendrá un tiempo de finalización:

Definición 3.2. *Sea ψ schedule factible, notaremos como $C_{\text{máx}}$ al tiempo de finalización de la totalidad de las operaciones, es decir:*

$$C_{\text{máx}}(\psi) = \max_{o \in \mathcal{O}} \{\psi(o) + p(o)\}$$

Con estas definiciones, resolver el problema resulta equivalente a encontrar un schedule factible ψ_{opt} que minimice el tiempo de finalización total:

$$C_{\text{máx}}(\psi_{opt}) = \min_{\psi \text{ } factible} \{C_{\text{máx}}(\psi)\}$$

Se deduce de lo anterior que para definir schedule factible, necesitamos verificar las dos condiciones sobre el conjunto de operaciones $\mathcal{O}$. No obstante, si tomamos un subconjunto de operaciones $Q \subset \mathcal{O}$, podremos extender la noción de schedule factible sobre este conjunto.

Definición 3.3. *Sea $\mathcal{O}$ el conjunto de operaciones de una instancia de JSSP, y sea $Q \subset \mathcal{O}, Q \neq \emptyset$ un subconjunto de operaciones. Llamaremos a una función $\psi : Q \to \mathbb{N}_{\geq 0}$* schedule parcial *si ψ es un schedule factible para Q.*

Los schedules parciales resultarán elementos vitales en nuestro trabajo, ya que podremos construir schedules factibles en forma iterativa, agregando a los subconjuntos de operaciones Q una operación $o \in (\mathcal{O} \backslash Q)$ hasta llegar a completar la totalidad de operaciones.

Proposicion 3.1. *Sea $\psi : \mathcal{O} \to \mathbb{N}_{\geq 0}$ un schedule factible. Si $\forall Q \subseteq \mathcal{O}$ definimos $\psi^Q : Q \to \mathbb{N}_{\geq 0}$, $\psi^Q(o) = \psi(o)$, entonces ψ^Q es un schedule parcial.*

Demostración. Basta con probar que ψ^Q es un schedule factible para Q.

Como ψ es factible y $Q \subseteq \mathcal{O}$, valen:

No obstante, dado que $\psi^Q(o) = \psi(o)$, valen que:

1. $\forall o_l, o_r \in Q : (j(o_l) = j(o_r) \ \wedge \ r < l) \Rightarrow \psi^Q(o_r) + p(o_r) \leq \psi^Q(o_l)$

2. $\forall o_l, o_r \in Q : (o_l \neq o_r \wedge \ m(o_r) = m(o_l)) \Rightarrow (\psi^Q(o_r) + p(o_r) \leq \psi^Q(o_l) \vee \psi^Q(o_l) + p(o_l) \leq \psi^Q(o_r))$

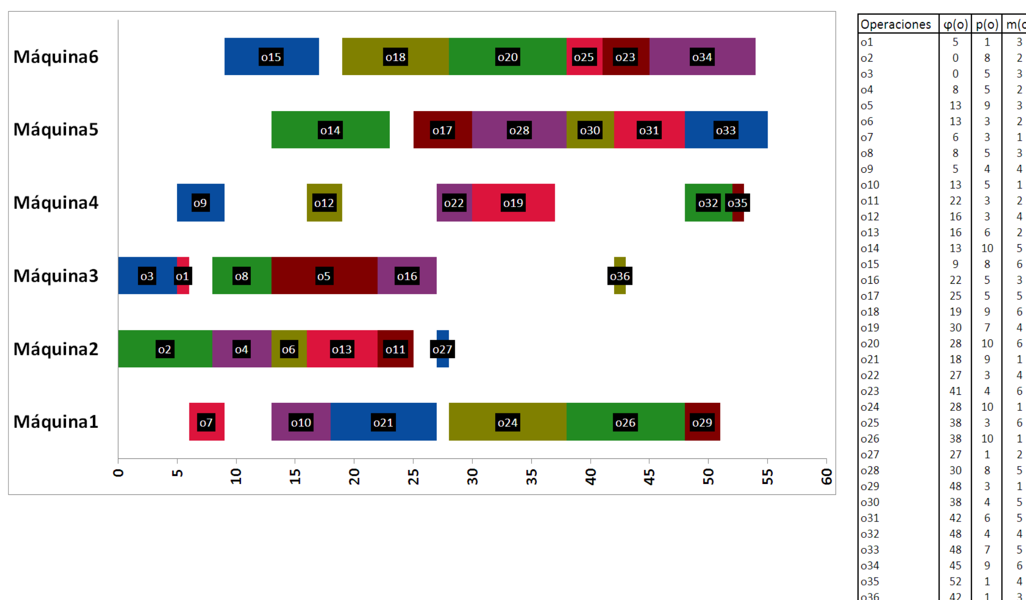
de donde se obtiene que ψ^Q es factible. □

La anterior proposición pone de manifiesto que cualquier *subschedule* de un schedule factible es, necesariamente, factible. Luego, para construir schedules factibles a partir de schedules parciales, necesitaremos asegurar la factibilidad de estos últimos.

3.3.2. Representación de las soluciones

A lo largo de este trabajo utilizaremos una forma conveniente y visual de representar schedules: *diagramas de Gantt*.

Los diagramas de Gantt son ideales para representar una asignación de procesos que ocupan lapsos temporales determinados. A modo de ejemplo, hemos expuesto el diagrama de Gantt de un schedule óptimo ψ para una instancia de seis jobs y seis máquinas conocida en la literatura como FT06.



Operaciones	φ(o)	p(o)	m(o)
o1	5	1	3
o2	0	8	2
o3	0	5	3
o4	8	5	2
o5	13	9	3
o6	13	3	2
o7	6	3	1
o8	8	5	3
o9	5	4	4
o10	13	5	1
o11	22	3	2
o12	16	3	4
o13	16	6	2
o14	13	10	5
o15	9	8	6
o16	22	5	3
o17	25	5	5
o18	19	9	6
o19	30	7	4
o20	28	10	6
o21	18	9	1
o22	27	3	4
o23	41	4	6
o24	28	10	1
o25	38	3	6
o26	38	10	1
o27	27	1	2
o28	30	8	5
o29	48	3	1
o30	38	4	5
o31	42	6	5
o32	48	4	4
o33	48	7	5
o34	45	9	6
o35	52	1	4
o36	42	1	3

Figura 4: Diágrama de Gantt FT06

Como se ve en la Figura 4, el eje x es asignado a la variable *tiempo*, mientras que el eje y se utiliza para mostrar las estaciones de trabajo o máquinas disponibles.

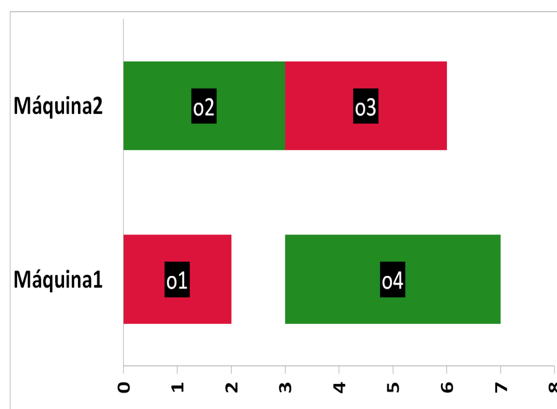
Dentro del área del gráfico se encuentran rectángulos con el nombre de cada operación, que representan el tiempo y la máquina utilizada por ésta. Por ejemplo, en la Figura 4 la operación o_5 se procesa en la Máquina 3 durante 9 unidades de tiempo, comenzando en el instante 13.

La gráfica anterior también nos permite comparar la representación formal del schedule con su diágrama de Gantt. Se observa que el diagrama de Gantt nos da una visión mucho más clara y concisa de cómo se distribuyen las operaciones que la simple exposición de ψ como función. Por ende, para facilitar la comprensión y clarificación de los conceptos, de aquí en adelante representaremos cualquier schedule (sea parcial o no) mediante diagramas de Gantt.

3.3.3. Orden entre operaciones

El concepto de schedule constituye el nexo entre las operaciones y el tiempo transcurrido para su ejecución. En este sentido, saber si una operación se ejecuta antes que otra crea un vínculo entre las operaciones programadas. Esta relación de precedencia temporal nos sugiere que todo schedule factible produce un orden propio entre sus operaciones. El siguiente ejemplo ilustrará dos posibles formas de establecer una relación de orden.

Ejemplo 3.1. *Supongamos que tenemos que* $\mathcal{O} = \{o_1, o_2, o_3, o_4\}$, ψ *schedule factible definido como sigue:*



1. *Si definimos la siguiente relación de orden entre operaciones...*

   $o_t \leq_\psi o_k$ *si se cumple algunas de las siguientes dos condiciones:*

   a) $\psi(o_t) < \psi(o_k)$

   b) $\psi(o_t) = \psi(o_k) \ \wedge \ m(o_t) \leq m(o_k)$

   tendremos que $o_1 <_\psi o_2 <_\psi o_4 <_\psi o_3$.

2. *Otro posible orden sería similar al anterior, pero respetando el tiempo de finalización de cada operación:*

   $o_t \leq_\psi o_k$ *si se cumplen algunas de las siguientes dos condiciones:*

   a) $\psi(o_t) + p(o_t) < \psi(o_k) + p(o_k)$

   b) $\psi(o_t) + p(o_t) = \psi(o_k) + p(o_k) \ \wedge \ m(o_t) \leq m(o_k)$

   En este caso, la jerarquía sería $o_1 <_\psi o_2 <_\psi o_3 <_\psi o_4$.

Notemos que si bien el schedule es el mismo, o_3 y o_4 intercambian *lugares* cuando cambiamos la relación de orden de 1) a 2). Es decir que el orden de las operaciones dependerá del criterio que utilicemos para establecer una precedencia temporal ligada a cada schedule ψ.

En el resto del trabajo adoptaremos como relación de orden entre operaciones la establecida en el ejemplo 2) y la notaremos con el operador binario $\leq_\psi$.

Veamos que, efectivamente, $\leq_\psi$ es una relación de orden entre las operaciones de una instancia del JSSP.

Proposicion 3.2. *Sea ψ factible, y el operador binario $\leq_\psi$ definido como en el segundo caso del Ejemplo 3.1. Son válidas:*

1. $\forall o \in \mathcal{O}, o \leq_\psi o.$
2. $\forall o_l, o_r \in \mathcal{O}, o_l \leq_\psi o_r, o_r \leq_\psi o_l \Rightarrow o_l = o_r.$
3. $\forall o_l, o_r, o_t \in \mathcal{O} : \ o_l \leq_\psi o_r, o_r \leq_\psi o_t \Rightarrow o_l \leq_\psi o_t$
4. $\forall o_l, o_r \in \mathcal{O} : \ o_l \leq_\psi o_r \vee o_r \leq_\psi o_l$

Es decir, para todo schedule factible ψ, la relación binaria $\leq_\psi$ define una relación de orden total en el conjunto de operaciones $\mathcal{O}$.

Demostración. 1. Sea $o \in \mathcal{O}$. Entonces, como $m(o) = m(o) \Rightarrow m(o) \leq m(o)$, se cumple que:

$$\psi(o) + p(o) = \psi(o) + p(o) \ \wedge \ m(o) \leq m(o) \Rightarrow o \leq_\psi o$$

2. Sean $o_l, o_r \in \mathcal{O}$. Luego, si $o_l \leq_\psi o_r$ entonces vale alguna de las siguientes afirmaciones:

$$\psi(o_l) + p(o_l) < \psi(o_r) + p(o_r) \tag{1}$$

$$\psi(o_l) + p(o_l) = \psi(o_r) + p(o_r) \wedge m(o_l) \leq m(o_r) \tag{2}$$

De la misma manera, como $o_r \leq_\psi o_l$, vale una de las siguientes afirmaciones:

$$\psi(o_r) + p(o_r) < \psi(o_l) + p(o_l) \tag{3}$$

$$\psi(o_r) + p(o_r) = \psi(o_l) + p(o_l) \wedge m(o_r) \leq m(o_l) \tag{4}$$

Entonces, si (1) resultará verdadera, tendríamos que $\psi(o_l) + p(o_l) < \psi(o_r) + p(o_r)$, por lo que (3) y (4) resultarían falsas.

Luego, no puede valer (1).

Analogamente, se puede ver que (3) no puede ser verdadera, ya que resultarían (1) y (2) falsas.

Por ende, deben valer (2) y (4). De lo que se deduce:

$$\psi(o_l) + p(o_l) = \psi(o_r) + p(o_r) \tag{5}$$

$$m(o_l) \leq m(o_r) \leq m(o_l) \Rightarrow m(o_l) = m(o_r) \tag{6}$$

De (5) y del hecho de que $\forall o \in \mathcal{O},\ p(o) \in \mathbb{N}$ se deduce que:

$$\psi(o_l) + p(o_l) > \psi(o_r) \tag{7}$$

$$\psi(o_r) + p(o_r) > \psi(o_l) \tag{8}$$

Como sabemos que ψ es factible, vale:

$$\forall o_l, o_r, o_l \neq o_r \ \wedge \ m(o_l) = m(o_r) \Rightarrow$$

$$\Rightarrow \psi(o_l) + p(o_l) \leq \psi(o_r) \ \vee \ \psi(o_r) + p(o_r) \leq \psi(o_l) \tag{9}$$

Pero como se cumple (7) y (8), resulta falsa la consecuencia de (9). Luego, por el contrarecíproco, debe ser falso que:

$$\forall o_l, o_r, o_l \neq o_r \ \wedge \ m(o_l) = m(o_r) \tag{10}$$

Finalmente, de (6) y de la falsedad de (10) se infiere que $o_l = o_r$.

3. Sean $o_l, o_r, o_t \in \mathcal{O}$ tales que $o_l \leq_\psi o_r$ y $o_r \leq_\psi o_t$.

   Para ver que $o_l \leq o_t$, debemos chequear que algunas de las dos condiciones sea verdadera:

   a) $\psi(o_l) + p(o_l) < \psi(o_t) + p(o_t)$

   b) $\psi(o_l) + p(o_l) = \psi(o_t) + p(o_t) \wedge m(o_l) \leq m(o_t)$

   Como $o_l \leq_\psi o_r$ entonces vale alguna de las siguientes afirmaciones:

$$\psi(o_l) + p(o_l) < \psi(o_r) + p(o_r) \tag{11}$$

$$\psi(o_l) + p(o_l) = \psi(o_r) + p(o_r) \wedge m(o_l) \leq m(o_r) \tag{12}$$

   Analogamente, de que $o_r \leq_\psi o_t$ se deduce que vale alguna de las siguientes afirmaciones:

$$\psi(o_r) + p(o_r) < \psi(o_t) + p(o_t) \tag{13}$$

$$\psi(o_r) + p(o_r) = \psi(o_t) + p(o_t) \wedge m(o_r) \leq m(o_t) \tag{14}$$

   Luego, tenemos los siguientes casos:

- (11) y (13) resultan válidas:

$$\psi(o_l) + p(o_l) < \psi(o_r) + p(o_r) < \psi(o_t) + p(o_t) \Rightarrow a)\ es\ Verdadera$$

- (11) y (14) resultan válidas. Entonces:

$$\psi(o_l) + p(o_l) < \psi(o_r) + p(o_r) = \psi(o_t) + p(o_t) \Rightarrow a)\ es\ Verdadera$$

- (12) y (13) resultan válidas. Luego:

$$\psi(o_l) + p(o_l) = \psi(o_r) + p(o_r) < \psi(o_t) + p(o_t) \Rightarrow a)\ es\ Verdadera$$

- (12) y (14) resultan válidas. Se infiere que:

$$\begin{aligned} \psi(o_l) + p(o_l) = \psi(o_r) + p(o_r) &= \psi(o_t) + p(o_t) \\ &\text{y } m(o_l) \leq m(o_r) \leq m(o_t) \\ &\Rightarrow b)\ es\ Verdadera \end{aligned}$$

En todos los casos a) o b) son verdaderas. Por ende, $o_l \leq_\psi o_t$.

4. Sean $o_l, o_r \in \mathcal{O}$. Entonces, dado que $\psi : \mathcal{O} \to \mathbb{N}_{\geq 0}$, $\forall o \in \mathcal{O} :\ p(o) \in \mathbb{N}$, y que $\mathbb{N}$ es un conjunto ordenado, se cumple alguna de las siguientes afirmaciones:

   a) $\psi(o_l) + p(o_l) < \psi(o_r) + p(o_r)$
   b) $\psi(o_l) + p(o_l) = \psi(o_r) + p(o_r)$
   c) $\psi(o_l) + p(o_l) > \psi(o_r) + p(o_r)$

   En el caso a), tenemos que $o_l \leq_\psi o_r$. Analogamente, en el caso c), se deduce que $o_r \leq_\psi o_l$.

   En el caso b), basta con observar que si:

$$\begin{cases} m(o_l) \leq m(o_r) \Rightarrow o_l \leq_\psi o_r \\ m(o_l) > m(o_r) \Rightarrow o_r \leq_\psi o_l \end{cases}$$

□

Los siguientes resultados son familiares y comunes a cualquier relación de orden, por lo que dejaremos de lado la demostración formal de cada uno de ellos. Dichas demostraciones pueden encontrarse en cualquier apunte o libro que trate el tema *Relaciones de orden*.

Definición 3.4. *Sea ψ un schedule factible. Basándonos en la relación de orden de 3.2, definiremos la operación $<_\psi$ asimétrica entre operaciones de $\mathcal{O}$ de la siguiente forma:*

$$\forall o_l, o_r \in \mathcal{O} : o_l <_\psi o_r \Leftrightarrow (o_l \leq_\psi o_r \wedge o_l \neq o_r)$$

Definición 3.5. *Un elemento b perteneciente al subconjunto B se denomina* mínimo *respecto de la relación de orden $\leq$ si es anterior a todos los elementos de B, i.e:*

$$b\, es\, minimo \Leftrightarrow \forall x \in B \Rightarrow b \leq x$$

Definición 3.6. *Diremos que un conjunto A es* bien ordenado *si $\forall B \subseteq A$ subconjunto, B tiene un elemento minimo.*

Lema 3.1. *Sea A un conjunto finito. Luego, A es totalmente ordenado $\Leftrightarrow A$ es bien ordenado.*

Lema 3.2. *Sea A un conjunto finito y totalmente ordenado. Entonces, $\forall B \subseteq A : \exists! b$ minimo de B.*

Corolario 3.1. *Sea ψ factible y $Q \subseteq \mathcal{O}$ un subconjunto. Entonces, $\exists! b \in Q$ elemento mínimo de Q.*

Demostración. Por lo visto en la Proposición 3.2, $\forall \psi$ factible, $\mathcal{O}$ es un conjunto finito y totalmente ordenado. Del Lema 3.1 se infiere que $\mathcal{O}$ es un conjunto bien ordenado. Luego, del Lema 3.2 tenemos que $\forall Q \subseteq \mathcal{O}, Q$ tiene un único elemento mínimo. □

El anterior corolario nos permite establecer un orden entre las operaciones para cada ψ schedule factible, sea parcial o no. Luego, $\forall \psi$ factible podremos escribir $\mathcal{O} = \{o_{i_1}, \cdots, o_{i_{nm}}\}$, donde $r < l \Leftrightarrow o_{i_r} <_\psi o_{i_l}$. El orden de las operaciones dependerá del schedule ψ.

Introduciremos alguna notación adicional que será útil para simplificar la escritura.

Definición 3.7. $\forall o \in \mathcal{O}$, *definiremos:*

- $\mathcal{J}(o) := \{o' \in \mathcal{O} : j(o) = j(o')\}$.
- $\mathcal{M}(o) := \{o' \in \mathcal{O} : m(o) = m(o')\}$.
- $\mathcal{J}_{pre}(o) := \{o' \in \mathcal{O} : j(o) = j(o') \wedge indice(o') < indice(o)\}$.

Como trabajaremos mucho con schedules parciales, es importante ver qué condiciones resultan necesarias sobre un subconjunto Q de operaciones para poder definir estos schedules sobre Q.

Definición 3.8. *Sea $Q \subseteq \mathcal{O}$ un subconjunto de operaciones. Diremos que Q es* factible *si $Q \neq \emptyset \wedge (\forall o \in Q : \mathcal{J}_{pre}(o) \subseteq Q)$.*

Si, por ejemplo, tuviéramos una instancia de dos operaciones con dos máquinas, entonces el siguiente subconjunto de operaciones $Q = \{o_1, o_4\}$ no cumplirá lo antes dicho, ya que la operación o_2 precede a la o_4 y $o_2 \notin Q$. Resulta claro que para definir un schedule parcial sobre Q es necesario que $o_2 \in Q$ si $o_4 \in Q$, ya que de otra forma se violaría la segunda condición de factibilidad inefectiblemente.

3.3.4. Schedules activos

Observemos que ninguna de las dos condiciones de factibilidad imponen restricciones sobre el problema en el sentido de minimizar el tiempo de finalización total. Por ejemplo, podríamos tener los siguientes dos schedules de la Figura 5:

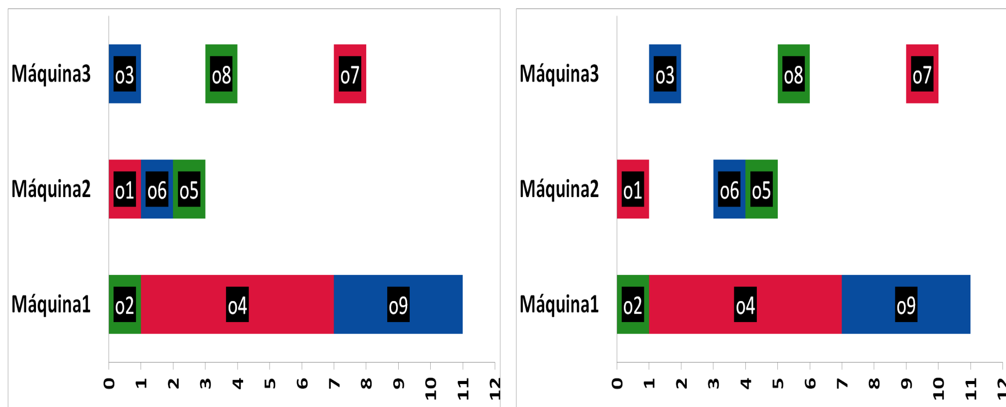


Figura 5: Schedule activo (izquierda) vs. Schedule no activo (derecha)

En principio, ambos schedules corresponden a la misma instancia de 3 jobs y 3 máquinas. Además, ambos resultan factibles y tienen el mismo orden entre las operaciones: $o_2 < o_1 < o_3 < o_6 < o_5 < o_8 < o_4 < o_7 < o_9$.

No obstante, notemos que el schedule de la derecha produce una demora en el inicio de la operación o_6 y, por lo tanto, también de o_5. Esta demora modifica los tiempos de inicio de estas operaciones, pero no interfiere con ninguna otra: por eso el orden de las operaciones en ambas planificaciones es el mismo.

Como nuestro objetivo es la minimización del tiempo de finalización, quisiéramos evitar schedules que tengan un tipo de comportamiento *ocioso*. El ejemplo muestra que no basta con las condiciones de factibilidad y orden para lograr este cometido, por lo que habrá que imponer condiciones adicionales.

Siguiendo a [11], llamaremos *activos* a los schedules como el de la izquierda. Así, un schedule activo será aquel en el que cada operación se inicie lo antes posible, sin introducir demoras que dejen las máquinas ociosas innecesariamente.

Definición 3.9. *Sea $Q \subseteq \mathcal{O}$ factible y $\psi : Q \to \mathbb{N}_{\geq 0}$ un schedule factible sobre Q. Notaremos como $Pre(\psi, o) = \{o' \in Q \cap \big(\mathcal{J}_{pre}(o) \cup \mathcal{M}(o)\big) : \psi(o') < \psi(o)\}$. Luego, diremos que ψ es activo si $\forall o \in Q$:*

$$\psi(o) = \begin{cases} 0 \ si \ Pre(\psi, o) = \emptyset \\ \text{máx}_{o' \in Pre(\psi,o)}\{\psi(o') + p(o')\} \ caso \ contrario \end{cases}$$

La anterior definición no hace más que formalizar la noción de schedule activo.

En base a lo dicho, no resulta muy díficil justificar la validez de la siguiente proposición. Debido a esto, dejaremos su demostración de lado.

Proposicion 3.3. *Sea ψ un schedule factible. Entonces, $\exists \psi'$ schedule factible y activo que respeta el mismo orden entre las operaciones que ψ y tal que $\forall o \in \mathcal{O} : \psi'(o) \leq \psi(o)$.*

Ahora sabemos que cualquier schedule factible ψ tiene un representante ψ' (podría ser el mismo schedule que ψ) dentro de los schedules activos para el cual las operaciones se planifican lo antes posible. Esta característica hace que el schedule ψ' sea un *mejor* schedule que ψ.

Corolario 3.2. *Sea ψ un schedule factible. Entonces, $\exists \psi'$ schedule factible y activo tal que que respeta el mismo orden entre las operaciones que ψ y $C_{\text{máx}}(\psi') \leq C_{\text{máx}}(\psi)$. Si además, ψ resulta ser óptimo, entonces ψ' también lo será.*

Demostración. Por la proposición 3.3, tenemos que $\exists \psi'$ schedule factible y activo tal que $\psi'(o) \leq \psi(o)$. Por lo tanto, de la definición de $C_{\text{máx}}$ se deduce:

$$C_{\text{máx}}(\psi') = \underset{o \in \mathcal{O}}{\text{máx}}\{\psi'(o) + p(o)\} \leq \underset{o \in \mathcal{O}}{\text{máx}}\{\psi(o) + p(o)\} = C_{\text{máx}}(\psi)$$

lo que demuestra la primera parte del corolario.

Ahora, supongamos que ψ es una solución óptima del problema. Luego, sabemos que $\forall \psi''$ schedule factible: $C_{\text{máx}}(\psi) \leq C_{\text{máx}}(\psi'')$. En particular tomando $\psi' = \psi''$.

Por lo tanto, se deduce que $\forall \psi''$ schedule factible:

$$C_{\text{máx}}(\psi') = C_{\text{máx}}(\psi) \leq C_{\text{máx}}(\psi'')$$

de donde ψ' resulta ser óptima. □

El anterior corolario resulta de vital importancia, ya que reduce el espacio de búsqueda para sólo considerar schedules que sean activos.

Por último, introduciremos la siguiente notación que nos será muy útil:

Definición 3.10. *Sea $Q \subseteq \mathcal{O}$ un subconjunto de operaciones factible. Definiremos:*

$$\Psi(Q) := \{\psi : Q \to \mathbb{N}_{\geq 0},\ \psi\ factible\ y\ activo\}$$

3.4. Complejidad del problema

Es sabido que el JSSP resulta $NP - HARD$ para instancias con $m \geq 3$, ó $n \geq 3$. La demostración de estos hechos resulta engorrosa y creemos que escapa a los contenidos de este trabajo. No obstante, si el lector está interesado, puede encontrar estas demostraciones en [3].

4. Aplicación de programación dinámica al JSSP

4.1. Preliminares

En este capítulo se analizará en profundidad el algoritmo de PD desarrollado por Gromicho y compañía en su trabajo [6]. Además, se propondrán algunas variantes y métodos alternativos.

La notación original de [6] ha sido modificada en su gran mayoría, debido en parte a las diferencias idiomáticas, pero también a que ha sido necesario incorporar nueva notación y nuevos conceptos para poder desarrollar algunas demostraciones que en [6] estaban sólo esbozadas. Finalmente, debemos remarcar que encontramos problemas en la teoría desarrollada en [6] para probar que el algoritmo propuesto encuentra una solución óptima: varios de los resultados preliminares que conducen a esta conclusión son erróneos. En el Apéndice presentamos un ejemplo que respalda esta afirmación. En consecuencia, en esta tesis presentamos un camino alternativo para probar la corrección del algoritmo. La argumentación que aquí exponemos fue desarrollada en colaboración con J. Gromicho y J. van Hoorn, y aparecerá en [7].

A modo de ejemplo, aprovecharemos para introducir una instancia que utilizaremos recurrentemente a través del capítulo. Llamaremos a esta instancia, definida en el Cuadro 1, $\mathcal{I}$. En la Figura 6 se muestra la planificación óptima para esta instancia.

Operaciones	o_1	o_2	o_3	o_4	o_5	o_6	o_7	o_8	o_9
$p(o)$	2	2	2	4	1	1	1	3	3
$m(o)$	1	1	3	3	2	2	2	3	1

Cuadro 1: Instancia $\mathcal{I}$.

4.2. El JSSP como problema de secuenciamiento

En esta subsección se presentará el marco formal para poder plantear al JSSP como un problema de secuenciamiento, similar al presentado en la Subsección 2.2. Para ello mostraremos cómo puede asociarse cada planificación con una secuencia de operaciones y estudiaremos las ventajas e imperfecciones de esta asociación.

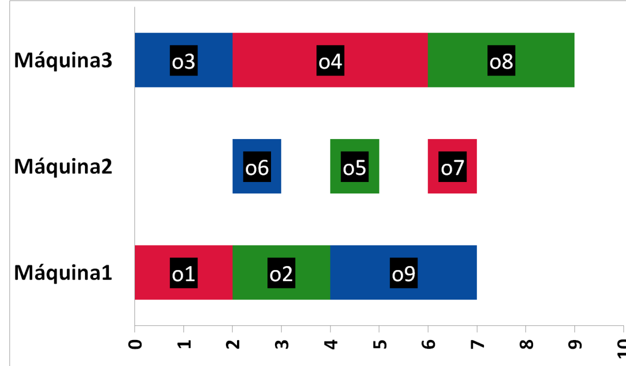


Figura 6: Planificación óptima para $\mathcal{I}$

4.2.1. Representación del problema mediante secuencias

Definición 4.1. *Una* secuencia *s será una tira de longitud $d(s)$, compuesta de operaciones de $\mathcal{O}$. Notaremos: $s = [o_{\sigma(1)}, \cdots, o_{\sigma(d(s))}]$, donde*

1. *$q(s) = Q = \{o_{\sigma(1)}, \cdots, o_{\sigma(d(s))}\}$.*
2. *$d(s) = |Q|$*
3. *$\sigma : \{1, \cdots, d(s)\} \to \{r \in \mathbb{N} : o_r \in Q\}$ inyectiva.*

La permutación σ será la encargada de asociar la posición dentro de la secuencia con cada una de las operaciones que la integran. Por esto, definiremos la igualdad entre secuencias de la siguiente forma:

Definición 4.2. *Sean $s^1 = [o_{\sigma_1(1)}, \cdots, o_{\sigma_1(x_1)}], s^2 = [o_{\sigma_2(1)}, \cdots, o_{\sigma_2(x_2)}]$ secuencias. Luego, $s^1 = s^2$ si y sólo si $\sigma_1 \equiv \sigma_2$*

Dado que las secuencias pueden ser vistas como tiras ordenadas de operaciones, utilizaremos la siguiente notación para relacionar subconjuntos de operaciones con la posición dentro de la secuencia.

Definición 4.3. *Sea $s = [o_{\sigma(1)}, \cdots, o_{\sigma(d(s)})]$ una secuencia, y sean $x, y : 1 \leq x \leq y \leq d(s)$. Definiremos:*

- *$s_{[x]}$ la $x-$ésima operación en s. Así, $s_{[x]} = o_{\sigma(x)}$.*
- *$s_{[x,y]} := [o_{\sigma(x)}, \cdots, o_{\sigma(y)}]$ representará la* subsecuencia *de operaciones de s que están entre las posiciones x e y.*

De forma análoga al capítulo anterior, como más adelante relacionaremos secuencias y schedules, debemos asegurar que las secuencias cumplan las condiciones mínimas de factibilidad de los schedules.

Definición 4.4. *Diremos que una secuencia s es factible si* $\forall x \in \{1, \cdots, k\}$ *se cumple:*

$$\mathcal{J}_{pre}(s_{[x]}) \subseteq \{s_{[1]}, \cdots, s_{[x-1]}\} = q(s_{[1,x-1]})$$

Observemos que cada secuencia s es una permutación de un conjunto de operaciones $q(s)$. De esta manera, podremos asociar a cada subconjunto de operaciones $Q \subseteq \mathcal{O}$ un conjunto de secuencias.

Definición 4.5. *Sea $Q \subseteq \mathcal{O}$ un subconjunto factible de operaciones. Definiremos el espacio de todas las secuencias asociadas a $Q \subseteq \mathcal{O}$ como $S(Q) = \{s \text{ secuencia } factible : q(s) = Q\}$.*

Por último, la cantidad de operaciones del conjunto $q(s)$ define la *longitud* de la secuencia s, que hemos denotado como $d(s)$. La longitud máxima será nm, que estará asociada a secuencias de $S(\mathcal{O})$. De esta manera, catalogaremos a las secuencias según su longitud de la siguiente forma:

Definición 4.6. *Sea una secuencia s. Diremos que:*

- *s es* parcial *si $d(s) < nm$*
- *s es* completa *si $d(s) = nm$*

Como mencionamos en la Subsección 3.2, toda planificación factible ψ establece una relación de orden temporal entre las operaciones de Q. Esta relación temporal nos indicará qué operación se ejecutará primero, cuál después, y así sucesivamente. Luego, cada permutación de las operaciones de Q representará un orden diferente de precedencia temporal (pero no por eso un schedule diferente). Las secuencias constituirán una representación de dichas permutaciones.

En lo que resta de esta subsección expondremos la forma en que se relacionan schedules y secuencias, así como también, bajo ciertos supuestos, demostraremos que hay una relación unívoca entre ambas.

El primer paso será ver cómo asignar a cada secuencia s un schedule ψ_s. Por lo visto en la Subsección 3.2, sería conveniente que este schedule ψ_s sea activo. Luego, definiremos la transformación de s a ψ_s de la siguiente forma:

Definición 4.7. *Sea $Q \subseteq \mathcal{O}$ un subconjunto factible, y $s \in S(Q)$ una secuencia. $\forall y \in \{1, \cdots, k\}$ definiremos*

- $Pre(s, y) = \{s_{[x]} \in (\mathcal{J}(s_{[y]}) \cup \mathcal{M}(s_{[y]})) : x < y\}$
- $\psi_s : Q \to \mathbb{N}_{\geq 0}$ *tal que:*

$$\psi_s(s_{[y]}) = \begin{cases} 0 \;\; si \; Pre(s, y) = \emptyset \\ \max_{o' \in Pre(s,y)}\{\psi_s(o') + p(o')\} \end{cases}$$

Proposicion 4.1. *ψ_s es factible.*

Demostración. Basta con verificar que ψ_s cumple ambas condiciones de factibilidad.

Sean $o_l, o_r \in Q : j(o_l) = j(o_r) \wedge r < l$. Como $s \in S(Q)$, entonces s es factible. Luego:

$$\exists x, y \in \{1, \cdots, k\}, x < y : o_r = s_{[x]} \wedge o_l = s_{[y]}$$

Como $j(o_l) = j(o_r)$, por definición, tendremos que $o_r \in Pre(s, y)$, por lo que $Pre(s, y) \neq \emptyset$.

Luego, $\psi_s(o_l) = \text{máx}_{o' \in Pre(s,y)}\{\psi_s(o') + p(o')\} \geq \psi_s(o_r) + p(o_r)$. Con esto hemos probado la primera condición. Veamos la segunda:

Sean $o_l, o_r \in Q : m(o_l) = m(o_r) \wedge r \neq l$. Supongamos, sin pérdida de generalidad que $r < l$.

Luego, podemos repetir el razonamiento anterior para deducir que $o_r \in Pre(s, y)$, ya que $m(o_l) = m(s_{[y]}) = m(o_r) = m(s_{[x]})$ y $x < y$. Nuevamente, tenemos que $\psi_s(o_l) = \text{máx}_{o' \in Pre(s,y)}\{\psi_s(o') + p(o')\} \geq \psi_s(o_r) + p(o_r)$, como queríamos ver.

El caso en que $l < r$ resulta análogo. □

De este modo, la Definición 4.4, nos garantiza que a partir de una secuencia s se pueda construir un schedule factible ψ_s.

Por otro lado, notemos que si s es parcial, entonces $|Q| < nm$, y, por lo tanto, ψ_s definirá una planificación parcial factible. De la misma forma, si s es completa, entonces ψ_s representará un schedule factible completo. Por esto es conveniente distinguir entre secuencias parciales y completas.

Por último, ψ_s tiene la importante característica de programar las operaciones lo antes posible, siempre manteniendo la factibilidad del schedule. Esta característica hace que ψ_s sea un schedule activo, como vemos a continuación:

Proposicion 4.2. *ψ_s es activo.*

Demostración. De las Definiciones 3.9 y 4.7 se deduce que bastará con ver que $\forall o \in Q : Pre(\psi_s, o) = Pre(s, o)$. Veámoslo:

- $Pre(\psi_s, o) \subseteq Pre(s, o)$:

  Sea $o' \in Pre(\psi_s, o)$. Luego, valen las siguientes condiciones:

  1. $o' \in Q$
  2. $(j(o') = j(o) \wedge indice(o') < indice(o)) \vee m(o') = m(o)$
  3. $\psi_s(o') < \psi_s(o)$

De la segunda condición se deduce que $o' \in (\mathcal{J}(o) \cup \mathcal{M}(o))$.

Por otro lado, sean $x, y : o' = s_{[x]} \wedge o = s_{[y]}$. Resta ver que $x < y$.

Supongamos que no. Luego, como $o' \neq o$ tendremos que $x > y$.

En cualquier caso, se cumple que:

- $s_{[y]} \in (\mathcal{J}(s_{[x]}) \cup \mathcal{M}(s_{[x]}))$
- $x > y$

Entonces, $s_{[y]} \in Pre(s, x)$. De aquí deducimos que:

$$\psi_s(o') = \max_{o'' \in Pre(s,x)} \{\psi_s(o'') + p(o'')\} \geq \psi_s(s_{[y]}) + p(s_{[y]}) =$$

$$= \psi_s(o) + p(o) \Rightarrow \psi_s(o') > \psi_s(o)$$

lo cual contradice que $o' \in Pre(\psi_s, o)$.

Del absurdo se infiere que $x < y$.

- $Pre(\psi_s, o) \supseteq Pre(s, o)$:

  Sea $o' \in Pre(s, o)$. Nuevamente, sean $x, y : o' = s_{[x]} \wedge o = s_{[y]}$. Luego, valen las siguientes condiciones:

  1. $j(o') = j(o) \ \vee \ m(o') = m(o)$
  2. $o' \in Q$
  3. $x < y$

  En el caso de que $j(o') = j(o)$, como s es una secuencia factible, entonces de la Definición 4.4 y del ítem 3 se deduce que $o' \in \mathcal{J}_{pre}(o)$, pues de otra manera se produciría una contradicción. En el caso de que $m(o') = m(o)$, entonces $o' \in \mathcal{M}(o)$. Luego, $o' \in (\mathcal{J}_{pre}(o) \cup \mathcal{M}(o))$.

  Resta ver que $\psi_s(o') < \psi_s(o)$. Por definición de ψ_s, se tiene que:

$$\psi_s(o) = \psi_s(s_{[y]}) = \max_{o'' \in Pre(s,y)} \{\psi_s(o'') + p(o'')\} \geq \psi_s(s_{[x]}) + p(s_{[x]}) =$$

$$= \psi_s(o') + p(o') \Rightarrow \psi_s(o) > \psi_s(o')$$

□

Ya podemos asociar a cada secuencia un schedule, que además resultará único. Pero para ver el problema como un problema de secuenciamiento, es deseable que podamos establecer una relación inversa entre schedules y secuencias. Para esto, utilizaremos la noción de orden definida en la Proposición 3.2.

Definición 4.8. *Sea $Q \subseteq \mathcal{O}$ factible tal que $|Q| = k$, y sea $\psi \in \Psi(Q)$ un schedule factible. Diremos que s es* ordenada *con respecto a ψ si se cumple que*

$$\forall x, y \in \{1, \cdots, k\} : x < y \Leftrightarrow s_{[x]} <_\psi s_{[y]}$$

En caso de no ser así, diremos que s es desordenada*.*

De esta manera podremos asociar a cada schedule una secuencia: asociaremos al schedule ψ la secuencia s_ψ que represente sus operaciones en forma ordenada. Es decir, si escribiésemos a Q en forma ordenada según ψ como $\{o_{\sigma(1)}, \cdots, o_{\sigma(k)}\}$, entonces $s_\psi = [o_{\sigma(1)}, \cdots, o_{\sigma(k)}]$. Además, s_ψ resultará ser única, como demostraremos a continuación.

Proposicion 4.3. *Sea $\psi \in \Psi(Q)$ una planificación factible. Entonces, $\exists s_\psi \in S(Q)$ secuencia ordenada con respecto a ψ y resulta ser única.*

Demostración. Por lo visto en la Proposición 3.2, sabemos que $\mathcal{O}$ es un conjunto totalmente ordenado bajo la relación de orden $\leq_\psi$. Luego, $\forall Q \subseteq \mathcal{O}$, Q también será totalmente ordenado. Por lo tanto, existe una única forma de escribir a Q en forma ordenada.

Tenemos que $\exists! \sigma : \{1, \cdots, k\} \rightarrow \{r : o_r \in Q\}$ tal que $\{o_{\sigma(1)}, \cdots, o_{\sigma(k)}\}$ es una representación ordenada de Q, donde $o_{\sigma(x)} <_\psi o_{\sigma(y)} \Leftrightarrow x < y$.

Difiniremos $s_\psi := [o_{\sigma(1)}, \cdots, o_{\sigma(k)}]$. Por definición, s_ψ es ordenada y es única.

Basta probar que s_ψ es factible para que $s_\psi \in S(Q)$.

Supongamos que s_ψ no sea factible. Luego, $\exists x, y \in \{1, \cdots, k\}$ tal que:

$$j(o_{\sigma(x)}) = j(o_{\sigma(y)}) \land \sigma(x) < \sigma(y) \land x > y$$

Luego, por definición de s_ψ, sabemos que $o_{\sigma(y)} <_\psi o_{\sigma(x)}$. Según la relación de orden en 3.2, se infiere que:

$$\psi(o_{\sigma(y)}) + p(o_{\sigma(y)}) \leq \psi(o_{\sigma(x)}) + p(o_{\sigma(x)})$$

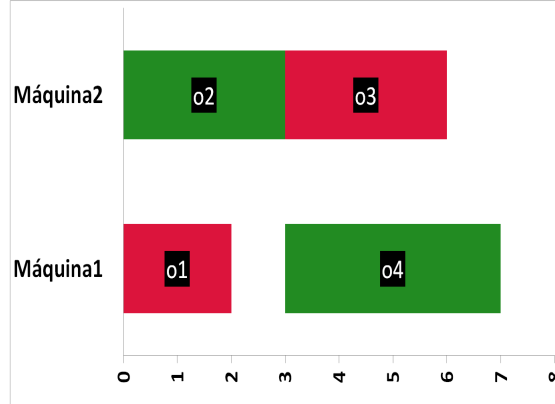
luego,

$$\psi(o_{\sigma(y)}) \leq \psi(o_{\sigma(x)}) + p(o_{\sigma(x)})$$

Lo cual contradice el hecho de que ψ es factible. Del absurdo obtenemos que s_ψ debe ser factible. ∎

Para fijar ideas, proponemos el siguiente ejemplo sobre un conjunto de operaciones Q.

Ejemplo 4.1. *Dado $\mathcal{O} = \{o_1, o_2, o_3, o_4\}$ y el siguiente schedule $\psi \in \Psi(\mathcal{O})$:*



- *La secuencia* $s_1 = [o_1, o_2, o_3, o_4]$ *es una secuencia completa ordenada.*
- *La secuencia* $s_2 = [o_1, o_2, o_4, o_3]$ *es una secuencia completa desordenada.*
- *La secuencia* $s_3 = [o_1, o_2]$ *es una secuencia parcial ordenada.*
- *La secuencia* $s_4 = [o_3, o_1]$ *es una secuencia parcial desordenada.*

Si observamos el ejemplo anterior, nos percataremos que dado un subconjunto de operaciones Q, existen secuencias $s \in S(Q)$ tales que la secuencia asignada a ψ_s no es igual a s. Esto sucederá, básicamente, cuando tomemos s desordenada, ya que $s' = s_{\psi_s}$ será ordenada, y por lo tanto $s' \neq s$. Sin embargo, podemos restringir $S(Q)$ para lograr esta propiedad de la siguiente forma.

Definición 4.9. *Sea* $Q \subseteq \mathcal{O}$ *factible. Definiremos:* $\mathring{S}(Q) = \{s \in S(Q) : s_{(\psi_s)} = s\}$.

La Definición 4.9 resulta de vital importancia para ver al problema como un problema de secuenciamiento, ya que nos asegura que al asignar un schedule ψ_s a una secuencia s ordenada, la secuencia asignada a dicho schedule s_{ψ_s} será s.

Definición 4.10. *Sea* $Q \subseteq \mathcal{O}$ *factible y* $s \in S(Q)$*. Definiremos:*

$$C_{\text{máx}}(s) = C_{\text{máx}}(\psi_s) = \max_{o \in Q}\{\psi_s(o) + p(o)\}$$

El $C_{\text{máx}}(s)$ será el valor del tiempo de finalización del schedule asociado a s. De esta forma, una secuencia óptima será una secuencia completa cuyo $C_{\text{máx}}$ resulte mínimo entre las secuencias completas. Nos permitiremos el abuso de notación ya que creemos resultará claro para el lector diferenciar schedules de secuencias.

Proposicion 4.4. *Sea $Q \subseteq \mathcal{O}$ factible, $|Q| = k$ y $s \in \mathring{S}(Q)$. Entonces,*

$$C_{\text{máx}}(s) = \psi_s(s_{[k]}) + p(s_{[k]})$$

Demostración. Como s es ordenada, entonces $\forall x, y \in \{1, \cdots, k\}$:

$$\psi_s(s_{[x]}) + p(s_{[x]}) \leq \psi_s(s_{[y]}) + p(s_{[y]}) \Leftrightarrow x \leq y$$

Luego, $\forall x \in \{1, \cdots, k\}$:

$$\psi_s(s_{[x]}) + p(s_{[x]}) \leq \psi_s(s_{[k]}) + p(s_{[k]}) \Rightarrow$$

$$\Rightarrow \forall o \in Q : \psi_s(o) + p(o) \leq \psi_s(s_{[k]}) + p(s_{[k]})$$

De aquí se infiere que

$$C_{\text{máx}}(s) = \max_{o \in Q}\{\psi_s(o) + p(o)\} = \psi_s(s_{[k]}) + p(s_{[k]})$$

□

La anterior proposición nos otorga una propiedad valiosa de las secuencias ordenadas respecto de su tiempo de finalización.

Observación 4.1. *Por último, observemos que dado un schedule ψ factible y activo, $s = s_\psi$ la secuencia ordenada correspondiente a ψ y s' una secuencia desordenada de forma que $\psi_{s'} = \psi$, entonces $\psi_s = \psi = \psi_{s'}$. Luego, se infiere que si tenemos una secuencia que es desordenada, al ordenar dicha secuencia las operaciones no cambian su tiempo de inicio, pues a ambas secuencias les corresponde el mismo schedule.*

4.2.2. Expansión de secuencias y generación de soluciones

En esta subsección nos aprovecharemos de la representación mediante secuencias para mostrar cómo se obtedrán schedules $\psi \in \Psi(\mathcal{O})$ con el fin de hallar la solución óptima. En las próximas subsecciones demostraremos la validez de este mecanismo y sentaremos las bases formales para el desarrollo de un esquema iterativo que nos permitirá resolver el JSSP.

En principio, observemos que para obtener schedules $\psi \in \Psi(\mathcal{O})$, necesitaremos secuencias que resulten completas. El procedimiento para obtenerlas será análogo al mecanismo de expansión expuesto en la Subsección 2.2.

Empezaremos por definir el operador binario *expansión* de una secuencia e introducir nueva notación.

Definición 4.11. *Sea $Q \subset \mathcal{O}$ factible con $|Q| = k$ ($k < nm$), $s \in S(Q)$ una secuencia parcial y una operación $o \in (\mathcal{O} \backslash Q)$.*

Llamaremos expansión *al operador binario que consiste en añadir la operación en el* último *lugar de s. Así, si $s' = expansion(s, o)$, entonces:*

- $q(s') = Q \cup \{o\}$
- $d(s') = d(s) + 1 = k + 1$
- $s' = [o_{\sigma'(1)}, \cdots, o_{\sigma'(k+1)}] \Rightarrow \forall x \in \{1, \cdots, k\} : \sigma'(x) = \sigma(x),\ o_{\sigma'(k+1)} = o$

Denotaremos al operador expansion *como* "$+$". *Luego:*

$$s' = expansion(s, o) = s + o$$

Definición 4.12. *Sea s^1 una secuencia parcial de longitud k, y sea s^2 una secuencia tal que $q(s^2) = \mathcal{O} \backslash q(s^1)$. Definiremos la operación de* completar a s^1 con la secuencia s^2, *cuyo resultado será la secuencia completa producto de expandir iterativamente a s^1 con las operaciones de s^2 según el orden de aparición. Notaremos a esta operación binaria entre dos secuencias con operaciones complementarias como $\oplus$. Así:*

$$(s^1 \oplus s^2)_{[1,k]} = s^1 \wedge (s^1 \oplus s^2)_{[k+1,nm]} = s^2$$

Definición 4.13. *Sea $Q \subset \mathcal{O}$ factible con $|Q| = k$ ($k < nm$), y $s \in S(Q)$. Llamaremos* completación *de s al conjunto de secuencias completas obtenidas al ir expandiendo s en forma iterativa y lo notaremos como $Comp(s)$. Así:*

$$Comp(s) := \{s \oplus s', s' \in S(\mathcal{O} \backslash Q)\}$$

A su vez, definiremos el conjunto de schedules asociados a $Comp(s)$:

$$Comp_\psi(s) := \{\psi \text{ schedule} : s_\psi \in Comp(s)\}$$

Por último, definiremos $\overset{\circ}{Comp}(s)$ y $\overset{\circ}{Comp}_\psi(s)$ de forma análoga, pero restringiendo la completación a sólo secuencias ordenadas. Luego:

$$\overset{\circ}{Comp}(s) := Comp(s) \cap \overset{\circ}{S}(\mathcal{O})$$

$$\overset{\circ}{Comp}_\psi(s) := \{\psi \text{ schedule} : s_\psi \in \overset{\circ}{Comp}(s)\}$$

Observación 4.2. *De las anteriores definiciones se infiere que:*

$$\forall s' \in Comp(s) : s'_{[1,k]} = s$$

Como $\overset{\circ}{Comp}(s) \subseteq Comp(s)$, también vale la propiedad anterior para secuencias $s' \in \overset{\circ}{Comp}(s)$.

De esta forma podremos construir secuencias completas al ir expandiendo iterativamente secuencias parciales.

Una vez descripto el mecanismo de expansión, lo que sigue es definir bajo qué condiciones se expandirán las secuencias. Es decir, dada una secuencia $s \in S(Q)$, ¿cuáles serán las operaciones o para expandir s?

Por un lado, observemos que no todas las operaciones $o \in (\mathcal{O} \backslash Q)$ sirven necesariamente para expandir s en $s' = s + o$ de forma que s' sea factible. Para que $s' \in S(Q \cup \{o\})$, necesitaremos que $Q \cup \{o\}$ sea un subconjunto de operaciones factible. Por otro lado, si s es ordenada, será importante ver si podremos expandir a s con o en $s' = s + o$, de forma tal que s' sea ordenada también.

Introduciremos la siguiente notación:

Definición 4.14. *Sea $Q \subseteq \mathcal{O}$ un subconjunto de operaciones factible.*

1. *Denotaremos como $\lambda(Q) \subseteq Q$ al conjunto de las últimas operaciones correspondientes a cada job que se encuentren en Q. Así:*

$$\lambda(Q) = \bigcup_{i \in \{1, \cdots, n\}} \Big\{ o_l : l = \max_{o \in Q : j(o) = i} \{indice(o)\} \Big\}$$

2. *Denotaremos como $\varepsilon(Q) \subseteq (\mathcal{O} \backslash Q)$ al conjunto de las próximas operaciones correspondientes a cada job. Es decir:*

$$\varepsilon(Q) = \bigcup_{i \in \{1, \cdots, n\}} \Big\{ o_l : l = \min_{o \in (\mathcal{O} \backslash Q) : j(o) = i} \{indice(o)\} \Big\}$$

3. *Sea $s \in \mathring{S}(Q)$ una secuencia ordenada. Luego, definiremos $\eta(s) \subseteq \varepsilon(Q)$ como el conjunto de operaciones que pueden expandir s en una secuencia ordenada. Más formalmente:*

$$\eta(s) = \Big\{ o \in \varepsilon(Q) : s + o \in \mathring{S}(Q \cup \{o\}) \Big\}$$

Observación 4.3. *Los siguientes resultados son válidos y serán de suma utilidad en el futuro. Creemos que son lo suficientemente claros como para justificar la ausencia de una demostración formal.*

- $\forall Q \neq \emptyset \; factible : \lambda(Q) \neq \emptyset$
- $\varepsilon(\mathcal{O}) = \emptyset$*, pues ya no hay más operaciones con que expandir.*
- $\forall Q \subseteq \mathcal{O} : \#(\lambda(Q)) \leq n$*, pues* $\#\Big\{ o_l : l = \max_{o_r \in Q, j(o_r) = i} \{r\} \Big\}$ *resulta ser 0 ó 1.*

- $\forall Q \subseteq \mathcal{O} : \#(\varepsilon(Q)) \leq n$, *pues* $\#\Big\{o_l : l = \min_{o_r \in (\mathcal{O} \setminus Q), j(o_r)=i} \{r\}\Big\}$ *resulta ser 0 ó 1.*
- *Se deduce de lo anterior que* $\forall Q \subseteq \mathcal{O}, s \in \mathring{S}(Q) : \#(\eta(s)) \leq n$.

Observemos que tanto $\varepsilon(Q)$ como $\eta(s)$ son conjuntos de operaciones que no están en Q, por lo que esperan a ser programadas. Además, $\varepsilon(Q)$ constituye el conjunto de todas las operaciones o tales que $Q \cup \{o\}$ resulta factible. Como queremos trabajar con secuencias factibles, debemos asegurar que cada vez que expandamos una secuencia, dicha expansión retorne una secuencia factible. Por esto, en principio, nos interesará expandir s sólo con operaciones de $\varepsilon(Q)$.

Proposicion 4.5. *Sea* $Q \subset \mathcal{O}$ *factible, y* $s \in S(Q)$. *Luego,* $\forall o \in \varepsilon(Q)$:

1. $Q \cup \{o\}$ *es factible.*
2. $s' = s + o \in S(Q \cup \{o\})$.
3. $\forall o' \in Q : \psi_{s'}(o') = \psi_s(o')$.

Demostración. Supongamos que $|Q| = k$.

1. Supongamos que $Q \cup \{o\}$ no es un conjunto factible. Luego, $\exists o^* \in Q \cup \{o\} : \exists o' \in \mathcal{J}_{pre}(o^*) \wedge o' \notin (Q \cup \{o\})$. Tenemos dos casos posibles:

   a) Si $o^* \in Q$, entonces como $o' \notin Q$, Q sería infactible, lo cual lleva a un absurdo.

   b) Si $o^* = o \Rightarrow o' \in \mathcal{J}_{pre}(o)$. Luego, valen:
      - $j(o') = j(o)$
      - $indice(o') < indice(o)$
      - $o' \notin Q$

      Por lo tanto, $o \neq o_l$, con $l = \min_{\{u \in (\mathcal{O} \setminus Q) : j(u) = j(o)\}} \{indice(u)\} \Rightarrow o \notin \varepsilon(Q)$, lo cual también es un absurdo.

   De lo anterior se deduce que $Q \cup \{o\}$ debe ser un conjunto factible.

2. Basta ver que s' es una secuencia factible. Como $s'_{[1,k]} = s$ y $s \in S(Q)$, entonces $\forall x \in \{1, \cdots, k\}$:

$$\mathcal{J}_{pre}(s'_{[x]}) \subseteq \{s'_{[1]}, \cdots, s'_{[x-1]}\}$$

   Si $x = k + 1$, entonces

- $s'_{[x]} = s'_{[k+1]} = o$
- $\{s'_{[1]}, \cdots, s'_{[x-1]}\} = \{s'_{[1]}, \cdots, s'_{[k]}\} = Q$

Luego, vale:

$$Q \cup \{o\} \; es \; factible \Leftrightarrow \mathcal{J}_{pre}(o) \subseteq Q \Leftrightarrow \mathcal{J}_{pre}(s'_{[x]}) \subseteq \{s'_{[1]}, \cdots, s'_{[x-1]}\}$$

con lo cual queda demostrado que s' es factible.

3. Como $s'_{[1,k]} = s$ tenemos que $\forall x \in \{1, \cdots, k\} : Pre(s, x) = Pre(s', x)$. De aquí y de la Definición 4.7 se deduce que

$$\forall x \in \{1, \cdots, k\} : \psi_s(s_{[x]}) = \psi_{s'}(s_{[x]}) \Rightarrow \forall o' \in Q : \psi_s(o') = \psi_{s'}(o')$$

como queríamos ver.

□

Como es de esperarse, cada vez que expandamos una secuencia, al aumentar el número de operaciones, también aumentará el tiempo de finalización. Lo dicho se demuestra en la siguiente Proposición:

Proposicion 4.6. *Sea $Q \subset \mathcal{O}$ un subconjunto factible, $s \in S(Q)$ una secuencia parcial y $o \in \varepsilon(Q)$. Entonces:*

$$C_{\text{máx}}(s) \leq C_{\text{máx}}(s + o)$$

Demostración. De la definición de $C_{\text{máx}}(s)$ se infiere que:

$$C_{\text{máx}}(s+o) = \max_{o' \in Q \cup \{o\}} \{\psi_s(o') + p(o')\} = \text{máx}\{\max_{o' \in Q}\{\psi_s(o') + p(o')\}; \psi_s(o) + p(o)\} =$$

$$= \text{máx}\{C_{\text{máx}}(s) + \psi_s(o) + p(o)\} \geq C_{\text{máx}}(s)$$

como queríamos ver.

□

Por último, introduciremos la siguiente notación para discriminar el tiempo al que inicia la expansión de una secuencia mediante cada operación de $\varepsilon(Q)$.

Definición 4.15. *Sea $Q \subset \mathcal{O}$ un subconjunto factible, $s \in S(Q)$ una secuencia parcial y $o \in \varepsilon(Q)$. Definiremos $\psi(s, o)$ como el menor tiempo en que puede ser programada o si s es expandida con o. Más formalmente:*

$$\psi(s, o) = \psi_{s+o}(o)$$

Con esto dicho, ya estamos en condiciones de obtener secuencias completas mediante la expansión iterativa de secuencias parciales. Así, podremos general todas las posibles schedules factibles y activos, para luego evaluar cuál minimiza el tiempo de finalización. El procedimiento se podría describir así:

1. Generamos todas las secuencias $s = [o_r]$, con $r = 1, \cdots, n$. Llamaremos al conjunto de secuencias actuales X.

2. Para cada secuencia $s \in X$, computamos el conjunto $\varepsilon(q(s))$. Generamos todas las secuencias $s' = s + o, o \in \varepsilon(q(s))$ y las agregaremos a X. Luego, eliminaremos la secuencia s de X.

3. Si las secuencias de X resultan completas, pasaremos al próximo paso. Sino, volveremos 2).

4. Calcularemos la solución óptima del problema como

$$s_{opt} = \arg\min_{s \in X} C_{\text{máx}}(s).$$

La Figura 7 describe el procedimiento anterior como un árbol de búsqueda, en donde cada nodo es una secuencia s y cada rama corresponde a la expansión de dicha secuencia con una operación determinada $o \in \varepsilon(q(s))$.

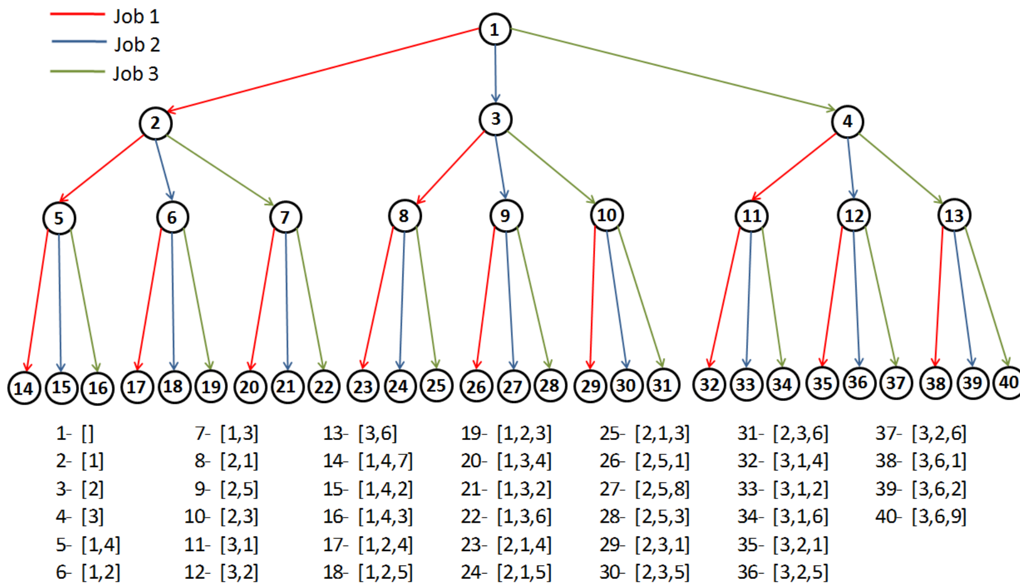


Figura 7: Árbol de búsqueda de $\mathcal{I}$ hasta la etapa 3

Así expuesto, este procedimiento resulta ser un algoritmo de fuerza bruta. Como cada secuencia s será expandida por una cantidad n de operaciones o (salvo en los casos en donde $\#(\varepsilon(q(s))) < n$, es decir, cuando $d(x) > nm - n$) la cantidad de secuencias $s' = s + o$ generadas es igual a n. Luego, si suponemos que $n = 10$ y $m = 10$, en la etapa número 10 tendremos una cantidad de secuencias $10^{10} = 10{,}000{,}000{,}000$, lo que resulta prácticamente intratable en terminos computacionales. Este hecho sólo comprueba el cáracter exponencial del problema.

Sin embargo, este mecanismo de expansión puede mejorarse significativamente aprovechando la estructura de secuencias que hemos definido. Como veremos a continuación, la solución óptima puede hallarse expandiendo cada secuencia s sólo con las operaciones o que dan como resultado una secuencia $s + o$ *ordenada*. Esto reducirá la cantidad de nodos en el árbol de búsqueda. Ilustramos esta situación con el árbol de búsqueda reducido para la instancia $\mathcal{I}$ en la Figura 8.

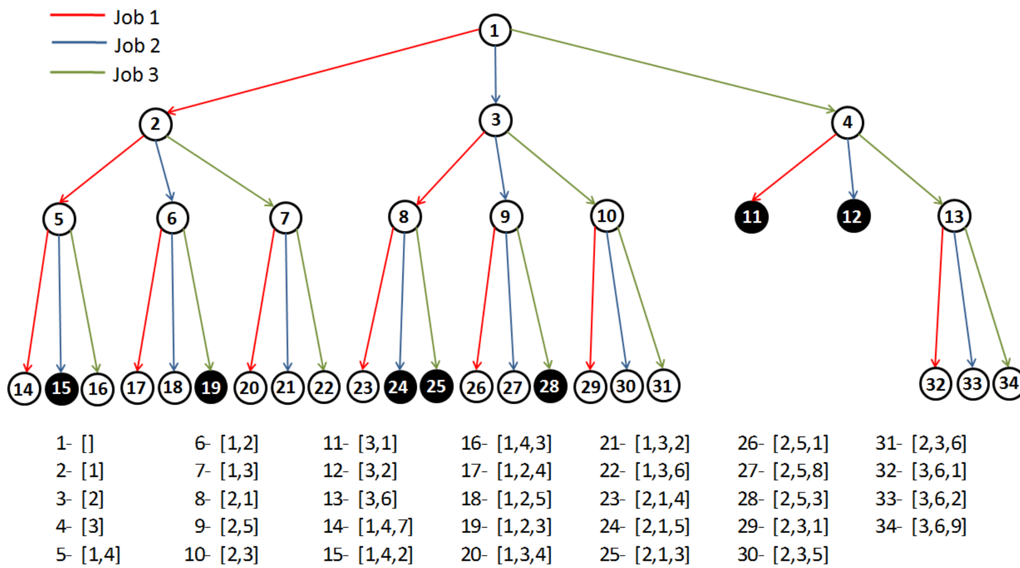


Figura 8: Árbol de búsqueda de $\mathcal{I}$ hasta la etapa 3 con expansión ordenada

No resulta díficil justificar que este mecanismo genera una solución óptima del problema. En efecto, si ψ_{opt} fuera un schedule óptimo para el JSSP, entonces $s_{\psi_{opt}}$ es la secuencia ordenada asociada. Como toda subsecuencia de una secuencia ordenada necesariamente es ordenada (pues de otra forma se obtendría una contradicción), entonces $s_{\psi_{opt}}$ estará en el conjunto final si generamos sólo secuencias que sean ordenadas. Luego, obtendremos la solución óptima ψ_{opt}.

En las siguientes subsecciones nos abocaremos a mejorar y depurar este mecanismo para poder obtener un algoritmo más aceptable que nos permita solucionar algunas instancias del JSSP.

4.3. El principio de optimalidad para el JSSP

4.3.1. Analogía con el TSP en un primer acercamiento

Nuestra representación del JSSP evoca la del problema de scheduling expuesto en la Sección 2 (Ver Problema 6), y por lo tanto hace suponer que, aprovechando esta analogía, podremos aplicar PD al JSSP tal como lo hicimos con el Problema 6, optimizando así la búsqueda de la solución exacta. Lamentablemente, el paralelismo entre ambos problemas no alcanza algunos puntos fundamentales del planteo clásico de PD. En particular, no es posible formular un principio de optimalidad para nuestra formulación del JSSP. A continuación recordamos brevemente el planteo por PD del Problema 6 para señalar las diferencias con el JSSP.

La ecuación funcional para el Problema 6 era:

$$\begin{cases} n(B) = 1 \Rightarrow \forall k : C(\{k\}) = c_k(p_k) \\ \\ n(B) > 1 \Rightarrow C(B) = \text{mín}_{k \in B}[C(B \backslash \{k\}) + c_k(p_k)] \end{cases}$$

En primera instancia, la adaptación parece inmediata, haciendo las siguientes redefiniciones:

1. $\mathring{S}(Q)$ tomaría el lugar de B, con $Q \subseteq \mathcal{O}$ factible.
2. $C(B)$ sería entonces $C(\mathring{S}(Q))$, y quedaría definido como

$$\min_{s \in \mathring{S}(Q)} \{C_{\text{máx}}(s)\}$$

3. Luego, $c_k(p_k)$ sería $c(s, o)$, y denotaría el costo de expandir la secuencia s con la operación o. Más formalmente, $c(s, o) = C_{\text{máx}}(s + o) - C_{\text{máx}}(s)$

Dentro de este contexto, podremos escribir la ecuación funcional relacionando los subconjuntos de la siguiente manera:

$$\begin{cases} |Q| = 1 \Rightarrow \forall k : C(\{o\}) = p_o \\ \\ |Q| > 1 \Rightarrow C(S(Q)) = \text{mín}_{s \in S(Q)}[C(S(Q \backslash \{o\})) + c(s, o)] \end{cases}$$

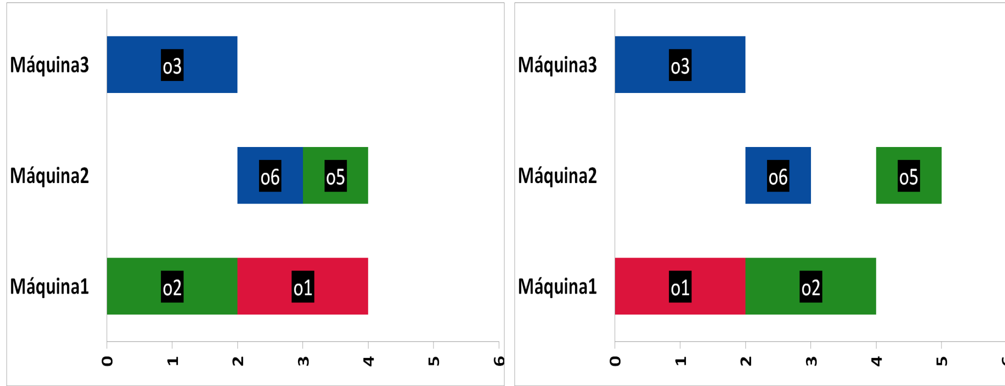


Figura 9: Schedule parcial que minimiza el $C_{\text{máx}}$ (izquierda) vs. Schedule parcial óptimo (derecha)

Si bien parece que la adaptación del problema resulta válida, falta verificar que la ecuación funcional respete el principio de optimalidad. Basta con ver un ejemplo para mostrar que esto no sucede.

En la Figura 9, se muestran dos schedules con el mismo subconjunto de operaciones $Q = \{o_1, o_2, o_3, o_5, o_6\}$. Mientras que el primero minimiza el $C_{\text{máx}}$ para todas las secuencias de Q, el segundo conduce al óptimo de la Figura 6, lo que demuestra que no alcanza con construir secuencias factibles cuyo tiempo parcial sea mínimo. Esencialmente: no es cierto que las subsecuencias de una secuencia completa óptima sean óptimas. Este hecho se debe a que las restricciones de factibilidad dificultan el problema, haciendolo más complejo.

Por el contrario, en el Problema 6 no tenemos restricciones de precedencia sobre el orden en que se deben programar las tareas, lo que hace que la ecuación funcional funcione a la perfección.

La conclusión es que un funcional que minimice el $C_{\text{máx}}$ de las secuencias parciales no sirve para el JSSP. Por esto, debemos buscar una forma de asegurar que el principio de optimalidad vuelva a tener validez.

Definición 4.16. *Sea $Q \subset \mathcal{O}$ factible, $s \in \mathring{S}(Q)$ secuencia parcial ordenada y $o \in \varepsilon(Q)$. Definiremos como $\xi : \mathring{S}(Q) \times \varepsilon(Q) \to \mathbb{N}$ a la siguiente función:*

$$\xi(s, o) = \begin{cases} \psi(s, o) + p(o), \text{si } o \in \eta(s) \\ C_{\text{máx}}(s) + p(o), \text{ en caso contrario} \end{cases}$$

Proposicion 4.7. *Sea $s \in \mathring{S}(Q)$ secuencia parcial ordenada y $o \in \varepsilon(Q)$. Son válidas:*

1. $\psi(s, o) \leq C_{\text{máx}}(s)$.

2. *Si* $o \in \eta(s) : C_{\text{máx}}(s+o) = \psi(s,o) + p(o)$.

3. $C_{\text{máx}}(s) \leq \xi(s,o)$.

4. $\xi(s,o) \leq C_{\text{máx}}(s) + p(o)$.

5. $\psi(s,o) \leq \xi(s,o)$.

6. *Si* $o \notin \eta(s), \forall \psi \in \mathring{Comp}_{\psi}(s) : \psi(o) \geq C_{\text{máx}}(s)$

Demostración. Consideraremos que $|Q| = k$. Además, de la Definición 4.15, tenemos que:

$$\psi(s,o) = \psi_{s+o}(o) = \begin{cases} 0 \quad si\ Pre(s+o,k+1) = \emptyset \\ \text{máx}_{o' \in Pre(s+o,k+1)}\{\psi_{s+o}(o') + p(o')\} \end{cases}$$

donde, de la Definición de 4.7, tenemos que

$$Pre(s+o,k+1) = \{s_{[x]} \in (\mathcal{J}(s_{[k+1]}) \cup \mathcal{M}(s_{[k+1]})) : x < k+1\} =$$
$$= \{o' \in Q \cap (\mathcal{J}(o) \cup \mathcal{M}(o))\}$$

ya que $o = (s+o)_{[k+1]}$ y $Q = q\big((s+o)_{[1,k]}\big)$. Con esto en mente, comenzaremos a demostrar los ítems.

1. Si $\{o' \in Q \cap (\mathcal{J}(o) \cup \mathcal{M}(o))\} \Rightarrow \psi(s,o) = 0 \leq C_{\text{máx}}(s)$. En caso contrario, como $\{o' \in Q \cap (\mathcal{J}(o) \cup \mathcal{M}(o))\} \subseteq Q$, por lo que:

$$\psi(s,o) = \max_{o' \in Q \cap (\mathcal{J}(o) \cup \mathcal{M}(o))}\{\psi(o') + p(o')\} \leq \max_{o' \in Q}\{\psi(o') + p(o')\}$$
$$= C_{\text{máx}}(s)$$

2. Como $o \in \eta(s)$, entonces $s+o$ es una secuencia ordenada. Además, como o es la última operación de $s+o$, por la Proposición 4.4, tenemos que

$$C_{\text{máx}}(s+o) = \psi_{s+o}(o) + p(o)$$

Finalmente, de que $\psi_{s+o}(o) = \psi(s,o)$, obtenemos que

$$C_{\text{máx}}(s+o) = \psi(s,o) + p(o)$$

como queríamos ver.

3. Tenemos dos posibles casos:

- Si $o \in \eta(s) \Rightarrow \xi(s,o) = \psi(s,o) + p(o)$.
Pero como $o \in \eta(s)$, por lo visto en 2 tenemos que $C_{\text{máx}}(s) \leq C_{\text{máx}}(s+o) = \psi(s,o) + p(o) = \xi(s,o)$
- Si $o \notin \eta(s) \Rightarrow \xi(s,o) = C_{\text{máx}}(s) + p(o)$. Resulta trivial que $C_{\text{máx}}(s) \leq C_{\text{máx}}(s) + p(o) = \xi(s,o)$.

En ambos casos se cumple la desigualdad.

4. Tenemos dos posibles casos:

- Si $o \in \eta(s) \Rightarrow \xi(s,o) = \psi(s,o) + p(o)$.
Por lo visto en 1, tenemos que $\psi(s,o) \leq C_{\text{máx}}(s) \Rightarrow \xi(s,o) = \psi(s,o) + p(o) \leq C_{\text{máx}}(s) + p(o)$
- Si $o \notin \eta(s) \Rightarrow \xi(s,o) = C_{\text{máx}}(s)+p(o)$. Resulta trivial que $\xi(s,o) \leq C_{\text{máx}}(s) + p(o)$.

En ambos casos se cumple la desigualdad.

5. Es consecuencia directa de 1 y 3:

$$\psi(s,o) \leq C_{\text{máx}}(s) \leq \xi(s,o)$$

6. Sea $\psi \in \mathring{Comp}_{\psi}(s)$, y sea $s^1 = s_{\psi} \in \mathring{Comp}(s)$ su secuencia asociada. Como $o \notin Q$ y $o \notin \eta(s)$, entonces $o = s^1_{[x+1]}$, con $x > k$.

Dado que la sucesión de $C_{\text{máx}}(s^1_{[1,y]})$ es creciente en y (Proposición 4.6), entonces, para que $o \in \eta(s^1_{[1,x]})$ debe ser que $\psi(s^1_{[1,x]}, o) > \psi(s,o)$. Luego, dado que $s^1_{[1,k]} = s$, se infiere que $\exists o' \in \big(Pre(s^1,x) \backslash Pre(s^1,k)\big)$, con $m(o') = m(o)$. Supongamos que $o' = s^1_{[y]}$, con $k < y < x$. Luego, tendremos que:

a) $\psi(o) = \psi(s^1_{[1,x]}, o) \geq \psi(o') + p(o')$
b) $C_{\text{máx}}(s^1_{[1,k]}) \leq C_{\text{máx}}(s^1_{[1,y]}) = \psi(o') + p(o')$

Por lo tanto, tendremos que:

$$C_{\text{máx}}(s) = C_{\text{máx}}(s^1_{[1,k]}) \leq \psi(o') + p(o') \leq \psi(o)$$

□

Notemos que el funcional $\xi(s, o)$ podría interpretarse como el presumible tiempo de finalización de la expansión de cada secuencia $s \in \mathring{S}(Q)$, con cada operación en $\varepsilon(Q)$. Si $o \in \eta(s)$, entonces $s + o$ es ordenada y $C_{\text{máx}}(s + o) = \xi(s, o)$. Si, por el contrario, $o \in (\varepsilon(Q) \setminus \eta(s))$, no expandiremos s con o, pues no resultaría ordenada. No obstante, si eventualmente, a través de expansiones sucesivas de s obtenemos la secuencia s' y $o \in \eta(s')$, entonces sabemos que el tiempo de finalización de $s' + o$ será mayor o igual a $\xi(s, o)$ (ver ítem 6 de la Proposición 4.7).

Por lo tanto, $\xi(s, o)$ nos ofrece una cota inferior al tiempo de finalización de la operación o para la secuencia s.

4.3.2. Secuencias comparables y dominadas

Si observamos el ejemplo de aplicación de programación dinámica sobre el Problema 6, vemos que se demuestra que para toda secuencia parcial de mínimo costo, todas sus subsecuencias también deben resultar mínimas. Si ahora tratamos de adaptar el mismo razonamiento al JSSP, necesitaremos encontrar un funcional que nos asegure que cualquier secuencia parcial que sea mínima bajo ese funcional, tenga todas sus subsecuencias parciales también mínimas. No obstante, dicho funcional debe cumplir también que bajo el espacio de secuencias completas el mínimo coincida con una solución óptima del problema.

Ahora bien, en el problema de scheduling mencionado, el principio de optimalidad resulta válido ya que todas las secuencias que tengan programadas las mismas táreas comparten el mismo conjunto de posibles completaciones. Luego, siempre la mejor opción se reduce a tomar la secuencia parcial de mínimo costo, pues cualquier completación de otra secuencia será dominada por la misma completación de la secuencia mínima.

Esto nos da una idea de cómo podríamos hallar el funcional adecuado en el caso del JSSP. En principio buscaremos alguna condición que nos asegure que cualquier completación de una secuencia s_1, será dominada por alguna completación de otra secuencia s_2.

Una condición así debería asegurar que todas las operaciones programadas en una completación $s^1 \in \mathring{Comp}(s_1)$ puedan ser programadas de la misma forma en $s^2 \in \mathring{Comp}(s_2)$. Más formalmente:

$$\forall o \in \mathcal{O} : \psi_{s^2}(o) + p(o) \leq \psi_{s^1}(o) + p(o)$$

Si bien lo dicho resulta muy intuitivo, la dificultad reside en hallar una expresión dependiente sólo de las secuencias parciales que certifique la validez de la condición. Veamos el siguiente ejemplo:

Ejemplo 4.2. *Supongamos que tenemos el subconjunto factible de operaciones* $Q = \{o_1, o_2, o_3, o_5, o_6\}$*, y los siguientes schedules parciales pertenecientes a* $s_1 = [o_2, o_3, o_6, o_1, o_5]$, $s_2 = [o_2, o_3, o_5, o_1, o_6]$ *respectivamente.*

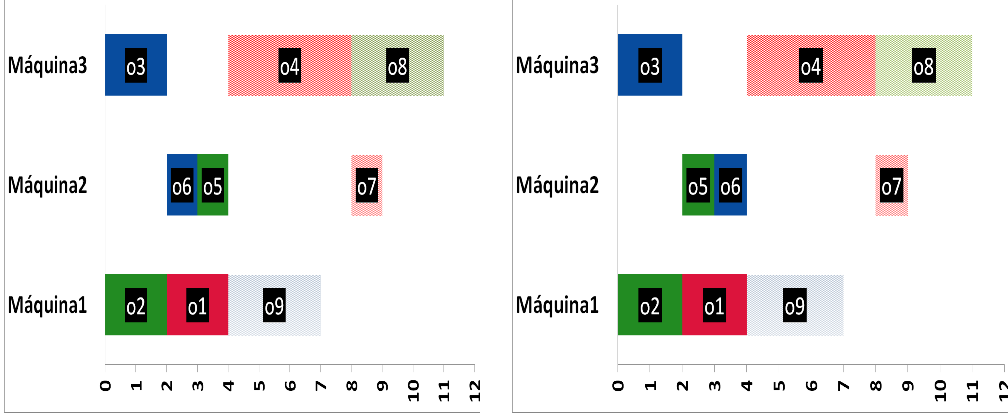


Figura 10: Schedule parcial dominado (izquierda) vs. Schedule parcial dominante (derecha)

En la Figura 10 se puede ver la expansión de s_1, s_2 *mediante la subsecuencia* $s_c = [o_9, o_4, o_7, o_8]$*, generando dos secuencias completas con el mismo* s^1, s^2 *tales que*

$$\forall o \in (O \backslash Q) : \psi_{s^2}(o) + p(o) \leq \psi_{s^1}(o) + p(o)$$

En el ejemplo anterior $s_1, s_2 \in \mathring{S}(Q)$, por lo que resta planificar el mismo conjunto de operaciones sobre ambas secuencias. También, resulta claro que $\forall o \in \varepsilon(Q) : \psi(s_2, o) \leq \psi(s_1, o)$. No obstante, esto no es determinante si nos ponemos a pensar que no siempre toda operación $o \in \eta(s_1)$, puede expandir s_2 ordenadamente (i.e: $o \in \eta(s_2)$). Observemos que en la Figura 10 sólo se muestra una posible expansión ordenada de ambas secuencias. Es por esto que, para poder distinguir estos casos de forma adecuada, utilizaremos el funcional ξ.

A continuación, probamos una serie de resultados que nos permitirán comparar y descartar secuencias, como las del Ejemplo 4.2, mediante la comparación entre $\xi(s_1, o)$ y $\xi(s_2, o)$.

Proposicion 4.8. *Sea* $Q \subseteq \mathcal{O}$ *factible, y* $s_1, s_2 \in \mathring{S}(Q)$ *secuencias parciales ordenadas.*

Si $\forall o \in \varepsilon(Q) : \xi(s_2, o) \leq \xi(s_1, o)$*, entonces valen:*

1. $\forall o \in \eta(s_1) \cap \eta(s_2)$*, vale que* $\psi(s_2, o) \leq \psi(s_1, o)$ *y* $C_{\text{máx}}(s_2 + o) \leq C_{\text{máx}}(s_1 + o)$*.*

2. *$\forall o \in \varepsilon(Q)$ tal que $o \in \eta(s_1)$ y $o \notin \eta(s_2)$, vale que $\psi(s_2, o) < \psi(s_1, o)$ y $C_{\text{máx}}(s_2 + o) < C_{\text{máx}}(s_1 + o)$.*

3. *Si $C_{\text{máx}}(s_1) < C_{\text{máx}}(s_2) \Rightarrow \forall o \in \varepsilon(Q) : o \in \eta(s_2)$*

Demostración. 1. Sea $o \in \eta(s_1) \cap \eta(s_2)$. Luego, tenemos que

$$\psi(s_2, o) + p(o) = \xi(s_2, o) \leq \xi(s_1, o) = \psi(s_1, o) + p(o) \tag{15}$$

Además, como $s_1 + o, s_2 + o$ son secuencias ordenadas, tenemos que:

$$\psi(s_i, o) + p(o) = C_{\text{máx}}(s_i + o),\ i = 1, 2 \tag{16}$$

Luego, de (15) y (16) se deduce que:

$$C_{\text{máx}}(s_2 + o) = \xi(s_2, o) \leq \xi(s_1, o) = C_{\text{máx}}(s_1 + o)$$

2. Por hipótesis tenemos que $o \in \eta(s_1) \Rightarrow \xi(s_1, o) = \psi(s_1, o) + p(o) = C_{\text{máx}}(s_1 + o)$

Por otro lado, $o \notin \eta(s_2) \Rightarrow \xi(s_2, o) = C_{\text{máx}}(s_2) + p(o)$

Luego, vale lo siguiente:

$$\begin{aligned} C_{\text{máx}}(s_2) + p(o) = \xi(s_2, o) \leq \xi(s_1, o) = \psi(s_1, o) + p(o) \Rightarrow \\ \Rightarrow C_{\text{máx}}(s_2) \leq \psi(s_1, o) \end{aligned} \tag{17}$$

Además, como $o \notin \eta(s_2)$, entonces:

$$\psi(s_2, o) + p(o) \leq C_{\text{máx}}(s_2) \Rightarrow \psi(s_2, o) < C_{\text{máx}}(s_2) \tag{18}$$

Sumado a esto, tenemos que

$$\begin{aligned} C_{\text{máx}}(s_2 + o) &= \max_{o' \in Q \cup \{o\}} \{\psi_{s_2}(o') + p(o')\} \\ &= \max\left(\max_{o' \in Q}\{\psi_{s_2}(o') + p(o')\}, \psi(s_2, o) + p(o)\right) = \\ &= \max\Big(C_{\text{máx}}(s_2), \psi(s_2, o) + p(o)\Big) = C_{\text{máx}}(s_2) \end{aligned} \tag{19}$$

De (17) y (18) se deduce que

$$\psi(s_2, o) < C_{\text{máx}}(s_2) \leq \psi(s_1, o)$$

mientras que de (17) y (19) se infiere:

$$C_{\text{máx}}(s_2 + o) = C_{\text{máx}}(s_2) \leq \psi(s_1, o) < \psi(s_1, o) + p(o) = C_{\text{máx}}(s_1 + o)$$

con lo que queda probado el resultado.

3. Supongamos que $\exists o \in \big(\varepsilon(Q)\backslash\eta(s_2)\big)$. Entonces

$$\xi(s_2, o) = C_{\text{máx}}(s_2) + p(o) > C_{\text{máx}}(s_1) + p(o) \tag{20}$$

Por otro lado, del cuarto ítem de la Proposición 4.7, tenemos que

$$\xi(s_1, o) \leq C_{\text{máx}}(s_1) + p(o) \tag{21}$$

De (20) y (21) se infiere que

$$\xi(s_2, o) > C_{\text{máx}}(s_1) + p(o) \geq \xi(s_1, o)$$

lo cual contradice el hecho de que $\forall o' \in \varepsilon(Q) : \xi(s_2, o') \leq \xi(s_1, o')$. Luego, se deduce que $\forall o' \in \varepsilon(Q) : o' \in \eta(s_2)$.

□

Definición 4.17. *Sea $s^1 \in \mathring{S}(\mathcal{O})$ una secuencia ordenada y completa. Sea $k \in \{1, \cdots, nm-1\}$, $s_1 = s^1_{[1,k]}$ y sea $s_2 \in \mathring{S}(q(s_1))$.*

Si ahora completamos s_2 con la secuencia $s^1_{[k+1,nm]}$, obtendremos una secuencia completa diferente a s^1, a la que llamaremos $\bar{s}^2$. A su vez, como dicha completación puede ser desordenada, denominaremos s^2 a la secuencia resultante de ordenar $\bar{s}^2$.

Así, definiremos el operador $\mu(s^1, k, s_2)$ como

$$\mu(s^1, k, s_2) := s^2$$

Básicamente, $\mu(s^1, k, s_2)$ sintetiza el proceso explicado anteriormente. Más formalmente:

1. $\bar{s}^2 = s_2 \oplus s^1_{[k+1,nm]}$

2. $s^2 = s_{\psi_{\bar{s}^2}}$

3. $\mu(s^1, k, s_2) = s^2$

El proceso descripto por el operador μ será utilizado extensivamente en lo sucesivo.

Corolario 4.1. *Sean $k = |Q|$ y $s_1, s_2 \in \mathring{S}(Q)$, secuencias parciales ordenadas que satisfacen: $\forall o \in \varepsilon(Q) : \xi(s_2, o) \leq \xi(s_1, o)$. Sea $s^1 \in Co\mathring{m}p(s_1)$. Entonces, $\forall x \geq k$ y $s^2 = \mu(s^1, k, s_2) \in \mathring{S}(\mathcal{O})$ valen:*

- $\psi_{s^2}(s^1_{[x+1]}) \leq \psi_{s^1}(s^1_{[x+1]})$
- $C_{\text{máx}}(s^2_{[1,x+1]}) \leq C_{\text{máx}}(s^1_{[1,x+1]})$

En particular, cuando $x = nm - 1$ tendremos que $C_{\text{máx}}(s^2) \leq C_{\text{máx}}(s^1)$.

Demostración. Sea $s^1 \in \overset{\circ}{Comp}(s_1)$ ordenada. Sea $\bar{s}^2 = s_2 \oplus s^1_{[k+1,nm]}$ la expansión de s_2 no ordenada.

Sea $x = k, k+1, \cdots, nm-1$. Denotaremos como $o^{x+1} = s^1_{[x+1]}$. En principio, veremos que si $o^{x+1} \in \eta(s^1_{[1,x]})$, entonces valen:

1. $\psi(\bar{s}^2_{[1,x]}, o^{x+1}) \leq \psi(s^1_{[1,x]}, o^{x+1})$
2. $C_{\text{máx}}(\bar{s}^2_{[1,x]} + o^{x+1}) \leq C_{\text{máx}}(s^1_{[1,x]} + o^{x+1})$

Haremos la demostración por inducción en x.

- CASO BASE: $x = k$

  Por la Proposición 4.8, tenemos que $\forall o \in \eta(s_1) : \psi(s_2, o) \leq \psi(s_1, o)$ y $C_{\text{máx}}(s_2 + o) \leq C_{\text{máx}}(s_1 + o)$. Luego, tomando $o = o^{x+1}$ obtenemos que valen 1) y 2).

- PASO INDUCTIVO: $k < x$

  Sea $o^{x+1} \in \eta(s^1_{[1,x]})$.

  1. Por definición, tenemos que:

     $$\psi(s,o) = \begin{cases} 0 \; si \; Pre(\psi_s, o) = \emptyset \\ \text{máx}_{o \in Pre(\psi_s,o)}\{\psi_s(o) + p(o)\} \; caso \; contrario \end{cases}$$

     En este contexto, como $q(s^1_{[1,x]}) = q(\bar{s}^2_{[1,x]})$, entonces tenemos que $Pre(\psi_{s^1_{[1,x]}}, o^{x+1}) = Pre(\psi_{\bar{s}^2_{[1,x]}}, o^{x+1})$. Luego, notaremos simplemente como $q(s_{[1,x]})$ y $Pre(\psi_{s_{[1,x]}}, o^{x+1})$ a los conjuntos anteriores. Tendremos dos casos:

     a) $Pre(\psi_{s_{[1,x]}}, o^{x+1}) = \emptyset$. Luego,

     $$\psi(s^1_{[1,x]}, o^{x+1}) = 0 = \psi(\bar{s}^2_{[1,x]}, o^{x+1})$$

     En particular, vale que $\psi(\bar{s}^2_{[1,x]}, o^{x+1}) \leq \psi(s^1_{[1,y]}, o^{x+1})$

     b) $Pre(\psi_{s_{[1,x]}}, o^{x+1}) \neq \emptyset$. Sea $o \in q(s_{[1,x]})$ tal que

     $$o = \underset{o' \in Pre(\psi_{s^1_{[1,x]}}, o^{x+1})}{\arg\max} \{\psi_{s^1_{[1,x]}}(o') + p(o')\}$$

     En caso de haber más de una operación o que cumpla lo dicho, tomaremos como o la que tenga la mayor posición dentro de la secuencia s^1.

     Dividiremos la demostración en los siguientes casos:

- $o \in Q$:
  Como $o \in Q$, $\nexists o' \in Pre(\psi_{s^1_{[1,x]}}, o^{x+1})$ tal que $o' = s_{[y]}$, con $y > k$ (i.e. $o' \notin Q$). En efecto, si existiese o', dado que s^1 es ordenada, entonces tendremos que $o <_{\psi_{s^1}} o'$, de donde se infiere que $o \neq \arg\max_{o' \in Pre(\psi_{s^1_{[1,x]}}, o^{x+1})}\{\psi_{s^1_{[1,x]}}(o') + p(o')\}$, lo que es absurdo.
  De lo anterior se obtiene que $o^{x+1} \in \varepsilon(Q)$ y

  $$Pre(\psi_{s^1_{[1,x]}}, o^{x+1}) = Pre(\psi_{s^1_{[1,k]}}, o^{x+1}) = Pre(\psi_{s_{[1,k]}}, o^{x+1}).$$

  Por lo tanto, tenemos que $\psi(s^1_{[1,x]}, o^{x+1}) = \psi(s^1_{[1,k]}, o^{x+1})$ y $\psi(\bar{s}^2_{[1,x]}, o^{x+1}) = \psi(\bar{s}^2_{[1,k]}, o^{x+1})$ Por otro lado, dado que $o^{x+1} \in \eta(s^1_{[1,x]})$, entonces

  $$\psi(s^1_{[1,x]}, o^{x+1}) + p(o^{x+1}) \geq C_{\text{máx}}(s^1_{[1,x]}) \geq C_{\text{máx}}(s^1_{[1,k]}).$$

  Además, si

  $$\psi(s^1_{[1,x]}, o^{x+1}) + p(o^{x+1}) = C_{\text{máx}}(s^1_{[1,k]}),$$

  entonces

  $$\psi(s^1_{[1,x]}, o^{x+1}) + p(o^{x+1}) = C_{\text{máx}}(s^1_{[1,x]})$$

  y $C_{\text{máx}}(s^1_{[1,x]}) = C_{\text{máx}}(s^1_{[1,k]})$, de donde necesariamente vale que $m(o^{x+1}) > m(s^1_{[x]}) > m(s^1_{[k]})$. Luego, se deduce que $o^{x+1} \in \eta(s^1_{[1,k]})$.
  Finalmente, utilizando la Proposición 4.8, y teniendo en cuenta que $s_1 = s^1_{[1,k]}$ y $s_2 = \bar{s}^2_{[1,k]}$

  $$\psi(\bar{s}^2_{[1,x]}, o^{x+1}) = \psi(\bar{s}^2_{[1,k]}, o^{x+1}) = \psi(s_2, o^{x+1}) \leq$$

  $$\leq \psi(s_1, o^{x+1}) = \psi(s^1_{[1,k]}, o^{x+1}) = \psi(s^1_{[1,x]}, o^{x+1})$$

- $o \in \big(q(s_{[1,x]}) \backslash Q\big)$:
  En este caso, por H.I., tendremos que $\psi_{\bar{s}^2_{[1,x]}}(o) \leq \psi_{s^1_{[1,x]}}(o)$ implica $\psi_{\bar{s}^2_{[1,x]}}(o) + p(o) \leq \psi_{s^1_{[1,x]}}(o) + p(o)$. Luego,

  $$\begin{aligned}\psi(\bar{s}^2_{[1,x]}, o^{x+1}) &= \psi_{\bar{s}^2_{[1,x]}}(o) + p(o) \leq \psi_{s^1_{[1,x]}}(o) + p(o)\\ &= \psi(s^1_{[1,x]}, o^{x+1})\end{aligned}$$

En cualquier caso, obtendremos que

$$\psi(\bar{s}^2_{[1,x]}, o^{x+1}) \leq \psi(s^1_{[1,x]}, o^{x+1}),$$

como queríamos ver.

2. Como $o^{x+1} \in \eta(s^1_{[1,x]})$, entonces tendremos que

$$C_{\text{máx}}(s^1_{[1,x]} + o^{x+1}) = \psi(s^1_{[1,x]}, o^{x+1}) + p(o^{x+1})$$

Por lo visto anteriormente,

$$\psi(\bar{s}^2_{[1,x]}, o^{x+1}) + p(o^{x+1}) \leq \psi(s^1_{[1,x]}, o^{x+1}) + p(o^{x+1}),$$

por lo que

$$\psi(\bar{s}^2_{[1,x]}, o^{x+1}) + p(o^{x+1}) \leq C_{\text{máx}}(s^1_{[1,x]} + o^{x+1})$$

Además, por H.I., tendremos que

$$C_{\text{máx}}(\bar{s}^2_{[1,x]}) \leq C_{\text{máx}}(s^1_{[1,x]}) \leq C_{\text{máx}}(s^1_{[1,x]} + o^{x+1}).$$

Luego

$$C_{\text{máx}}(\bar{s}^2_{[1,x]} + o^{x+1}) = \max_{o \in (q(\bar{s}^2_{[1,x]}) \cup \{o^{x+1}\})} \{\psi(\bar{s}^2_{[1,x]}, o) + p(o)\} =$$

$$\text{máx}\Big(\max_{o \in q(\bar{s}^2_{[1,x]})} \{\psi(\bar{s}^2_{[1,x]}, o) + p(o)\}; \psi(\bar{s}^2_{[1,x]}, o^{x+1}) + p(o^{x+1})\Big)$$

$$\leq C_{\text{máx}}(s^1_{[1,x]} + o^{x+1})$$

como queríamos ver.

Por último, observemos que $s^2 = s_{(\psi_{\bar{s}^2})}$, es decir, s^2 es $\bar{s}^2$ ordenada. Como ordenar una secuencia no modifica los tiempos de inicio de las operaciones, entonces tendremos que

$$\forall o \in \mathcal{O} : \psi_{\bar{s}^2}(o) = \psi_{s^2}(o)$$

por lo que los resultados anteriores sigan siendo válidos entre las secuencias s^1 y s^2.

□

Haremos algunas observaciones importantes acerca de lo anterior.

La primera observación es que los resultados anteriores sólo pueden aplicarse a secuencias que tengan programadas las mismas operaciones, es decir: $q(s_1) = q(s_2)$. Además, se requiere $\forall o \in \varepsilon(Q) : \xi(s_2, o) \leq \xi(s_1, o)$, lo que es una condición bastante fuerte. Basta con que $\exists o_1, o_2 \in \varepsilon(Q) : \xi(s_1, o_1) < \xi(s_2, o_1) \wedge \xi(s_2, o_2) < \xi(s_1, o_2)$ para que los resultados dejen de tener aplicación.

La segunda observación es que, gracias al Corolario 4.1 si $\forall o \in \varepsilon(Q) : \xi(s_2, o) \leq \xi(s_1, o)$, entonces podremos descartar s_1. En efecto, el corolario nos indica que para toda completación ordenada s^1 de s_1, existirá una secuencia ordenada s^2 con un $C_{\text{máx}}$ mejor o igual que el de s^1. Por lo tanto, si consideramos el conjunto de secuencias ordenadas y completas $\mathring{S}(\mathcal{O})$, podremos encontrar una solución mejor que cualquier completación de s_1. El Ejemplo 4.2 expone este hecho para una sola completación de s_1, s_2, en donde $s^1 = s_1 + [o_9, o_4, o_7, o_8]$ y $s^2 = s_2 + [o_9, o_4, o_7, o_8]$.

Por último, en función de los resultados obtenidos, se podría sugerir intuitivamente que el funcional ξ es hereditario respecto de la expansión de secuencias. Lamentablemente, esto no es cierto, como se muestra en el próximo ejemplo:

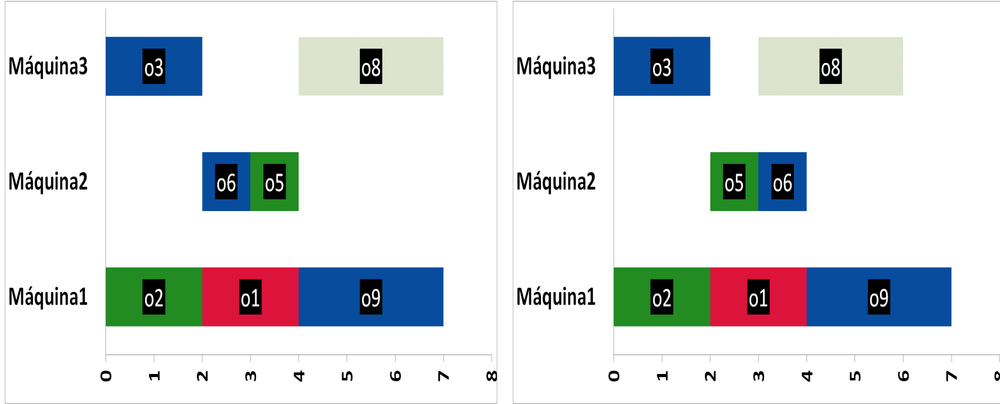


Figura 11: Schedule parcial dominante (izquierda) vs. Schedule parcial dominado (derecha)

Ejemplo 4.3. *El schedule de la izquierda corresponde a* $s'_1 = s_1 + [o_9]$ *y el de la derecha a* $s'_2 = s_2 + [o_9]$*, con* s_1, s_2 *las secuencias parciales del Ejemplo 4.2. En aquel ejemplo, teníamos que*

$$\forall o \in \varepsilon(Q) : \xi(s_2, o) \leq \xi(s_1, o)$$

En el ejemplo de la Figura 11 se ve que $o_8 \in \eta(s'_1)$, pero $o_8 \notin \eta(s'_2)$, por lo que tendremos

$$\xi(s'_1, o_8) = \psi(s'_1, o_8) + p(o_8) = 7 < 10 = C_{\text{máx}}(s_2) + p(o_8) = \xi(s'_2, o_8)$$

Esto resulta anti-intuivo con respecto al resultado obtenido en el Corolario 4.1. Por un lado, dado que s_1, s_2 están dentro de las hipótesis del corolario 4.1, entonces para cualquier completación ordenada de s^1, existe una completación de s^2 ordenada que es *mejor*. Pero, por otro lado, si tomamos s'_1, s'_2 como en el Ejemplo 4.3, tendríamos exactamente lo inverso: cualquier completación ordenada de s'_2 sería *peor* que una posible completación ordenada de s'_1. Luego, ¿De dónde surge este el conflicto?

En realidad, no existe tal conflicto, ya que estamos obviando el hecho de que, en la demostración del Corolario 4.1, la completación de s'_2 puede ser desordenada. Si miramos con atención la demostración de dicho corolario, veremos que lo que se hereda es el hecho de que la operación $s^1_{[y+1]}$ siempre se puede programar *más tempranamente* en s^2 que en s^1, pero no necesariamente de forma ordenada.

En efecto, la secuencia completa ordenada que dominará a la completación ordenada de s'_1, $s^1 = [o_2, o_3, o_6, o_1, o_5, o_9, o_8, o_4, o_7]$, será $s^2 = [o_2, o_3, o_5, o_1, o_6, o_8, o_9, o_4, o_7]$, como se ve en la Figura 12. Esta secuencia no pertenece al conjunto $\mathring{Comp}(s'_2)$, dado que $s^2_{[1,d(s'_2)]} \neq s'_2$. Por lo tanto, no es producto de una expansión ordenada de s'_2, sino de la secuencia ordenada $s'_3 = [o_2, o_3, o_5, o_1, o_6, o_8]$ que **no** es comparable con s'_1. Es por esto que s'_2 es realmente la secuencia parcial comparable con s'_1 que nos permite justificar su eliminación.

En conclusión, el Corolario 4.1 nos indica que cualquiera sea la completación de s'_1, existirá una secuencia completa y ordenada s^2 que resulte mejor, pero que no necesariamente sea producto de una expansión ordenada de s'_2. Retomaremos este ejemplo más adelante cuando expliquemos *dominancia directa* y *dominancia indirecta*.

Ya estamos en condiciones de establecer el criterio mediante el cuál se compararán dos secuencias parciales para decidir si alguna puede ser eliminada. Ahora bien, ¿qué pasaría si $\forall o \in \varepsilon(Q) : \xi(s_1, o) = \xi(s_2, o)$?

En este caso simplemente ambas secuencias se dominan mutuamente, por lo que cualquiera es candidata a ser descartada. Dado que resultaría inútil expandir ambas secuencias, simplemente se eliminará alguna de las dos. Para esto, basta con definir alguna regla básica de elección.

Dadas dos secuencias $s_1, s_2 \in \mathring{S}(Q)$ tales que $s_1 \neq s_2$, si x es la mínima posición tal que $s_{1[x]} \neq s_{2[x]}$, entonces elegiremos la secuencia s_i de forma que

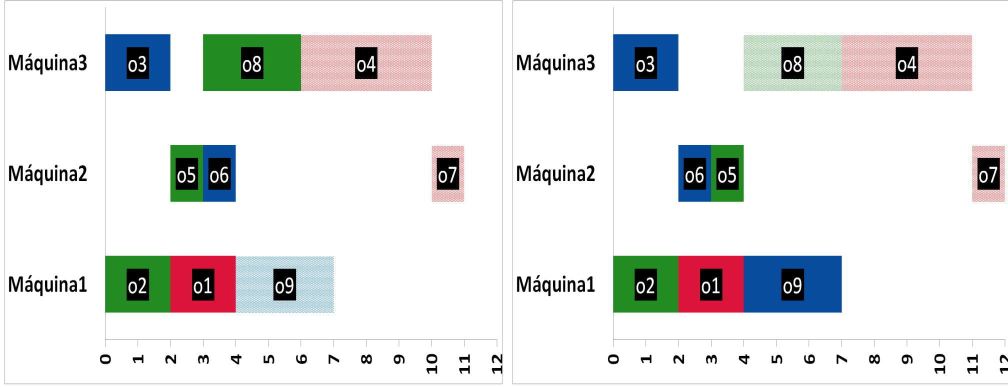


Figura 12: Schedule completo dominante (izquierda) Schedule completo dominado (derecha)

$indice(s_{i[x]})$ sea el menor.

Para ejemplificar, si $s_1 = [o_1, o_3, o_2]$ y $s_2 = [o_1, o_2, o_3]$, entonces elegiremos s_2, dado que la primera operación en donde las secuencias se diferencian es en la segunda, y $indice(s_{2[2]}) = indice(o_2) = 2 < 3 = indice(o_3) = indice(s_{1[2]})$. Notaremos este hecho como $s_2 \lessdot s_1$.

No resulta díficil demostrar que $\lessdot$ constituye una relación de orden total sobre las secuencias en $\mathring{S}(Q)$. Por lo tanto, la anterior regla nos servirá para elegir una única secuencia en el caso de que $\forall o \in \varepsilon(Q) : \xi(s_1, o) = \xi(s_2, o)$, cualquiera sea la circunstancia.

En relación a todo lo dicho hasta aquí, introduciremos el concepto de *dominancia* entre secuencias mediante la siguiente cadena de resultados y definiciones:

Definición 4.18. *Sea $Q \subseteq \mathcal{O}$ factible y $s_1, s_2 \in \mathring{S}(Q)$ dos secuencias ordenadas. Si $Q = \mathcal{O}$, entonces $s_2 \preceq s_1$ si se cumple alguna de las siguientes condiciones:*

$$\begin{cases} C_{\text{máx}}(s_2) < C_{\text{máx}}(s_1) \\ \\ C_{\text{máx}}(s_2) = C_{\text{máx}}(s_1) \wedge s_2 \lessdot s_1 \end{cases}$$

En caso de que $Q \subset \mathcal{O}$, entonces $s_2 \preceq s_1$ si se cumple alguna de las siguientes condiciones:

$$\begin{cases} \forall o \in \varepsilon(Q) : \xi(s_2, o) \leq \xi(s_1, o) \wedge \exists o' \in \varepsilon(Q) : \xi(s_2, o) < \xi(s_1, o) \\ \\ \forall o \in \varepsilon(Q) : \xi(s_2, o) = \xi(s_1, o) \wedge s_2 \lessdot s_1 \end{cases}$$

En ambos casos diremos que s_2 domina a s_1, *o que* s_1 es dominada por s_2.

Proposicion 4.9. *Sea* $Q \subseteq \mathcal{O}$ *factible. Sean* $s_1, s_2, s_3 \in \mathring{S}(Q)$. *Si* $s_3 \preceq s_2, s_2 \preceq s_1$, *entonces* $s_3 \preceq s_1$. *Es decir, la relación establecida por* $\preceq$ *es transitiva.*

Demostración. Si $Q = \mathcal{O}$, entonces, de la Definición 4.18 se infiere que $(C_{\text{máx}}(s_3) \leq C_{\text{máx}}(s_2)) \wedge (C_{\text{máx}}(s_2) \leq C_{\text{máx}}(s_1)) \Rightarrow C_{\text{máx}}(s_3) \leq C_{\text{máx}}(s_1)$. Luego, $(C_{\text{máx}}(s_3) < C_{\text{máx}}(s_1)) \vee (C_{\text{máx}}(s_3) = C_{\text{máx}}(s_1))$. En el primer caso, se infiere directamente que $s_3 \preceq s_1$.

En el segundo caso, observemos que entonces tendremos que $C_{\text{máx}}(s_3) = C_{\text{máx}}(s_2) = C_{\text{máx}}(s_1)$, por lo que, necesariamente tendremos que $(s_3 \lessdot s_2) \wedge (s_2 \lessdot s_1)$. Como $\lessdot$ es transitiva, se tiene que $s_3 \lessdot s_1$, de donde se deduce nuevamente que $s_3 \preceq s_1$.

En el caso de que $Q \neq \mathcal{O}$, entonces tendremos que, $\forall o \in \varepsilon(Q) : \big(\xi(s_3, o) \leq \xi(s_2, o)\big) \wedge \big(\xi(s_2, o) \leq \xi(s_1, o)\big) \Rightarrow \xi(s_3, o) \leq \xi(s_1, o)$. Luego, tendremos que $\big(\exists o' \in \varepsilon(Q) : \xi(s_3, o) < \xi(s_1, o)\big) \vee \big(\forall o \in \varepsilon : \xi(s_3, o) = \xi(s_1, o)\big)$. En el primer caso, por definición tendremos que $s_3 \preceq s_1$.

En el segundo caso, de forma análoga al caso correpondiente a $Q = \mathcal{O}$ y $C_{\text{máx}}(s_3) = C_{\text{máx}}(s_1)$, obtendremos también que $s_3 \preceq s_1$.

□

La Proposición 4.9 resulta de vital importancia, ya que nos da la certeza de que al eliminar secuencias dominadas, podremos eliminarlas en cualquier orden sin alterar el resultado final. Además, evita que se produzcan ciclos de dominancia, lo cual constituiría un problema teórico insalvable.

Dado que de la Proposición 4.9 también se puede inferir una relación de orden entre las secuencias de $\mathring{S}(Q)$, llamaremos *secuencias minimales* a las secuencias dominantes de $\mathring{S}(Q)$. Es decir, una secuencia s es minimal si $\nexists s' \in \mathring{S}(Q) : s' \neq s \wedge s' \preceq s$.

Observación 4.4. *Si* $Q \neq \mathcal{O}$, *notemos que puede haber más de una secuencia minimal. Nuevamente, si* $\exists o_1, o_2 \in \varepsilon(Q) : \xi(s_1, o_1) < \xi(s_2, o_1) \wedge \xi(s_2, o_2) < \xi(s_1, o_2)$, *entonces* $s_1 \npreceq s_2 \wedge s_2 \npreceq s_1$.

En base a lo dicho, definiremos los siguientes conjuntos:

Definición 4.19. *Sea* $Q \subseteq \mathcal{O}$ *factible, y* $X(Q) \subseteq \mathring{S}(Q)$ *un subconjunto de secuencias ordenadas de* Q. *Entonces*

$$\overset{(\preceq)}{X}(Q) := \{s \in X(Q) : \nexists s' \in \mathring{S}(Q), s' \neq s : s' \preceq s\}$$

La noción de dominancia es la clave para el desarrollo del algoritmo propuesto en [6]. Esquemáticamente, el algoritmo consistirá en:

1. Inicialmente, construir todos los conjuntos de secuencias de una sola operación
$$X(\{o\}) = \{(o)\}, \forall o \in \epsilon(\emptyset).$$
Fijar $i = 1$, el número de etapa.

2. Expandir todas las secuencias existentes, produciendo secuencias de cardinal $i+1$. Esto construye conjuntos de secuencias $X(Q)$ para todo Q de cardinal $i+1$.

3. Analizar las relaciones de dominancia dentro de cada $X(Q)$ y eliminar las secuencias dominadas, obteniendo $\overset{(\preceq)}{X}(Q)$. Fijar $i = i+1$ y retornar al punto 2.

El descarte de secuencias dominadas reduce enormemente el espacio de búsqueda del algoritmo. Sin embargo, entraña también una dificultad teórica importante: ¿cómo garantizar que el algoritmo encuentra una solución óptima? En principio, sería posible que todas las soluciones óptimas resultaran descartadas. Cuando una secuenca parcial s_1 es dominada por otra s_2, la primera se descarta, porque sabemos que para cualquier posible completación s^1 de s_1, existirá una solución s^2 con igual o mejor $C_{\text{máx}}$. Si esta solución surgiese de una completación ordenada de s_2, tendríamos garantizado el buen funcionamiento del algoritmo, dado que s_2 es preservada y expandida (en tanto no sea dominada por otra secuencia). Sin embargo, hemos visto que s^2 *no tiene por qué* surgir de una completación de s_2, sino que puede corresponder a la completación ordenada de una tercer secuencia s_3 que no sea comparable con s_1. Esto abre la posibilidad de que s_3 sea también descartada haciendo que la solución s^2 nunca se genere. En consecuencia, los elementos con los que contamos hasta aquí no nos permiten garantizar que las soluciones óptimas de una instancia no puedan dominarse unas a las otras en distintas etapas, haciendo que el algoritmo las descarte a todas, hallando, por lo tanto, una solución no-óptima.

Los siguientes apartados están dedicados a probar que esta situación no puede darse, lo cual nos permite concluir que el algoritmo es exacto, es decir: que encuentra un óptimo.

Es importante reiterar que la demostración aquí expuesta difiere de la proporcionada en [6], que contenía algunas afirmaciones erróneas.

4.4. Dominancia directa, indirecta y cadenas de dominancia

En esta subsección incorporaremos algunas definiciones y resultados adicionales que serán de vital importancia para demostrar el resultado principal de este capítulo.

Observando el Ejemplo 4.3 y los comentarios adicionales, vemos que la secuencia s^2 completa que justifica la eliminación de s'_1 se puede generar a través de s'_2, pero de forma desordenada. En particular, la operación o_8 se *adelanta* en 1 unidad de tiempo, haciendo que la secuencia ordenada s'_3 correspondiente a la misma etapa sea diferente a s'_2. Esto, nos sugiere la siguiente definición.

Definición 4.20. *Sea $s^1 \in \mathring{S}(\mathcal{O})$ una secuencia ordenada y completa. Sea $k \in \{1, \cdots, nm-1\}$, $s_1 = s^1_{[1,k]}$ y sea $s_2 \in \mathring{S}(q(s_1))$, tal que $s_2 \preceq s_1$. Sea $s^2 = \mu(s^1, k, s_2)$ según la Definición 4.17.*

Diremos que s^2 domina directamente *a s^1 en la etapa k si $s^2_{[1,k]} = s_2$. En caso contrario diremos que s^2* domina indirectamente *a s^1.*

En ocasiones (como en los párrafos siguientes) abusaremos ligeramente de la notación diciendo que s_2 domina directa o indirectamente a s_1. Sin embargo, debe tenerse en cuenta que estas nociones dependen fuertemente de la completación particular de las secuencias que se esté considerando. Así, es posible que la secuencia parcial s_2 domine a la secuencia s_1, y que esta dominancia sea directa para una cierta completación s^1 de s_1 e indirecta para otra completación s^{1*}.

Los Ejemplos 4.2 y 4.3 nos sirven para ilustrar ambos tipos de dominancia. Por ejemplo, sea $s^1 = [o_2, o_3, o_6, o_1, o_5, o_9, o_8, o_4, o_7]$. En el caso de que $k = 5$, obtendremos la subsecuencia s_1 del Ejemplo 4.2. Aquí, s_2 *domina directamente* a s_1, dado que $s^2 = \mu(s^1, k, s_2) = [o_2, o_3, o_5, o_1, o_6, o_8, o_9, o_4, o_7]$, y $s_2 = s^2_{[1,k]}$.

Por el contrario, si tomamos $k = 6$, obtendremos s'_1, que en principio es dominada por s'_2. En este caso, s^2 es la misma que antes, pero $s'_2 \neq s^2_{[1,k]}$. Por esto, la dominancia es indirecta.

De esta forma, la dominancia directa se corresponderá con los casos en que la secuencia completa dominante se pueda generar ordenadamente a partir de la secuencia parcial dominante. La dominancia indirecta indicará lo contrario.

Proposicion 4.10. *Sea $s^1 \in \mathring{S}(\mathcal{O})$, y sea $k \in \{1, \cdots, nm-1\} : s^1_{[1,k]}$ es dominada por $s_2 \in \mathring{S}(q(s^1_{[1,k]}))$. Si $\forall o \in \varepsilon(q(s^1_{[1,k]})) : o \in \eta(s_2)$, entonces la dominancia resulta directa.*

Demostración. Sea $o \in \varepsilon(q(s_2))$. Luego, por hipótesis, tenemos que $o \in \eta(s_2)$, por lo que se cumple alguna de las siguientes dos condiciones:

- $C_{\text{máx}}(s_2) < \psi(s_2, o) + p(o)$
- $C_{\text{máx}}(s_2) = \psi(s_2, o) + p(o) \wedge m(s_{2[k]}) < m(o)$

Sea $s^2 = \mu(s^1, k, s_2)$, y sea $\bar{s}^2 = s_2 \oplus s^1_{[k+1,nm]}$ la secuencia expandida sin ordenar. Sea x tal que $\bar{s}^2_{[x]} = o$. Como $o \notin q(s_2)$, entonces $x > k$.

Demostraremos que dada una operación $o \notin q(s_2)$, o ocupa una posición mayor que k en s^2. Dividiremos la demostración en dos casos:

1. $o \in \varepsilon(q(s_2))$
2. $o \notin \varepsilon(q(s_2))$

1. Como $o \in \varepsilon(q(s_2))$, entonces $o \in \eta(s_2)$. Luego, como $x > k$, entonces $q(s_2) \subseteq q(\bar{s}^2_{[1,x]})$. De donde se sigue:

$$\psi(s_2, o) + p(o) \leq \psi_{\bar{s}^2_{[1,x]}}(o) + p(o) = \psi_{\bar{s}^2}(o) + p(o)$$

   Además como reordenar la secuencia no altera el tiempo de inicio de las operaciones, tendremos que

$$\psi(s_2, o) + p(o) \leq \psi_{\bar{s}^2}(o) + p(o) = \psi_{s^2}(o) + p(o)$$

   Por lo tanto, vale alguna de las siguientes dos condiciones:

   - $C_{\text{máx}}(s_2) < \psi(s_2, o) + p(o) \leq \psi_{s^2}(o) + p(o)$
   - $C_{\text{máx}}(s_2) = \psi(s_2, o) + p(o) \leq \psi_{s^2}(o) + p(o) \wedge m(s_{2[k]}) < m(o)$

   En cualquier caso, tendremos que $s_{2[k]} <_{\psi_{s^2}} o$, como queríamos probar.

2. Si $o \notin \varepsilon(q(s_2))$, entonces como $\bar{s}^2$ es una secuencia factible, necesariamente $\exists y < x$ tal que $\bar{s}^2_{[y]} \in \varepsilon(q(s_2))$. Llamaremos $o' = \bar{s}^2_{[y]}$. Luego, tendremos que:

$$\psi_{\bar{s}^2}(o') + p(o') \leq \psi_{\bar{s}^2}(o) + p(o)$$

   Nuevamente, como reordenar no altera el tiempo de inicio de las operaciones, tendremos que:

$$\psi_{s^2}(o') + p(o') = \psi_{\bar{s}^2}(o') + p(o') \leq \psi_{\bar{s}^2}(o) + p(o) = \psi_{s^2}(o) + p(o)$$

   De lo probado en 1) y de la última desigualdad se obtiene $s_{2[k]} <_{\psi_{s^2}} o' <_{\psi_{s^2}} o$, por lo que o ocupa una posición mayor que k.

Finalmente, como $\forall o \notin q(s_2)$, o ocupa una posición mayor que k en s^2, entonces se infiere que $s^2_{[1,k]} = s_2$. Luego, por definición, la dominancia resulta ser directa.

□

Proposicion 4.11. *Sea $s^1 \in \mathring{S}(\mathcal{O})$, y sea $k \in \{1, \cdots, nm-1\} : s^1_{[1,k]}$ es dominada indirectamente por $s_2 \in \mathring{S}(q(s^1_{[1,k]}))$. Entonces, $\exists o \in q(s^2_{[1,k]})$ tal que $o \notin \eta(s_2)$.*

Demostración. En principio, tenemos que $\big(\varepsilon(q(s_2))\backslash\eta(s_2)\big) \neq \emptyset$. De no suceder esto, tendríamos que $\forall o \in \varepsilon(q(s_2)) : o \in \eta(s_2)$, y por la Proposición 4.10, s_2 dominaría directamente a $s^1_{[1,k]}$, lo cual es absurdo.

Luego, veamos que para alguna operación $o \in \big(\varepsilon(q(s_2))\backslash\eta(s_2)\big)$, vale que $o \in q(s^2_{[1,k]})$.

Ya que la dominancia es indirecta, tenemos que $q(s^2_{[1,k]}) \cap \big(\mathcal{O}\backslash q(s^1_{[1,k]})\big) \neq \emptyset$. Sea $o \in q(s^2_{[1,k]}) \cap \big(\mathcal{O}\backslash q(s^1_{[1,k]})\big)$. Tendremos los siguientes casos:

1. $o \in \varepsilon(q(s^1_{[1,k]})) \wedge o \notin \eta(s_2)$. En este caso, hemos probado el resultado.

2. $o \in \varepsilon(q(s^1_{[1,k]})) \wedge o \in \eta(s_2)$. En este caso, de la demostración de la Proposición 4.10, se deduce que $o \notin q(s^2_{[1,k]})$, lo cual conduce a un absurdo.

3. $o \notin \varepsilon(q(s^1_{[1,k]}))$. No es díficil ver que debe existir $o' \in \big(\mathcal{O}\backslash q(s^1_{[1,k]})\big)$, tal que $o' <_{\psi_{s^2}} o$ y $\big(j(o') = j(o) \vee m(o') = m(o)\big)$. De que $o' <_{\psi_{s^2}} o$ y $o \in q(s^2_{[1,k]})$, se infiere que $o' \in q(s^2_{[1,k]})$.

En el caso 1) el resultado está probado. El caso 2) lleva a una contradicción, por lo que no es posible que suceda. Por otro lado, si sucediera el caso 3), podemos obtener una nueva operación o' sobre la que ejecutar el mismo análisis. Es fácil ver que el caso 3) se puede repetir una cantidad finitas de veces, por lo que inevitablemente sucederá 1). □

Definición 4.21. *Sea $s \in \mathring{S}(\mathcal{O})$ una secuencia completa y ordenada. Definiremos con el nombre de* Cadena de dominancia *que empieza en s a una sucesión finita o infinita de secuencias $\{s^i\}_{i \in I \subseteq \mathbb{N}} \subseteq \mathring{S}(\mathcal{O})$ tal que:*

1. $s^1 = s$

2. $\forall i < |I|, \exists k^i, \bar{s}^{i+1} \in \mathring{S}(q(s^i_{[1,k^i]}))$ *tal que* $\bar{s}^{i+1} \preceq s^i_{[1,k^i]}$

3. $\forall i < |I|, s^{i+1} = \mu(s^i, k^i, \bar{s}^{i+1})$

Es decir: la sucesión $\{s^i\}$ contiene secuencias completas que se dominan una a las otras, directa o indirectamente, en distintas etapas. El número k^i designa la etapa en que s^{i+1} domina a s^i.

Nuestro objetivo es demostrar que el algoritmo encuentra una solución óptima. Para ello, nuestra argumentación será básicamente la siguiente: supongamos que tenemos una solución óptima s^1. Si esta solución es generada por el algoritmo, el resultado queda probado. Si, en cambio, s^1 no es generada por el algoritmo, entonces s^1 es descartada en alguna etapa. Es decir, existe una subsecuencia $s_1 = s^1_{[1,k^1]}$ de s^1 que es dominada por alguna secuencia s_2. Esta dominancia indica que alguna completación de s_2 dará una nueva solución óptima: s^2. Nuevamente, si s^2 es generada por el algoritmo, el resultado queda probado. Si no, alguna subsecuencia de s^2 será dominada, dando lugar a una nueva solución óptima s^3, etc. Como la cantidad de soluciones óptimas es finita, este proceso se acaba, *salvo* que la cadena de dominancia contenga un ciclo, es decir: que exista un s^i que sea igual a s^1, lo cual produciría el descarte de todas las soluciones de la cadena. La siguiente proposición muestra que esto no es posible, bajo ciertas hipótesis.

Proposicion 4.12. *Sea $s \in \mathring{S}(\mathcal{O})$, y $\{s^i\}_{i \in I \subseteq \mathbb{N}}$ una cadena de dominancia que empieza en s, y supongamos que s^2 domina* indirectamente *a $s^1 = s$. Supongamos además que $k^i \leq k^1$ para todo i. Es decir: que todas las dominancias se realizan en etapas iguales o anteriores a la etapa k^1. Entonces, $s^i \neq s$ para todo i.*

Demostración. De la definición de dominancia indirecta, por la Proposición 4.11 se deduce que $\exists o \in \varepsilon(q(s^1_{[1,k^1]}))$ tal que $o \notin \eta(\bar{s}^2)$ y $o \in q(s^2_{[1,k^1]})$. Es decir, o se adelanta a la última operación de $\bar{s}^2$, desordenando esta secuencia.

Probaremos dos resultados intermedios:

1. $C_{\text{máx}}(s^2_{[1,k^1+1]}) \leq \psi_{s^1}(o)$.

   En efecto, si $o \in \eta(s^1_{[1,k^1]})$, el resultado se deriva directamente de la relación de dominancia:

$$C_{\text{máx}}(s^2_{[1,k^1+1]}) \leq C_{\text{máx}}(\bar{s}^2) \leq \psi(s^1_{[1,k^1]}, o) \leq \psi_{s^1}(o).$$

   En cambio, si $o \notin \eta(s^1_{[1,k^1]})$, entonces, como s^1 es ordenada, entonces, al igual que en el ítem 6 de la Proposición 4.7, debe haber una operación $\tilde{o}$ tal que $m(\tilde{o}) = m(o)$ y que se programe en s^1 antes que o. Podemos suponer que $\tilde{o} \in \eta(s^1_{[1,k^1]})$, pues en caso contrario habrá otra secuencia $\tilde{\tilde{o}}$ anterior a $\tilde{o}$, etc. Por lo tanto, aplicando a $\tilde{o}$ el caso anterior:

$$C_{\text{máx}}(s^2_{[1,k^1+1]}) \leq \psi_{s^1}(\tilde{o}) < \psi_{s^1}(o),$$

   lo que prueba la afirmación.

2. $\psi_{s^i}(o) + p(o) \leq C_{\text{máx}}(s^2_{[1,k^1+1]})$ $\forall i \geq 2$.

   Lo probamos por inducción:

   - CASO BASE: $i = 2$.

     Como $o \in q(s^2_{[1,k^1]})$, o se programa en s^2 antes que la $(k^1+1)-$ésima operación, $s^2_{[k^1+1]}$. Luego:

     $$\psi_{s^2}(o) + p(o) \leq \psi_{s^2}(s^2_{[k^1+1]}) + p(s^2_{[k^1+1]}) = C_{\text{máx}}(s^2_{[1,k^1+1]}).$$

   - PASO INDUCTIVO: Analizamos dos casos:

     a) $o \notin q(s^i_{[1,k^i]})$:

        Como $s^{i+1} = \mu(s^i, k^i, \bar{s}^{i+1})$ y $\bar{s}^{i+1}$ domina a $s^i_{[1,k^i]}$, podemos aplicar el Corolario 4.1: toda operación fuera de $q(s^i_{[1,k^1]})$ se planifica antes en s^{i+1} que en s^i, con lo cual:

        $$\psi_{s^{i+1}}(o) + p(o) \leq \psi_{s^i}(o) + p(o) \leq C_{\text{máx}}(s^2_{[1,k^1+1]}),$$

        donde en la última desigualdad aplicamos la H.I.

     b) $o \in q(s^i_{[1,k^i]})$:

        Teniendo que cuenta que todas las dominancias se producen en etapas $k^i \leq k^1$ y aplicando iterativamente el Corolario 4.1 obtenemos:

        $$\begin{aligned} C_{\text{máx}}(s^i_{[1,k^i+1]}) \leq C_{\text{máx}}(s^i_{[1,k^1+1]}) \leq C_{\text{máx}}(s^{i-1}_{[1,k^1+1]}) \leq \dots \\ \leq C_{\text{máx}}(s^2_{[1,k^1+1]}). \end{aligned}$$

        Por lo tanto, para probar la afirmación 2. nos alcanza con ver que:

        $$\psi_{s^{i+1}}(o) + p(o) \leq C_{\text{máx}}(s^i_{[1,k^i+1]}) \tag{22}$$

        Si $\psi_{s^{i+1}}(o) + p(o) \leq C_{\text{máx}}(s^i_{[1,k^i]})$, (22) se sigue trivialmente pues $C_{\text{máx}}(s^i_{[1,k^i]}) \leq C_{\text{máx}}(s^i_{[1,k^i+1]})$.

        En caso contrario tenemos que:

        $$C_{\text{máx}}(\bar{s}^{i+1}) \geq \psi_{s^{i+1}}(o) + p(o) > C_{\text{máx}}(s^i_{[1,k^i]}),$$

        por lo que podemos aplicar el tercer ítem de la Proposición 4.8, que nos dice que toda operación o en $\varepsilon(q(\bar{s}^{i+1}))$ pertenece a $\eta(\bar{s}^{i+1})$, lo que implica que $\bar{s}^{i+1}$ domina *directamente* a $s^i_{[1,k^i]}$,

por lo que $s^{i+1}_{[1,k^i]} = \bar{s}^{i+1}$, lo que indica que o se planifica en s^{i+1} a lo sumo en la etapa k^i, por lo cual:

$$\begin{aligned}\psi_{s^{i+1}}(o) + p(o) &\leq C_{\text{máx}}(s^{i+1}_{[1,k^i]}) \leq C_{\text{máx}}(s^{i+1}_{[1,k^i+1]}) \\ &\leq C_{\text{máx}}(s^{i}_{[1,k^i+1]}),\end{aligned}$$

donde en el último paso aplicamos el Corolario 4.1 y probamos (22), concluyendo la demostración de la afirmación 2.

Finalmente, de 1. y 2. obtenemos que

$$\psi_{s^i}(o) + p(o) \leq C_{\text{máx}}(s^2_{[1,k^1+1]}) \leq \psi_{s^1}(o) \Rightarrow \psi_{s^i}(o) < \psi_{s^1}(o)$$

de donde se deduce que $s^i \neq s^1 = s$. □

4.4.1. Construcción de la solución óptima

Ya estamos en condiciones de probar que el algoritmo encuentra una solución óptima. Para ello, introduciremos primero algunas nociones que permitirán simplificar ligeramente la notación.

El algoritmo, que hemos descripto esquemáticamente con antelación, consiste en la construcción recursiva de los siguientes conjuntos:

Definición 4.22.

$$\mathcal{X}(Q) = \{[o_j]\},\ si\ Q = \{o_j : j \in \{1, \cdots, n\}\}$$

$$\mathcal{X}(Q) = \bigcup_{o \in \lambda(Q)} \bigcup_{s \in \mathcal{X}(\overset{(\preceq)}{Q} \setminus \{o\})\ con\ o \in \eta(s)} \{s + o\},\ si\ Q\ factible,\ |Q| > 1$$

Es decir: se generan las secuencias factibles de una sola operación y luego en cada etapa se generan todas las posibles expansiones ordenadas de las secuencias disponibles y se descartan las secuencias dominadas.

De esta forma, dentro de cada iteración del esquema recursivo los conjuntos factibles Q aumentarán su cardinal en 1. Así, de forma iterativa tendremos que, eventualmente, las secuencias de $\mathcal{X}(Q)$ serán completas, es decir, $Q = \mathcal{O}$.

A modo de ejemplo, en la Figura 13 se muestra la reducción de nodos mediante este mecanismo para la instancia $\mathcal{I}$. Se han denotado con color negro los nodos que representan secuencias desordenadas, y en rojo las secuencias ordenadas que son dominadas por otras secuencia minimales.

La siguiente notación nos servirá para resumir los conjuntos de secuencias generados por el algoritmo en cada etapa, antes y después de descartar las secuencias dominadas:

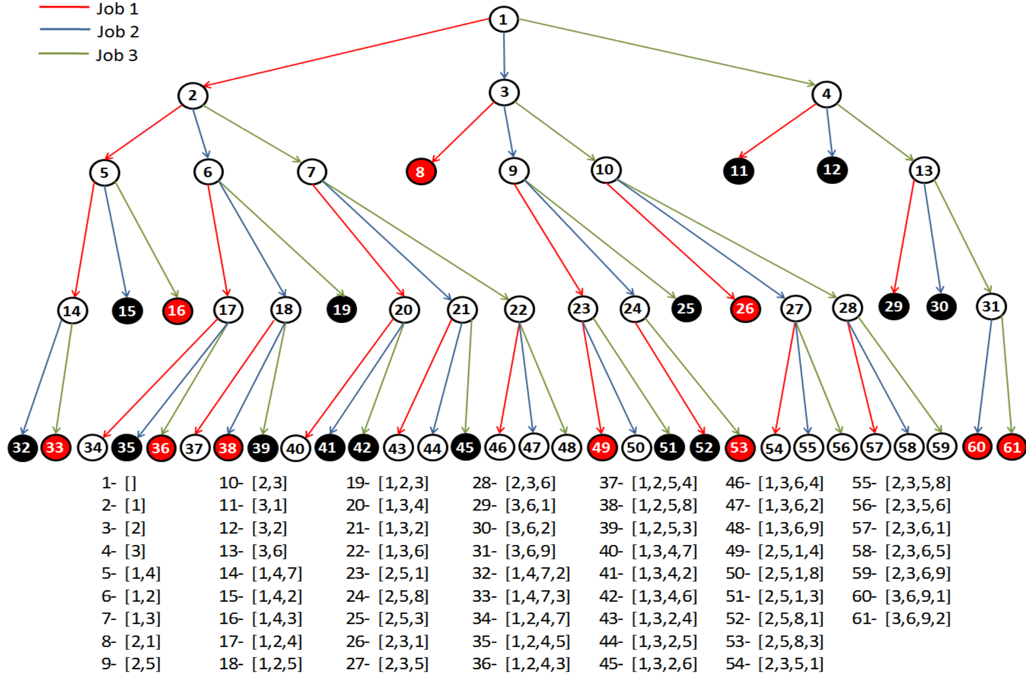


Figura 13: Árbol de búsqueda de $\mathcal{I}$ hasta la etapa 4 con expansión ordenada y eliminación por dominancia

Definición 4.23. *Sea $k = 1, 2, \cdots, nm$ y $|Q| = k$, notamos $X(Q) \subseteq \mathring{S}(Q)$ los conjuntos de secuencias ordenadas que el algoritmo genera al expandir secuencias no dominadas de cardinal $k-1$.*

- $X_k = \bigcup_{Q\ factible:|Q|=k} X(Q)$.
- $\overset{(\preceq)}{X}_k = \bigcup_{Q\ factible:|Q|=k} \left(\overset{(\preceq)}{X(Q)}\right)$

La siguiente proposición garantiza que en toda etapa del algoritmo hay disponible una secuencia parcial que puede completarse a una secuencia óptima.

Proposicion 4.13. *Sea $k \in \{1, 2, \cdots, nm-1\}$. Si $\exists s \in \mathring{S}(\mathcal{O})$ secuencia óptima tal que $s_{[1,k]} \in \overset{(\preceq)}{\mathcal{X}}_k$, entonces $\exists s' \in \mathring{S}(\mathcal{O})$ secuencia óptima tal que $s'_{[1,k+1]} \in \overset{(\preceq)}{\mathcal{X}}_{k+1}$.*

Demostración. Sea $s^1 = s \in \mathring{S}(\mathcal{O})$. Definiremos una cadena de dominancia que comienza en s del siguiente modo:

- $k^i - 1 =$ máximo índice l tal que $s^i_{[1,l]} \in \overset{(\preceq)}{\mathcal{X}}(q(s^i_{[1,l]}))$
- $\bar{s}^{i+1} \in \overset{(\preceq)}{\mathcal{X}}(q(s^i_{[1,k^i]}))$ tal que $\bar{s}^{i+1} \preceq s^i_{[1,k^i]}$
- $s^{i+1} = \mu(s^i, k^i, \bar{s}^{i+1})$

Básicamente, para cada secuencia completa s^i, buscaremos la subsecuencia de longitud máxima $(k^i - 1)$ de forma que dicha subsecuencia se encuentre en el conjunto $\overset{(\preceq)}{\mathcal{X}}_{k^i-1}$. Luego, tomaremos como $\bar{s}^{i+1} \in \overset{(\preceq)}{\mathcal{X}}(q(s^i_{[1,k^i]}))$ a una subsecuencia que domina a $s^i_{[1,k^i]}$. Observemos que como $s^i_{[1,k^i]} \in \mathcal{X}(q(s^i_{[1,k^i]}))$ y $s^i_{[1,k^i]} \notin \overset{(\preceq)}{\mathcal{X}}(q(s^i_{[1,k^i]}))$, queda probada la existencia de $\bar{s}^{i+1}$. Finalmente, expandiremos y ordenaremos $\bar{s}^{i+1}$ según el operador μ, para obtener la secuencia $s^{i+1} := \mu(s^i, k^i, \bar{s}^{i+1})$.

Naturalmente, el caso interesante es cuando $k = k^1 - 1$, puesto que si $k < k^1 - 1$, entonces la secuencia $s_{[1,k]}$ es expandida de manera ordenada a la secuencia $s_{[1,k+1]}$, y el resultado vale. Por el contrario, si $k > k^1 - 1$, entonces no es cierto que $s_{[1,k]}$ esté en $\overset{(\preceq)}{\mathcal{X}}_k$ y la hipótesis resulta falsa.

Gracias al Corolario 4.1, sabemos que todas las secuencias s^i son óptimas.

Separamos la demostración en dos partes, según s^2 domine a s^1 directa o indirectamente.

1. Si $\bar{s}^2$ domina directamente a $s^1_{[1,k^1]}$, entonces $k^2 > k^1$.

   En efecto, como $\bar{s}^2$ domina directamente a $s^1_{[1,k^1]}$, entonces se tiene que $s^2_{[1,k^1]} = \bar{s}^2$ y $\bar{s}^2 \in \overset{(\preceq)}{\mathcal{X}}(q(s^2_{[1,k^1]}))$. Luego, por definición de k^2, tenemos que $k^2 > k^1$.

   Observemos que en tal caso, la Proposición queda demostrada pues $s^2_{[1,k^1]}$ es expandida, obteniéndose una secuencia en óptima en una etapa posterior a k.

2. Si $\bar{s}^2$ domina indirectamente a $s^1_{[1,k^1]}$, entonces caben dos posibilidades:

   a) Todos los k^i son menores o iguales que k^1.
   En este caso, la idea es aplica la Proposición 4.12. Sin embargo, la aplicación directa de este resultado sólo garantiza que las secuencias de la cadena son distintas de s^1, es decir: excluye la posibilidad de que exista un ciclo de dominancia que incluya a s^1. Necesitamos un argumento algo más fino que nos permita probar

la imposibilidad de que $\{s^i\}$ contenga algún ciclo que no incluya a s^1. Lo hacemos por el absurdo: supongamos que existe un j_1 y un $j_2 > j_1$ tales que $s^{j_1} = s^{j_2}$. Tomemos entonces T el índice tal que: $k^T = \max\{k^i : j_1 \leq i \leq j_2\}$. Por definición de T vale que s^{T+1} domina a s^T indirectamente, pues si la dominancia fuera directa, k^{T+1} sería mayor que k^T. Podemos tomar entonces la cadena de dominancia que empieza en s^T dada por:

$$s^T, s^{T+1}, \ldots, s^{j_2} = s^{j_1}, s^{j_1+1} \ldots, s^T.$$

En esta cadena la primer dominancia es indirecta y todas las dominancias se realizan en etapas anteriores o iguales a la primera (por definición de T). En consecuencia, podemos aplicar aquí la Proposición 4.12, concluyendo que todas las secuencias de la cadena son distintas, lo cual es absurdo. Este absurdo proviene de suponer que $s^{j_1} = s^{j_2}$.

De este modo probamos que la cadena de dominancia no puede contener ciclos. Como la cantidad de soluciones óptimas es finita, el conjunto I también debe serlo y $s^{|I|}$ es una solución óptima que pertenece a la cadena y que no es dominada por ninguna otra, por lo cual es construida completa por el algoritmo. En particular: $s^{|I|}_{[1,k+1]} \in \overset{(\preceq)}{\mathcal{X}}_{k+1}$.

b) Existe algún $j \in I$ para el cual $k^j > k^1$. En tal caso, como $k^j \geq k+1$, la sucesión $s^j_{[1,k+1]} \in \overset{(\preceq)}{\mathcal{X}}_{k+1}$.

De este modo, la Proposición queda demostrada para cualquiera de los posibles casos.

□

Proposicion 4.14. $\overset{(\preceq)}{\mathcal{X}}(\mathcal{O}) = \overset{(\preceq)}{\mathcal{X}}_{nm} = \{s\}$, *con* s *una secuencia completa, ordenada y óptima.*

Demostración. Dividiremos la demostración en dos partes:

1. Probaremos que $\exists s \in \overset{(\preceq)}{\mathcal{X}}(\mathcal{O})$ secuencia óptima.

2. Probaremos que $\#(\overset{(\preceq)}{\mathcal{X}}(\mathcal{O})) = 1$.

1. Probaremos que $\exists s \in \mathring{S}(\mathcal{O})$ secuencia óptima tal que $\forall k = 1, \cdots, nm :$ $s_{[1,k]} \in \overset{(\preceq)}{\mathcal{X}}_k$. Haremos la prueba por inducción en el número de etapas k.

- CASO BASE: $k = 1$ La validez del enunciado resulta trivial en este caso, dado que $\overset{(\preceq)}{\mathcal{X}}_1 = \bigcup_{j=1,\cdots,n}[o_j] = \bigcup_{s\in\mathring{S}(\mathcal{O}),s\ optima} s_{[1,1]}$ Luego, la propiedad vale no sólo para una secuencia, sino para todas las secuencias ordenadas óptimas.
- PASO INDUCTIVO: Quiero ver que la propiedad vale para $k+1$. No obstante, como por hipótesis inductiva $\exists s \in \mathring{S}(\mathcal{O})$ óptima tal que $s_{[1,k]} \in \overset{(\preceq)}{\mathcal{X}}_k$ y $k < nm$, entonces por la Proposición 4.13 tenemos que $\exists s' \in \mathring{S}(\mathcal{O})$ óptima tal que $s'_{[1,k+1]} \in \overset{(\preceq)}{\mathcal{X}}_{k+1}$, como queríamos ver.

Finalmente, la propiedad vale para $k = nm$. Luego, $\exists s \in \mathring{S}(\mathcal{O})$ óptima tal que $s_{[1,nm]} = s \in \overset{(\preceq)}{\mathcal{X}}_{k+1} = \overset{(\preceq)}{\mathcal{X}}(\mathcal{O})$

2. Veamos que $\#\big(\overset{(\preceq)}{\mathcal{X}}(\mathcal{O})\big) = 1$.

   En principio, de 1) se deduce que $\overset{(\preceq)}{\mathcal{X}}(\mathcal{O}) \neq \emptyset$. Supongamos que $\exists s_1, s_2 \in \overset{(\preceq)}{\mathcal{X}}(\mathcal{O})$. Luego, tendremos que

   $$C_{\text{máx}}(s_1) \leq C_{\text{máx}}(s_2) \vee C_{\text{máx}}(s_2) < C_{\text{máx}}(s_1)$$

   En el caso de que $C_{\text{máx}}(s_1) \leq C_{\text{máx}}(s_2)$, entonces, por la Definición 4.18, $s_1 \preceq s_2$. De aquí se infiere que $s_1 \in \overset{(\preceq)}{\mathcal{X}}(\mathcal{O}) \wedge s_2 \notin \overset{(\preceq)}{\mathcal{X}}(\mathcal{O})$, ó $s_2 \in \overset{(\preceq)}{\mathcal{X}}(\mathcal{O}) \wedge s_1 \notin \overset{(\preceq)}{\mathcal{X}}(\mathcal{O})$.
   Si $C_{\text{máx}}(s_2) < C_{\text{máx}}(s_1)$, entonces es claro que $s_2 \preceq s_1 \wedge s_1 \npreceq s_2$, por lo que $s_2 \in \overset{(\preceq)}{\mathcal{X}}(\mathcal{O}) \wedge s_1 \notin \overset{(\preceq)}{\mathcal{X}}(\mathcal{O})$.
   Por lo tanto, en todos los casos se deduce que $s_1 = s_2$, dado que si $s_1 \neq s_2$, sería absurdo que $s_1 \in \overset{(\preceq)}{\mathcal{X}}(\mathcal{O}) \wedge s_2 \in \overset{(\preceq)}{\mathcal{X}}(\mathcal{O})$.

   Luego, $\#\big(\overset{(\preceq)}{\mathcal{X}}(\mathcal{O})\big) = 1$.

□

La Proposición 4.14 nos dice que $\overset{(\preceq)}{\mathcal{X}}(\mathcal{O}) = \{s\}$, con s secuencia óptima. Luego, cualquier algoritmo que genere iterativamente los conjuntos $\overset{(\preceq)}{\mathcal{X}}(Q)$ y devuelva $\overset{(\preceq)}{\mathcal{X}}(\mathcal{O})$ será considerado válido para resolver el JSSP.

Para finalizar este apartado, haremos algunos comentarios y observaciones adicionales con respecto al trabajo original que creemos pertinentes.

Con este esquema recursivo se evidencia que la descripción de los estados estará dada por los conjuntos factibles Q, de forma similar al problema 6 del capítulo 2. Además, nuestro principio de optimalidad nos asegura que dado un subconjunto factible de operaciones $Q \subseteq \mathcal{O}$ y una secuencia $s \in \overset{\preceq}{\mathcal{X}}(Q)$, entonces todas sus subsecuencias $s_{[1,x]}$ con $x = 1, 2, \cdots, |Q|$ están en sus respectivos conjuntos $\overset{\preceq}{\mathcal{X}}\big(q(s_{[1,x]})\big)$.

La Observación 4.4 pone de manifiesto las limitaciones de nuestro funcional. Dado que el orden definido por la dominancia es parcial, no podremos calcular el mínimo sobre todas las secuencias parciales comparables, sino que deberemos conformarnos con el conjunto de secuencias minimales. Así no se asociará una secuencia minimal a cada estado, sino un conjunto de secuencias minimales.

Con esto nos estamos saliendo de la versión clásica de PD, ya que no es posible plantear la ecuación de Bellman para este problema. En este punto, no resulta del todo claro hasta que punto podemos considerar el planteo en [6] dentro de la PD. Si bien es innegable que la metodología desarrollada guarda cierta relación con su versión clásica, resulta díficil diferenciar si realmente se trata de una aplicación de PD al JSSP, o simplemente de un criterio muy ingenioso y efectivo de poda. En cualquier caso, dejaremos que el lector saque sus propias conclusiones respecto de este tema. En el fondo, al demostrar la validez del esquema iterativo propuesto en 4.22, la categorización de la metología utilizada no parece tener relevancia.

En resumen, hemos hallado una forma iterativa de construir una solución exacta, que descarta soluciones parciales en base a una comparación a nivel de estado. Si bien el número de estados no tiene por qué ser equivalente al número de soluciones parciales, conceptualmente el método propuesto resulta sumamente parecido a PD.

4.5. Secuencias con operaciones retrasadas

En esta subsección introduciremos dos criterios adicionales que nos permitirán descartar mayor cantidad de secuencias parciales, mejorando el rendimiento de nuestro algoritmo expuesto en el próximo capítulo.

4.5.1. Eliminación de secuencias parciales con operaciones retrasadas

Supongamos que debemos resolver el problema de la instancia $\mathcal{I}$ y tenemos la secuencia parcial ordenada $[o_2, o_1]$. Dentro de este contexto, notemos que la operación o_3 se ejecuta en la máquina 3, y además, las restantes operaciones correspondientes a la máquina 3, (o_4 y o_8) no se programarán antes de $t = 2$. Por lo tanto, se podría programar perfectamente la operación o_3 a tiempo 0, ya que $p(o_3) = 2$. Sin embargo, dado que o_3 no está entre las operaciones de $\eta([o_2, o_1])$, no es posible programar o_3 a tiempo 0 de forma ordenada. Luego, cualquier secuencia ordenada en $Comp([o_2, o_1])$ tendrá un schedule asociado ψ_s tal que $\psi_s(o_3) \geq 2$. En otras palabras, cualquier expansión ordenada definirá un schedule en donde se podría *adelantar* la operación o_3 sin violar las condiciones de factibilidad, como se muestra en la Figura 14
.

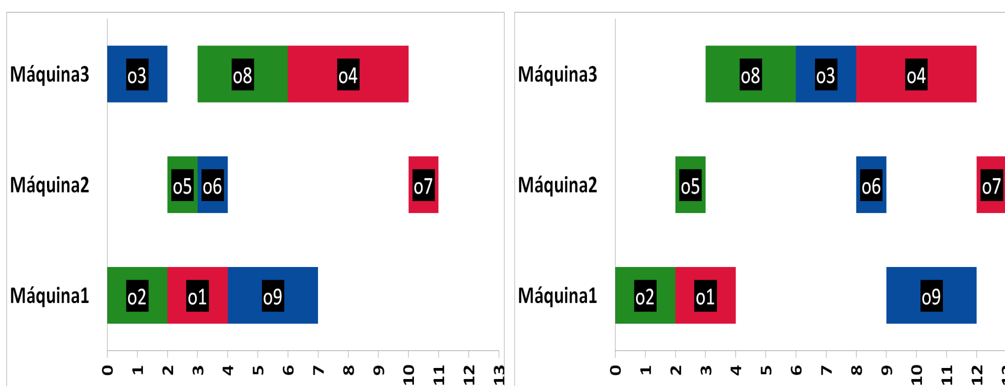


Figura 14: Expansión desordenada de $[o_2, o_1]$ sin retrasos (izquierda) vs. expansión ordenada de $[o_2, o_1]$ con retrasos (derecha)

Dentro de la literatura del JSSP, es común que el espacio de búsqueda se reduzca a sólo schedules en donde ninguna operación puede adelantarse sin producir el atraso de otra. En el ejemplo presentado, cualquier schedule producto de una expansión ordenada de $[o_2, o_1]$ escapa a esta reducción. Por ende, sería deseable hallar alguna condición extra, de forma de eliminar tempranamente las secuencias parciales en donde se pudiesen *adelantar* operaciones. Introduciremos dicha condición a través de las siguientes definiciones.

Definición 4.24. *Sea $Q \subset \mathcal{O}$ factible, sea $s \in \mathring{S}(Q)$. Para $i = 1, 2, \cdots, m$, definiremos:*

- $\eta(s,i) = \{o \in \eta(s) : m(o) = i\}$
- $\overline{\eta(s,i)} = \{o \in \eta(s,i) : \nexists o' \in \eta(s,i), o' \neq o, \text{ tal que } \psi(s,o') + p(o') \leq \psi(s,o)\}$

Finalmente, definiremos $\overline{\eta(s)} = \bigcup_{i=1}^{m} \overline{\eta(s,i)}$

Definición 4.25. *Sea* $Q \subset \mathcal{O}$ *factible, sea* $s \in \mathring{S}(Q)$. *Definiremos*

$$\widehat{\eta(s)} = \{o \in \overline{\eta(s)} : \nexists o' \in \overline{\eta(s)}, o' \neq o \text{ tal que } o' \notin \eta(s+o)\}$$

Finalmente, diremos que $o \in \eta(s)$ genera retrasos *si* $o \notin \widehat{\eta(s)}$.

Observemos que, dada una $o \in \eta(s)$, o generará retrasos si $o \notin \overline{\eta(s)}$, o bien $\exists o' \in \overline{\eta(s)}$ tal que se cumple alguna de las siguientes dos condiciones:

- $\psi(s,o') + p(o') < \psi(s,o) + p(o)$
- $\psi(s,o') + p(o') = \psi(s,o) + p(o) \wedge m(o') < m(o)$

Cabe aclarar que la segunda condición resulta equivalente a que $\exists o' \in \overline{\eta(s)} : o' \notin \eta(s+o)$.

En la Figura 15 se muestran ejemplos para las dos condiciones.

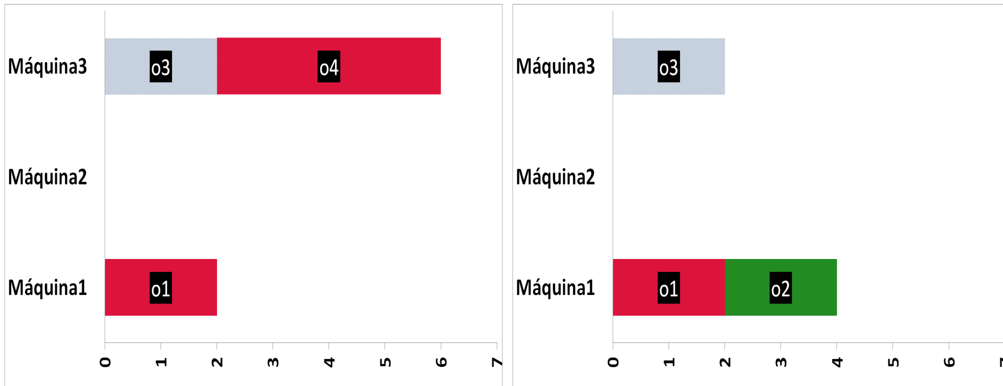


Figura 15: Secuencias con operaciones atrasadas

Cada schedule de la gráfica es producto de la expansión ordenada de la secuencia $[o_1]$ en la secuencia s. Las operaciones programadas se muestran nítidas. Además, se observa en ambos casos que $\psi(s,o_4) = 2 = \psi(s,o_3) + p(o_3)$, lo que se manifiesta representando la operación o_3 con un color tráslucido.

El schedule de la izquierda corresponde a la secuencia ordenada $s = [o_1, o_4]$. La operación $o_4 \notin \overline{\eta([o_1])}$, pues $\psi([o_1], o_3) + p(o_3) = \psi([o_1], o_4)$ y

$m(o_4) = m(o_3)$. De aquí se puede inferir que $o_4 \notin \widehat{\eta([o_1])}$. Además, dado que no hay más operaciones en $\eta([o_1])$ cuya ejecución corresponda a la máquina 3, entonces $o_3 \in \overline{\eta([o_1], 3)}$.

Por otra parte, en el schedule de la derecha tendremos que $s = [o_1, o_2]$. En este caso, $o_2 \notin \widehat{\eta([o_1])}$, pues $o_3 \in \overline{\eta([o_1]))}$ y $\psi([o_1], o_3) + p(o_3) = \psi([o_1], o_2) < \psi([o_1], o_2) + p(o_2)$.

Es claro que en ambos schedules la operación o_3 podría programarse a tiempo 0 sin ningún problema. No obstante, $o_3 \notin \eta([o_1, o_4])$ y $o_3 \notin \eta([o_1, o_2])$, por lo que hacer lo anterior desordenaría la secuencia. O mejor dicho, nunca obtendríamos una secuencia s' ordenada y completa en donde $\psi_{s'}(o_3) = 0$.

De aquí que si utilizasemos el conjunto de operaciones $\widehat{\eta(s)}$ para expandir cada secuencia ordenada en lugar de utilizar $\eta(s)$, podríamos evitar la generación de secuencias en donde alguna operación pueda adelantarse. Los resultados y definiciones a continuación justificarán la utilización de este mecanismo para resolver el JSSP de forma exacta.

En principio, para que las demostraciones resulten más sencillas de escribir, introduciremos el concepto de *secuencia nula*.

Definición 4.26. *Definiremos la secuencia nula* $[]$ *como una secuencia que no contiene ninguna operación. Así:*

- $q([]) = \emptyset$
- $d(s) = 0$
- $\lambda(q([])) = \emptyset$
- $\varepsilon(q([])) = \{o_j : 1 \leq j \leq n\}$
- $\eta([]) = \varepsilon(q([]))$
- $\forall o \in \eta([]) : [] + o = [o]$

Procederemos a demostrar que si reemplazamos en la Definición 4.22 la noción de $\eta(s)$ por $\widehat{\eta(s)}$, obtendremos también una secuencia ordenada y óptima.

Proposicion 4.15. *Sea* $Q \subset \mathcal{O}$ *y* $s \in \mathring{S}(Q)$ *ó* $s = []$. *Entonces,* $\forall o^1 \in \left(\eta(s) \backslash \widehat{\eta(s)}\right), s^1 \in Comp(\mathring{s} + o^1)$, $\exists o^2 \in \widehat{\eta(s)}, s^2 \in Comp(\mathring{s} + o^2)$ *tal que* $C_{\text{máx}}(s^2) \leq C_{\text{máx}}(s^1)$

Demostración. En principio si $o^1 \in \left(\eta(s) \backslash \widehat{\eta(s)}\right)$, entonces o^1 genera retrasos. Luego, $\exists o \in \widehat{\eta(s)}, o \neq o^1$ que cumplirá alguna de las condiciones mencionadas

recientemente. Tomaremos o^2 como la operación con el menor tiempo de finalización de entre las operaciones de $\widehat{\eta(s)}$. En caso de haber más de una, tomaremos como o^2 la que se ejecute en la máquina de menor índice.

Veamos que valen las siguientes dos afirmaciones:

- $\psi(s+o^2, o^1) = \psi(s, o^1)$
- $o^1 \in \eta(s+o^2)$

Para ello, dividiremos la prueba en dos casos

1. Si $m(o^1) = m(o^2)$. En este caso, $o^2 \in \overline{\eta(s, m(o^2))}$ y $o^1 \in \overline{\eta(s, m(o^2))}$. Por definición, tenemos que

$$\begin{aligned}\psi(s+o^2, o^1) &= \max_{o \in Pre(q(s+o^2), o^1)} \{\psi_{s+o^2}(o) + p(o)\} \\ &\leq \max_{o \in q(s+o^2)} \{\psi_{s+o^2}(o) + p(o)\}\end{aligned}$$

   Como $o^2 \in \eta(s) \Rightarrow C_{\text{máx}}(s+o^2) = \psi(s, o^2) + p(o^2)$. Luego:

$$\max_{o \in q(s+o^2)} \{\psi_{s+o^2}(o) + p(o)\} = C_{\text{máx}}(s+o^2) = \psi(s, o^2) + p(o^2)$$

   Por otro lado, por definición de $\overline{\eta(s, m(o^2))}$, tenemos que $\psi(s, o^2) + p(o^2) \leq \psi(s, o^1)$. De lo anterior y de aquí tendremos que

$$\psi(s+o^2, o^1) \leq \psi(s, o^2) + p(o^2) \leq \psi(s, o^1)$$

   Como $q(s) \subset q(s+o^2)$ entonces finalmente vale $\psi(s+o^2, o^1) = \psi(s, o^1)$. Además, resulta claro de lo anterior que $o^1 \in \eta(s+o^2)$, pues:

$$\psi(s+o^2, o^1) + p(o^1) > \psi(s+o^2, o^1) = \psi(s, o^1) =$$

$$= \psi(s, o^2) + p(o^2) = C_{\text{máx}}(s+o^2)$$

2. Si $m(o^1) \neq m(o^2)$, entonces $o^2 \in \overline{\eta(s, m(o^2))}$ y $o^2 \notin \eta(s+o^1)$. Como $j(o^1) \neq j(o^2)$, entonces $o^2 \notin Pre(q(s+o^2), o^1)$. Luego, se tiene que $Pre(q(s), o^1) = Pre(q(s+o^2), o^1)$, de donde, por la definición se infiere que

$$\psi(s+o^2, o^1) = \psi(s, o^1)$$

   De lo observado y de la definición de o^2 es claro que alguna de las siguientes condiciones es verdadera:

- $C_{\text{máx}}(s + o^2) = \psi(s, o^2) + p(o^2) < \psi(s, o^1) + p(o^1)$
- $C_{\text{máx}}(s+o^2) = \psi(s, o^2)+p(o^2) = \psi(s, o^1)+p(o^1) \wedge m(o^2) < m(o^1)$

Luego, dado que $\psi(s, o^1) = \psi(s + o^2, o^1)$, reemplazando en ambas desigualdades se obtiene que $o^1 \in \eta(s + o^2)$, como queríamos ver.

Ahora, definiremos $\bar{s}^2$ como la expansión ordenada de la secuencia $s + o^2$ con las operaciones de $s^1_{[k+1,nm]}$ a excepción de o^2, respetando el orden de aparición. Luego, de forma análoga a la demostración del Corolario 4.1 se puede probar por inducción que $C_{\text{máx}}(\bar{s}^2) \leq C_{\text{máx}}(s^1)$. Básicamente, con lo anterior hemos demostrado la validez del caso base para la operación o^1. Luego, utilizando la hipótesis inductiva podremos llegar al resultado.

Obviaremos la exposición de esta prueba por cuestiones inherentes a la extensión y simplicidad, ya que no posee ninguna dificultad adicional a la del Corolario 4.1. El único detalle es que o^2 ya se encuentra programada en $\bar{s}^2$, por lo que el tiempo de inicio de o^2 será mayor en s^1 que en $\bar{s}^2$.

Finalmente, si definimos s^2 como $\bar{s}^2$ ordenado, obtendremos que $s^2 \in \mathring{S}(Q)$ y $C_{\text{máx}}(s^2) = C_{\text{máx}}(\bar{s}^2) \leq C_{\text{máx}}(s^1)$, como queríamos ver.

Por último, para que $s^2 \in \mathring{Comp}(s + o^2)$, resta ver que $s^2_{[1,d(s)+1]} = s + o^2$. Para ello, supongamos que esto no pasa. Luego, como el orden de la secuencia se ve alterado, necesariamente tendremos $o \in \eta(s), o \neq o^2$ tal que $\psi(s, o) + p(o) \leq \psi(s, o^2) + p(o^2)$. Separaremos en dos casos la demostración:

1. En el caso de que $o \in \widehat{\eta(s)}$ es muy sencillo ver que o no puede desordenar la secuencia $s+o^2$, pues, o^2 es la operación en $\widehat{\eta(s)}$ con menor tiempo de finalización y menor máquina de ejecución. Por lo tanto resulta absurdo que o se programe antes que o^2 en s^2.

2. En el caso en que $o \in \left(\eta(s)\setminus\widehat{\eta(s)}\right)$, tendremos que o genera retrasos. Entonces $\exists o' \in \widehat{\eta(s)} : \psi(s, o') + p(o') \leq \psi(s, o) + p(o) \leq \psi(s, o^2) + p(o^2)$, por lo que nos remitiremos al caso 1) con las operaciones o' y o^2. Nuevamente, se llega a una contradicción.

Finalmente, del absurdo se deduce que $s^2_{[1,d(s)+1]} = s+o^2$, como queríamos ver.

$\square$

Definición 4.27.

$$\mathcal{Y}(Q) = \{[o_j]\},\ si\ Q = \{o_j \in \widehat{\eta([])}\}$$

$$\mathcal{Y}(Q) = \bigcup_{o\in\lambda(Q)} \bigcup_{s\in\overset{(\preceq)}{\mathcal{Y}}(Q\setminus\{o\})\ con\ o\in\widehat{\eta(s)}} \{s + o\},\ si\ Q\ factible,\ |Q| > 1$$

Proposicion 4.16. $\overset{(\preceq)}{\mathcal{Y}}(\mathcal{O}) = \{s\}$, *con s una secuencia completa, ordenada y óptima.*

Demostración. Probaremos que $\exists s \in \mathring{S}(\mathcal{O})$ secuencia óptima tal que $\forall k = 1, \cdots, nm : s_{[1,k]} \in \overset{(\preceq)}{\mathcal{Y}}_k$. Haremos la prueba por inducción en el número de etapas k.

- CASO BASE: $k = 1$ Como ya hemos probado que la prueba vale para $\mathcal{X}_1$, entonces tendremos que $s \in \mathring{S}(\mathcal{O})$ óptima, con $s_{[1,1]} = [o] \in \mathcal{X}_1$. Luego, si $o \in \widehat{\eta([])}$, entonces $[o] \in \mathcal{Y}_1$, con lo que queda probado el resultado.

  En caso contrario, por la Proposición 4.15, $\exists o' \in \widehat{\eta([])}, s' \in \mathring{S}(\mathcal{O})$ óptima, tal que $s'_{[1,1]} = [o']$, como queríamos ver.

- PASO INDUCTIVO: Por hipótesis inductiva tenemos que $\exists s \in \mathring{S}(\mathcal{O})$ óptima tal que $s_{[1,k]} \in \overset{(\preceq)}{\mathcal{Y}}_k$ y $k < nm$.

  Sea $o = s_{[k+1]}$. Si $o \in \widehat{\eta(s_{[1,k]})}$, entonces $s_{[1,k+1]} \in \mathcal{Y}_{k+1}$. En el caso de que $o \notin \widehat{\eta(s_{[1,k]})}$, entonces, nuevamente por la Proposición 4.15, $\exists o' \in \widehat{\eta(s_{[1,k]})}$ y $s' \in \mathring{S}(\mathcal{O})$ óptima, tal que $s'_{[1,k+1]} = s_{[1,k]} + o'$, por lo tanto, $s' \in \mathcal{Y}_{k+1}$.

  En conclusión, $\exists \bar{s} \in \mathring{S}(\mathcal{O})$ óptima tal que $s_{[1,k+1]} \in \mathcal{Y}_{k+1}$.

  Finalmente, para ver que $\exists s' \in \overset{(\preceq)}{\mathcal{Y}}_{k+1}$, puedo aplicar un argumento similar al de la Proposición 4.13, en donde la cadena de dominancia $\{s^i\}_{i \in I \subseteq \mathbb{N}}$ comenzará en $\bar{s}$ y $\forall i \in I : \{s^i\} \subseteq \mathcal{Y}_{k+1}$. Luego, como no se producen ciclos, tendremos que para algún $i \in I, s^i$ es óptima y $s_{[1,k+1]} \in \overset{(\preceq)}{\mathcal{Y}}_{k+1}$, como queríamos probar.

Finalmente, ver que $\#\big(\overset{(\preceq)}{\mathcal{Y}}(\mathcal{O})\big) = 1$ es análogo a la demostración de la Proposición 4.13.

□

Básicamente, el mecanismo de expansión iterativo planteado en 4.27 evita la generación de secuencias que tengan operaciones retrasadas. A su vez, si luego de descartar secuencias con retrasos, eliminamos las secuencias dominadas para sólo quedarnos con las secuencias minimales, también obtendremos un óptimo del problema.

Al reducir la cantidad de secuencias generadas por cada etapa, reducimos en gran número la cantidad de secuencias generadas en etapas posteriores. Eventualmente, obtendremos una secuencia óptima generando tanto los conjuntos $\overset{(\preceq)}{\mathcal{X}}(Q)$ como $\overset{(\preceq)}{\mathcal{Y}}(Q)$, pero mejoraremos los tiempos de procesamiento si utilizamos estos últimos.

En la Figura 16 se evidencia la reducción de secuencias por operaciones retrasadas dentro del árbol de la Figura 13. En ésta, podemos ver de azul las secuencias eliminadas por tener operaciones retrasadas.

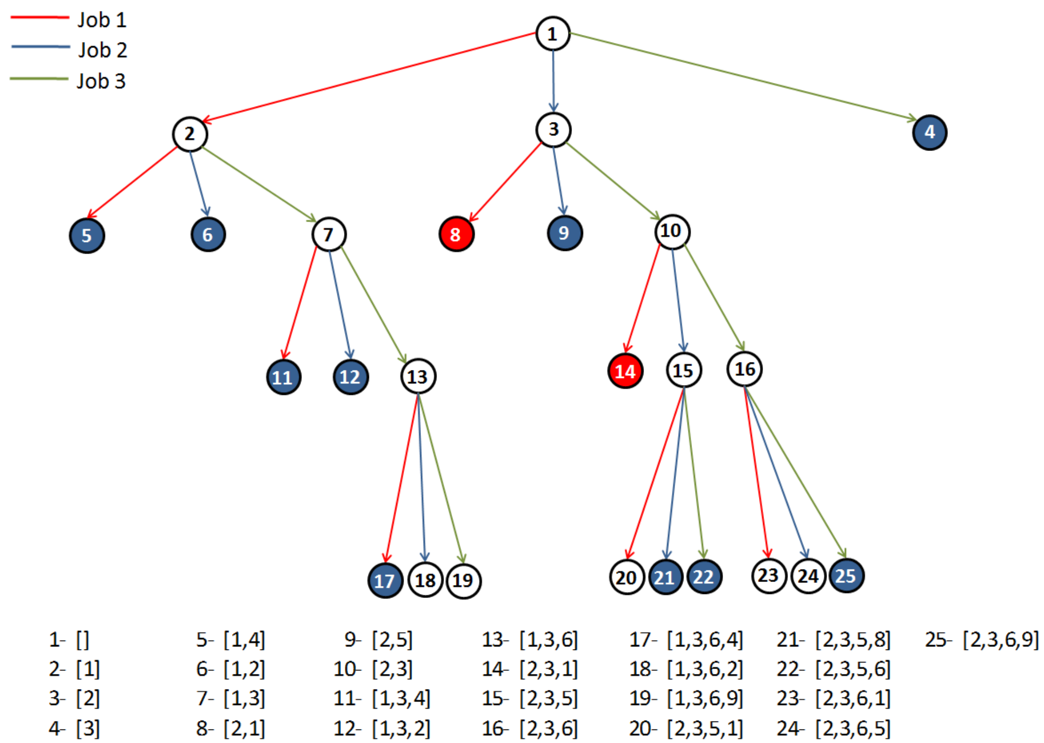


Figura 16: Árbol de búsqueda de $\mathcal{I}$ hasta la etapa 4 y eliminación de secuencias retrasadas

Es importante remarcar que nuestro criterio de eliminación de secuencias por operaciones retrasadas es distinto del utilizado en el apartado *State space reduction* de [6]. Más especificamente, nuestro criterio abarca un mayor abanico de posibilidades en donde cada secuencia no debe ser expandida con una operación que genere retrasos. Por ejemplo, según el criterio de [6], la operación o_4 no genera retraso para la secuencia $[o_1]$, mientras que según el criterio expuesto en la Definición 4.25 sí lo hace.

Por otro lado, el orden de eliminación de secuencias que aquí propone-

mos es el inverso del utilizado en [6]. Mientras allí se eliminan primero las secuencias dominadas, para luego descartar las secuencias con operaciones que generan retraso, nosotros optamos por descartar en primer instancia las secuencias con retrasos y recién entonces aplicar los criterios de dominancia. Esta diferencia obedece simplemente a una comodidad teórica: resulta más sencillo probar que el algoritmo encuentra una solución óptima con el orden de eliminación que aquí utilizamos. Cabe remarcar que la demostración presentada en [6] para la incorporación del descarte por operaciones con retraso depende de alguno de los argumentos que señalamos como erróneos, tanto en [7] como en el Apéndice de esta tesis.

4.5.2. Cotas inferiores y superiores

En esta subsección introduciremos un nuevo criterio de descarte que permite reducir la cantidad de secuencias mediante el uso de cotas. Este criterio no es considerado en [6].

Ya hemos mencionado en la Definición 1.6 que cualquier solución factible de un problema de minimización es una cota superior del problema. En nuestro caso, cada solución factible será una secuencia completa ordenada, y su $C_{\text{máx}}$ la cota superior. Por lo tanto, para obtener una cota superior debemos hallar secuencias completas y ordenadas en forma eficiente.

Para cumplir este cometido utilizaremos una heurística, que será presentada y explicada en el Capítulo 6. Por ahora, supondremos que tenemos alguna forma de obtener dichas secuencias.

Ya computadas estas secuencias completas, dado que el tiempo de finalización $C_{\text{máx}}$ de cada una de ellas es una cota superior de la solución óptima, calcularemos la cota superior CS como el mejor de estos tiempos. Así:

$$CS = \text{mín}\{C_{\text{máx}}(s)\}$$

con s una solución factible.

Además, llamaremos s_{CS} a alguna solución factible cuyo $C_{\text{máx}}$ resulta ser CS. Más formalmente:

$$s_{CS} = \arg\min\{C_{\text{máx}}(s)\}$$

Una vez obtenida una cota superior para el valor óptimo del problema, para cada secuencia parcial s, calcularemos un valor que acote inferiormente a todas las posibles completaciones de s. Llamaremos a esta cota $CI(s)$.

Luego, si resultase ser que $CI(s) \geq CS$, entonces cualquier completación s' de s tendrá un tiempo de finalización $C_{\text{máx}}(s')$ igual o peor que el de la secuencia s_{CS} correspondiente a la cota superior CS. Por lo tanto, podremos descartar la secuencia parcial s.

Una posibilidad para hallar $CI(s)$ es calcular una cota inferior al tiempo de finalización que dependa de cada máquina. Luego, $CI(s)$ será la máxima de todas estas cotas inferiores. Llamaremos $CI(s,i)$ a la cota inferior correspondiente a la máquina i. Así:

$$CI(s) = \max_{i \in \{1,\cdots,m\}} \{CI(s,i)\}$$

Para computar $CI(s,i)$, habrá que tener conocimiento tanto de las operaciones que faltan programar para la máquina i, como así también de las operaciones programadas. Las operaciones programadas servirán para obtener el mínimo tiempo al que pueden empezar a planificarse las operaciones restantes. Por otro lado, de estas últimas dependerá el tiempo mínimo de finalización de cada máquina.

Introduciremos la siguiente notación:

Definición 4.28. *Sea s una secuencia parcial. Notaremos:*

1. $\mathcal{O}(i) = \{o \in \mathcal{O} : m(o) = i\}$ *al conjunto de operaciones correspondientes a la máquina i.*

2. $\mathcal{O}_{\not s}(i) = \big(\mathcal{O}(i) \backslash q(s)\big)$ *al conjunto de operaciones correspondientes a la máquina i que aún no han sido programadas.*

3. $o(i,j) = \{o \in \mathcal{O} : m(o) = i \wedge j(o) = j\}$ *será la única operación del job j que se ejecute en la máquina i.*

Nuestra cota inferior para cada máquina $CI(s,i)$ constará de tres componentes: $CA(s,i)$, $TR(s,i)$ y $CO(s,i)$.

$CA(s,i)$ representará lo que denominaremos la *cabeza* de la cota inferior $CI(s,i)$. Este valor constituye una cota inferior al tiempo de inicialización de la primera operación que resta programarse en la máquina i. En caso de no quedar ninguna operación por programar, representará el tiempo de finalización de la máquina i.

Para computar $CA(s,i)$, para cada operación $o \in \mathcal{O}_{\not s}(i)$, tendremos dos casos: $o \in \varepsilon(q(s))$ u $o \notin \varepsilon(q(s))$.

En el primer caso, el tiempo mínimo está dado por $\xi(s,o) - p(o)$. Como ya hemos observado, en cualquier completación ordenada de s, el funcional $\xi(s,o)$ representa una cota inferior al tiempo de finalización de o. Luego, como el tiempo de inicio de cualquier operación se puede calcular como la diferencia entre su tiempo de finalización y de ejecución, tendremos que $\xi(s,o) - p(o)$ será una cota inferior del tiempo en que se pueda programar o.

Para clarificar lo mencionado, si por ejemplo suponemos que $s = [o_1, o_3]$, para la operación $o_2 \in \big(\mathcal{O}_{\not s}(1) \cap \eta(s)\big)$, este tiempo será

$$\xi(s, o_2) - p(o_2) = \psi(s, o_2) + p(o_2) - p(o_2) = \psi(s, o_2) = 2$$

En el segundo caso, notemos que hay una serie de operaciones del mismo job $j(o)$ que aún no han sido programadas, por lo que o no podrá ser ejecutada hasta que estas operaciones hayan finalizado. Luego, para hallar una cota inferior al tiempo de inicio de o, supondremos que las operaciones *precedentes* se ejecutarán en serie, una detrás de la otra. La primera operación precedente o' será la correspondiente al job $j(o)$ que se encuentre en el conjunto $\varepsilon(q(s))$, y su tiempo de inicio será nuevamente $\xi(s, o') - p(o')$.

Si tomamos el mismo ejemplo de antes, tendremos que, para $s = [o_1, o_3]$ y la operación $o_8 \in \big(\mathcal{O}_{\not s}(3) \cap (\mathcal{O} \backslash \varepsilon(q(s)))\big)$:

$$\xi(s, o_2) - p(o_2) + p(o_2) + p(o_5) = \psi(s, o_2) + p(o_2) + p(o_5) = 2 + 2 + 1 = 5$$

Finalmente, computaremos $CA(s, i)$ como el mínimo sobre todos estos tiempos. La siguiente definición formalizará la dicho anteriormente:

Definición 4.29. *Sea $Q \subseteq \mathcal{O}$ factible y $s \in \mathring{S}(Q)$. Definiremos:*

1. *$CA_s(i, j) = \{o \in \mathcal{J}_{pre}(o(i, j))\}$ será el conjunto de operaciones del job j que preceden a la operación de dicho job para la máquina i. Además, $CA_{\not s}(i, j) = CA_s(i, j) \backslash Q$.*

2. *Para cada operación $o \in \varepsilon(Q) : j(o) = j$, definiremos:*

$$\psi_j(s, o) = \begin{cases} \psi(s, o), \text{ si } o \in \eta(s) \\ C_{\text{máx}}(s), \text{ en caso contrario} \end{cases}$$

3. *Para cada operación $o(i, j) \in \mathcal{O}_{\not s}(i)$, sea $o \in \varepsilon(Q) : j(o) = j$. Definiremos*

$$CA(s, i, j) = \psi_j(s, o) + \sum_{\{o' \in CA_{\not s}(i,j)\}} p(o')$$

4. $CA(s, i) = \begin{cases} C_{\text{máx}}(\psi_s, i), \text{ si } \mathcal{O}_{\not s}(i) = \emptyset \\ \text{mín}_{o \in \mathcal{O}_{\not s}(i)} \{CA(s, i, j(o))\}, \text{ si } \mathcal{O}_{\not s}(i) \neq \emptyset \end{cases}$

La segunda componente de $CI(s, i)$, será $TR(s, i)$ que representará el *tiempo remanente* que la máquina i debe trabajar para programar las operaciones restantes y será igual a la adición de los tiempos de ejecución de las operaciones no programadas $o \in O_{\not s}(i)$.

Al sumar estos tiempos estamos asumiendo que la máquina no descansa, por lo que $TR(s,i)$ es una cota inferior del tiempo empleado hasta que no queden más operaciones en $\mathcal{O}_{\not s}(i)$. De no haber más operaciones que programar en la máquina i, $TR(s,i)$ será lógicamente nulo. Más formalmente:

Definición 4.30. *Sea $Q \subseteq \mathcal{O}$ factible y $s \in \mathring{S}(Q)$. Definiremos*

$$TR(s,i) = \sum_{\{o \in \mathcal{O}_{\not s}(i)\}} p(o)$$

Por último, $CO(s,i)$ será la *cola* de la máquina i. Es decir: $CO(s,i)$ será una cota inferior para el tiempo que llevará la ejecución de todas las operaciones después de haberse concluido el uso de la máquina i.

En el caso de que la máquina i no tenga entre sus operaciones no programadas ninguna operación que finalice un job, entonces para cualquier orden de estas operaciones todavía restarán planificar otras operaciones en otras máquinas. Luego, en este caso, podremos tomar a la cola $CO(s,i)$ como el mínimo del tiempo total de finalización de cada job, suponiendo ya programadas todas las operaciones de la máquina i.

Cabe destacar que la cola $CO(s,i)$ se calculará sólo para las operaciones de $\mathcal{O}_{\not s}(i)$, pues, para cada operación $o \in \big(q(s) \cap \mathcal{O}(i)\big)$ que ya ha sido programada en la máquina i, la cota inferior del tiempo de finalización del job correspondiente $j(o)$ esta contemplada dentro de la cabeza de la cota inferior para la máquina $CS(i')$, donde i' será la máquina de la próxima operación de dicho job $j(o)$, i.e, $i' = m(o')$, con $o' \in \varepsilon(Q) : j(o') = j(o)$. Para una demostración formal de esto, veáse la Proposición 4.18.

La próxima definición formalizará la noción de cola:

Definición 4.31. *Sea $Q \subseteq \mathcal{O}$ factible y $s \in \mathring{S}(Q)$. Definiremos:*

1. *$CO(i,j) = \{o \in \mathcal{J}(o(i,j)) : indice(o) > indice(o(i,j))\}$ será el conjunto de operaciones de cada job que resulta posterior a la operación de dicho job para la máquina i. Además, $CO_{\not s}(i,j) = CO(i,j) \backslash Q$.*
2. *$CO(s,i) = \min_{j \in \{1,\cdots,n\}} \{\sum_{\{o \in CO_{\not s}(i,j)\}} p(o)\}$*

Una vez explicado brevemente el sentido de cada componente, $CI(s,i)$ será la suma de éstas:

$$CI(s,i) = CA(s,i) + TR(s,i) + CO(s,i)$$

A continuación haremos la demostración formal de que $CI(s)$ es efectivamente una cota inferior para la secuencia s. Para esto, veremos que dada una secuencia parcial s, $CI(s)$ será menor o igual que $C_{\text{máx}}(s')$, $\forall s' \in \mathring{Comp}(s)$.

Definición 4.32. *Sea $\psi \in \Psi$ un schedule activo factible, y s_ψ su secuencia asociada. $\forall i \in \{1, \cdots, m\}$, definiremos:*

1. $[o_1^{(\psi,i)}, \cdots, o_n^{(\psi,i)}]$*, una secuencia $s^{(i)}$ que representa el orden en que las operaciones fueron programadas en la máquina i. Así, $\forall r, l = 1, \cdots, n$: $o_r^{(\psi,i)} < o_l^{(\psi,i)} \Leftrightarrow \psi(o_r^{(\psi,i)}) < \psi(o_l^{(\psi,i)})$.*

2. $C_{\text{máx}}(\psi, i) = \psi(o_n^{(\psi,i)}) + p(o_n^{(\psi,i)})$*, es el tiempo de finalización de la máquina i.*

3. $C_{min}(\psi, i) = \psi(o_1^{(\psi,i)})$ *es el tiempo de inicio de la máquina i.*

4. *Si s es una subsecuencia de s_ψ (parcial o total), definiremos:*

$$C_{min}(s,i) = \begin{cases} C_{\text{máx}}(\psi_s, i), \ \textit{si } \mathcal{O}(i) \subseteq q(s) \\ \text{mín}_{\{r: o_r^{(\psi,i)} \notin q(s)\}} \{\psi(o_r^{(\psi,i)})\}, \ \textit{en caso contrario} \end{cases}$$

el tiempo de inicio de la primera operación de la máquina i que no esta en $q(s)$.

Respecto de la anterior notación, es importante dejar en claro que el índice r de $o_r^{(\psi,i)}$ no guarda ninguna relación con el índice de las operaciones. Para denotar el índice de $o_r^{(\psi,i)}$ escribiremos $indice(o_r^{(\psi,i)})$.

Proposicion 4.17. *Sea $Q \subseteq \mathcal{O}$ factible y $s \in \mathring{S}(Q)$. Entonces, $CI(s)$ es una cota inferior de s, es decir:*

$$\forall \psi \in \mathring{Comp}_\psi(s) : CI(s) \leq C_{\text{máx}}(\psi)$$

Demostración. Sea $\psi \in \mathring{Comp}_\psi(s)$, y sea $i \in \{1, \cdots, m\}$ el índice de la máquina i-ésima.

Dividiremos la demostración en los siguientes ítems:

1. $CA(s,i) \leq C_{min}(s,i)$

2. $TR(s,i) \leq C_{\text{máx}}(\psi, i) - C_{min}(s,i)$

3. $CO(s,i) \leq C_{\text{máx}}(\psi) - C_{\text{máx}}(\psi, i)$

4. $CI(s,i) \leq C_{\text{máx}}(\psi)$

5. $CI(s) \leq C_{\text{máx}}(\psi)$

1. Si $\mathcal{O}_{\not s}(i) = \emptyset$, por definición resulta que:

$$CA(s,i) = C_{\text{máx}}(\psi_s, i) = C_{min}(s,i)$$

por lo que el resultado resulta válido.

Supongamos que $\mathcal{O}_{\not s}(i) \neq \emptyset$. Luego,

$$CA(s,i) = \min_{o' \in \mathcal{O}_{\not s}(i)} \{CA(s,i,j(o'))\}$$

Sea $o \in \mathcal{O}_{\not s}(i)$, y sea $j = j(o)$. Separaremos en dos casos:

- $CA_{\not s}(i,j) = \emptyset$:
En este caso, de la definición de $CA_{\not s}(i,j)$ y del hecho de que $o \in \mathcal{O}_{\not s}(i)$ se infiere que $o \in \varepsilon(Q)$
Por lo visto en la Proposición 4.7, ítem 6, tenemos que $\psi(o) \geq \psi_j(s,o)$. Como $CA_{\not s}(i,j) = \emptyset$, tendremos que:

$$\begin{aligned}\psi(o) \geq \psi_j(s,o) &= \psi_j(s,o) + \sum_{o' \in CA_{\not s}(i,j)} p(o') \\ &= CA(s,i,j) \geq CA(s,i)\end{aligned}$$

- $CA_{\not s}(i,j) \neq \emptyset$:
Como ψ es factible, se deben programar todas las operaciones de $CA_{\not s}(i,j)$ antes de que se pueda programar o. Escribiremos $CA_{\not s}(i,j) = \{o_1^{(i,j)}, \cdots, o_k^{(i,j)}\}$ de forma que las operaciones estén ordenadas en forma creciente según su *indice*. Así:

$$\forall r, l = 1, \cdots, k : o_r^{(i,j)} < o_l^{(i,j)} \Leftrightarrow indice(o_r^{(i,j)}) < indice(o_l^{(i,j)})$$

Por lo tanto, tendremos que:

$$\forall r = 1, \cdots, k-1 : \psi(o_r^{(i,j)}) + p(o_r^{(i,j)}) \leq \psi(o_{r+1}^{(i,j)}),$$

con $o_1^{(i,j)} \in \varepsilon(Q)$.
En forma análogo a lo visto en el caso $CA_{\not s}(i,j) = \emptyset$, resulta que $\psi(o_1^{(i,j)}) \geq \psi_j(s, o_1^{(i,j)})$.
Juntando todo, obtendremos la siguiente desigualdad:

$$\begin{aligned}\psi(o) &\geq \psi(o_k^{(i,j)}) + p(o_k^{(i,j)}) \geq \psi(o_{k-1}^{(i,j)}) + p(o_{k-1}^{(i,j)}) + p(o_k^{(i,j)}) \\ &\geq \psi(o_1^{(i,j)}) + p(o_1^{(i,j)}) + \cdots + p(o_k^{(i,j)}) \\ &= \psi(o_1^{(i,j)}) + \sum_{o' \in CA_{\not s}(i,j)} p(o') \geq \psi_j(s, o_1^{(i,j)}) + \sum_{o' \in CA_{\not s}(i,j)} p(o') \\ &= CA(s,i,j) \geq CA(s,i)\end{aligned}$$

Luego, en ambos casos resulta que $\psi(o) \geq CA(s,i)$. Como $o \in (\mathcal{O}_{\not s}(i)$ es arbitraria, entonces resultará que:

$$C_{min}(s,i) = \min_{o \in \mathcal{O}_{\not s}(i)} \{\psi(o)\} \geq CA(s,i)$$

como queríamos ver.

2. Si $\mathcal{O}_{\not s}(i) = \emptyset$, entonces $TR(s,i) = 0$. Luego, como

$$C_{\text{máx}}(\psi,i) - C_{min}(s,i) \geq 0,$$

es trivial que $TR(s,i) \leq C_{\text{máx}}(\psi,i) - C_{min}(s,i)$.

Supongamos que $\mathcal{O}_{\not s}(i) \neq \emptyset$. Sea r tal que $C_{min}(s,i) = \psi(o_r^{(\psi,i)})$. Como ψ es factible, tendremos que:

$$\forall l = 1, \cdots, n-1 : \psi(o_l^{(\psi,i)}) < \psi(o_{l+1}^{(\psi,i)})$$

Utilizando la anterior desigualdad, obtenemos:

$$C_{\text{máx}}(\psi,i) - C_{min}(s,i) = \psi(o_n^{(\psi,i)}) + p(o_n^{(\psi,i)}) - C_{min}(s,i) \geq$$

$$\geq \psi(o_{n-1}^{(\psi,i)}) + p(o_{n-1}^{(\psi,i)}) + p(o_n^{(\psi,i)}) - C_{min}(s,i) \geq \cdots \geq$$

$$\geq \psi(o_r^{(\psi,i)}) + p(o_r^{(\psi,i)}) + \cdots + p(o_n^{(\psi,i)}) - C_{min}(s,i) = (*)$$

Ahora, como $C_{min}(s,i) = \psi(o_r^{(\psi,i)})$:

$$(*) = C_{min}(s,i) + p(o_r^{(\psi,i)}) + \cdots + p(o_n^{(\psi,i)}) - C_{min}(s,i) =$$

$$= p(o_r^{(\psi,i)}) + \cdots + p(o_n^{(\psi,i)}) = TR(s,i),$$

de donde se deduce la desigualdad.

3. Sea $j = 1, \cdots, n$. Partiremos en los siguientes dos casos:

- $CO_{\not s}(i,j) = \emptyset$. En este caso, es claro que $\sum_{\{o \in CO_{\not s}(i,j)\}} p(o) = 0$. Luego, resulta trivial que

$$C_{\text{máx}}(\psi) \geq C_{\text{máx}}(\psi,i) = C_{\text{máx}}(\psi,i) + \sum_{\{o \in CO_{\not s}(i,j)\}} p(o)$$

- $CO_{\not s}(i,j) \neq \emptyset$. Escribiremos a $CO_{\not s}(i,j) = \{o_1^{(i,j)}, \cdots, o_k^{(i,j)}\}$ de forma que las operaciones de $CO_{\not s}(i,j)$ se encuentren ordenadas por el *indice*. De esta forma, tendremos que:

$$\forall r,l = 1, \cdots, k : o_r^{(i,j)} < o_l^{(i,j)} \Leftrightarrow indice(o_r^{(i,j)}) < indice(o_l^{(i,j)})$$

Luego, como ψ es factible, son válidas:

- $\forall r = 1, \cdots, k-1 : \psi(o_r^{(i,j)}) + p(o_r^{(i,j)}) \leq \psi(o_{r+1}^{(i,j)})$
- $C_{\text{máx}}(\psi, i) = \psi(o_n^{(\psi,i)}) + p(o_n^{(\psi,i)}) \leq \psi(o_1^{(i,j(o_n^{(\psi,i)}))})$

Entonces,

$$C_{\text{máx}}(\psi) = \max_{o \in \mathcal{O}}\{\psi(o) + p(o)\} \geq \max_{j=1,\cdots,n}\{\psi(o_k^{(i,j)}) + p(o_k^{(i,j)})\} \geq$$

$$\geq \max_{j=1,\cdots,n}\{\psi(o_{k-1}^{(i,j)}) + p(o_{k-1}^{(i,j)}) + p(o_k^{(i,j)})\} \geq \cdots \geq$$

$$\geq \max_{j=1,\cdots,n}\{\psi(o_1^{(i,j)}) + p(o_1^{(i,j)}) + \cdots + p(o_k^{(i,j)})\} = (*)$$

En particular, para $j = j(o_n^{(\psi,i)})$ tendremos que:

$$(*) \geq \psi(o_1^{(i,j(o_n^{(\psi,i)}))}) + p(o_1^{(i,j(o_n^{(\psi,i)}))}) + \cdots + p(o_k^{(i,j(o_n^{(\psi,i)}))}) \geq$$

$$\geq C_{\text{máx}}(\psi, i) + p(o_1^{(i,j(o_n^{(i)}))}) + \cdots + p(o_k^{(i,j(o_n^{(i)}))}) =$$

$$C_{\text{máx}}(\psi, i) + \sum_{\{o \in CO_{\not{s}}(i,j)\}} p(o)$$

En ambos casos llegamos a la misma desigualdad.

Ahora, por definición, tendremos que $\forall j = 1, \cdots, n : CO(s,i) \leq \sum_{\{o \in CO_{\not{s}}(i,j)\}} p(o)$ de donde se deduce que:

$$C_{\text{máx}}(\psi) \geq C_{\text{máx}}(\psi, i) + \sum_{\{o \in CO_{\not{s}}(i,j)\}} p(o) \geq C_{\text{máx}}(\psi, i) + CO(s,i)$$

como queríamos ver.

4. De los anteriores ítems y de la definición de $CI(s,i)$ se deduce la desigualdad:

$$CI(s,i) = CA(s,i) + TR(s,i) + CO(s,i) \leq$$

$$\leq C_{min}(s,i) + (C_{\text{máx}}(\psi, i) - C_{min}(s,i)) + CO(s,i) =$$

$$= C_{\text{máx}}(\psi, i) + CO(s,i) \leq C_{\text{máx}}(\psi)$$

5. Como los ítems anteriores se han demostrado para una máquina i arbitraria y un schedule $\psi \in \mathring{Comp}_{\psi}(s)$ arbitrario, entonces la desigualdad del ítem 4) es válida $\forall i = 1, \cdots, m, \forall \psi \in \mathring{Comp}_{\psi}(s)$. Luego:

$$CI(s) = \max_{i \in \{1,\cdots,m\}}\{CI(s,i)\} \leq C_{\text{máx}}(\psi)$$

□

Para fijar ideas, expondremos el siguiente ejemplo.

Ejemplo 4.4. *En este ejemplo se consideraremos la secuencia* $s = [o_1, o_3]$ *correspondiente a la instancia* $\mathcal{I}$*. En la Figura 17 se puede ver gráficamente cómo se calcularía la cota inferior para dicha secuencia.*

Además, en el Cuadro 2 se muestran cuáles son los valores para $CA(s,i)$*,* $TR(s,i)$*,* $CO(s,i)$ *y* $CI(s,i)$ *para* $i = 1, 2, 3$*, así como también el job que para el que se realizan* $CA(s,i)$ *y* $CO(s,i)$*.*

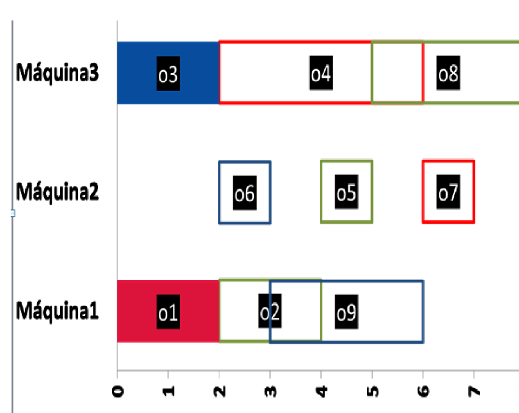


Figura 17: Posibles expansiones con cada job para la secuencia s

Máquina	CA(s,i)	Job de $CA(s,i)$	$TR(s,i)$	$CO(s,i)$	Job de $CO(s,i)$	$CI(s,i)$
1	2	2	5	0	3	**7**
2	3	3	3	0	1	**6**
3	2	1	7	0	2	**9**

Cuadro 2: Cómputo de $CI(s)$

Hasta aquí hemos explicado y probado que efectivamente se pueden descartar secuencias parciales s cuya cota inferior sea mayor que la cota superior CS. En lo que resta de esta subsección veremos cómo incorporar a nuestro esquema iterativo este criterio de eliminación.

Como nuestro esquema iterativo utiliza para expandir las operaciones en $\widehat{\eta(s)}$, lo más natural sería que la eliminación por cotas se produzca luego de la expansión. No obstante, debemos definir si sucederá antes ó después de la eliminación por dominancia.

La siguiente proposición y corolario nos mostrarán que el orden relativo entre la eliminación por dominancia y por cotas es indiferente.

Proposicion 4.18. *Sea* $Q \subset \mathcal{O}$ *factible,* $s_1, s_2 \in X(Q)$*, dos secuencias parciales tales que* $s_2 \preceq s_1$*. Además, sea* CS *una cota superior conocida.*

Luego, si $CS \leq CI(s_2)$*, entonces* $CS \leq CI(s_1)$*.*

Demostración. Sea $i \in \{1, \cdots, m\}$ alguna máquina.

En principio, como $q(s_1) = q(s_2) = Q$, se deduce de la Definición 4.30 que $TR(s_2, i) = TR(s_1, i)$. Analogamente, de la Definición 4.31, $CO_{\not s_1}(i,j) = CO_{\not s_2}(i,j)$, por lo que $CO(s_1, i) = CO(s_2, i)$.

Tendremos dos casos:

1. $\mathcal{O}(i) \backslash Q \neq \emptyset$

   Luego, por Definición 4.29 tendremos que:

$$CA(s_r, i) = \min_{j \in \{1, \cdots, n\}} \{\psi_j(s_r, o) + \sum_{o \in CA_{\not s_r}(i,j)} p(o)\}$$

   Nuevamente, de la definición se deduce que $CA_{\not s_1}(i,j) = CA_{\not s_2}(i,j)$, por lo que

$$\sum_{o \in CA_{\not s_1}(i,j)} p(o) = \sum_{o \in CA_{\not s_2}(i,j)} p(o) \tag{23}$$

   Observemos que $\forall o \in \varepsilon(Q), j = j(o) : \psi_j(s_r, o) = \xi(s_r, o) - p(o)$. Por lo tanto, de que $s_2 \preceq s_1$ se deduce que

$$\psi_j(s_2, o) = \xi(s_2, o) - p(o) \leq \xi(s_1, o) - p(o) = \psi_j(s_1, o) \Rightarrow$$

$$\Rightarrow \psi_j(s_2, o) \leq \psi_j(s_1, o) \tag{24}$$

   De (23) y (24) tendremos que

$$\begin{aligned} \psi_j(s_2, o) + \sum_{o \in CA_{\not s_2}(i,j)} p(o) &= \psi_j(s_2, o) + \sum_{o \in CA_{\not s_1}(i,j)} p(o) \\ &\leq \psi_j(s_1, o) + \sum_{o \in CA_{\not s_1}(i,j)} p(o) \end{aligned}$$

   Luego, tomando el mínimo se infiere que $CA(s_2, i) \leq CA(s_1, i)$.

   Finalmente, como i es arbitraria, obtendremos que

$$CI(s_2, i) = CA(s_2, i) + TR(s_2, i) + CO(s_2, i) \leq$$

$$\leq CA(s_1, i) + TR(s_1, i) + CO(s_1, i) = CI(s_1, i) \Rightarrow$$
$$\Rightarrow CI(s_2) = \max_{i \in \{1,\cdots,m\}} \{CI(s_2, i)\} \leq \max_{i \in \{1,\cdots,m\}} \{CI(s_1, i)\} = CI(s_1)$$

de donde se deduce que $CS \leq CI(s_2) \leq CI(s_1)$, como queríamos ver.

2. $\mathcal{O}(i) \backslash Q = \emptyset$

   En este caso, tendremos que $O_{\not{s}_1}(i) = \emptyset$, de donde podemos deducir que $TR(s_1, i) = 0$ y $\forall j = 1, \cdots, n : CO_{\not{s}_1}(i, j) = \emptyset \Rightarrow CO(s_1, i) = 0$

   Por otro lado, de la Definición 4.29, tendremos que $CA = C_{\text{máx}}(\psi_s, i)$, que es el tiempo en que finalizo la última tarea de la máquina i.

   En principio, tendremos

$$C_{\text{máx}}(\psi_{s_1}, i) \leq C_{\text{máx}}(\psi_{s_1}) = C_{\text{máx}}(s_1) \tag{25}$$

   Por otro lado, como s_1 no es completa, entonces $\exists i' \in \{1, \cdots, m\} : O_{\not{s}_1}(i') \neq \emptyset$. Luego, de las Definiciones 4.30 y 4.31 se infieren que:

   Sea $o \in \mathcal{O}_{\not{s}_1}(i')$. Entonces, utilizando la Proposición 4.7 y la Definición 4.29 tendremos que

$$\forall j = 1, \cdots, n : \psi_j(s_1, o) = \xi(s_1, o) - p(o) \geq C_{\text{máx}}(s_1) - p(o) \Rightarrow$$
$$\Rightarrow CA(s_1, i') \geq \min_{\{j : o \in \mathcal{O}_{\not{s}}(i') \wedge j = j(o)\}} \{\psi_j(s_1, o)\} \geq C_{\text{máx}}(s_1) - p(o) \tag{26}$$

   Además, como $o \in \mathcal{O}_{\not{s}}(i')$, entonces

$$TR(s_1, i') \geq p(o) \tag{27}$$

   De los (26) y (27) deducimos que

$$CI(s_1, i') = CA(s_1, i') + TR(s_1, i') + CO(s_1, i') \geq$$
$$\geq CA(s_1, i') + TR(s_1, i') \geq C_{\text{máx}}(s_1) - p(o) + p(o) = C_{\text{máx}}(s_1) \tag{28}$$

   Luego, de (25) y (28) tendremos que

$$CI(s_1, i) \leq CI(s_1, i')$$

   Como i es arbitraria, se deduce que

$$CI(s_1) = \max_{i \in \{1,\cdots,m\}} \{CI(s_1, i)\} = \max_{i \in \{1,\cdots,m\} : \mathcal{O}_{\not{s}_1}(i) \neq \emptyset} \{CI(s_1, i)\}$$

   por lo que la demostración se remite al caso 1.

□

Definición 4.33. *Sea $Q \subseteq \mathcal{O}$ factible y CS una cota superior conocida. Sea $X(Q) \subseteq \mathring{S}(Q)$ un conjunto de secuencias parciales y ordenadas. Definiremos*

$$\overset{(CS)}{X}(Q) = \{s \in X(Q) : CS \leq CI(s)\}$$

Corolario 4.2. *Sea $Q \subseteq \mathcal{O}$ factible y CS una cota superior conocida. Entonces:*

$$\overset{(\preceq)}{\overset{(CS)}{X}}(Q) = \overset{(CS)}{\overset{(\preceq)}{X}}(Q)$$

Demostración.

- $\overset{(\preceq)}{\overset{(CS)}{X}}(Q) \subseteq \overset{(CS)}{\overset{(\preceq)}{X}}(Q)$:

  Sea $s \in \overset{(\preceq)}{\overset{(CS)}{X}}(Q)$. Luego, se infiere que $CI(s) < CS$, y que $\nexists s' \in \overset{(CS)}{X}(Q)$ tal que $s' \preceq s$.

  Supongamos que $s \notin \overset{(\preceq)}{X}(Q)$. Entonces, por definición, tendremos que $\exists s' \in X(Q)$ tal que $s' \preceq s$. De lo anterior se deduce que $s' \notin \overset{(CS)}{X}(Q)$, por lo que $CS \leq CI(s')$. Pero, si este fuese el caso, dada la Proposición 4.18 y que $s' \preceq s$, entonces $CS \leq CI(s)$. Esto contradice las hipótesis, por lo que se concluye que $s \in \overset{(\preceq)}{X}(Q)$.

  Por último, como $CI(s) < CS$, se tiene que $s \in \overset{(CS)}{\overset{(\preceq)}{X}}(Q)$, como se quería ver.

- $\overset{(CS)}{\overset{(\preceq)}{X}}(Q) \subseteq \overset{(\preceq)}{\overset{(CS)}{X}}(Q)$:

  Sea $s \in \overset{(CS)}{\overset{(\preceq)}{X}}(Q)$. Luego, $CI(s) < CS$ y $\not\exists s' \in X(Q) : s' \preceq s$.

  De la primera implicación se deduce que $s \in \overset{(CS)}{X}(Q)$. Además, si $\nexists s' \in X(Q)$ tal que $s' \preceq s$, como $\overset{(CS)}{X}(Q) \subseteq X(Q)$, se infiere que $\nexists s' \in \overset{(CS)}{X}(Q)$ tal que $s' \preceq s$. Luego, $s \in \overset{(\preceq)}{\overset{(CS)}{X}}(Q)$

□

Como indica el Corolario 4.2, el orden relativo de eliminación por dominancia y por cotas es indiferente. Luego, lo más sensato sería que prevaleciera el proceso con menor complejidad.

Más adelante veremos que en la gran mayoría de los casos resulta menos costosa la eliminación por cotas que por dominancia. Esto impulsa la siguiente definición:

Definición 4.34. *Sea $Q \subseteq \mathcal{O}$ factible, y $X(Q) \subseteq \mathring{S}(Q)$ un conjunto de secuencias ordenadas. Definiremos:*

$$\widetilde{X(Q)} = \overset{(\preceq)}{\overset{(CS)}{X(Q)}}$$

Ya estamos en condiciones de incorporar a nuestro esquema iterativo la eliminación por cotas.

Definición 4.35.

$$\mathcal{Z}(Q) = \{[o_j]\},\ si\ Q = \{o_j \in \widehat{\eta([])}\}$$

$$\mathcal{Z}(Q) = \bigcup_{o\in\lambda(Q)} \ \bigcup_{s\in\widetilde{\mathcal{Z}(Q\backslash\{o\})}\ con\ o\in\widehat{\eta(s)}} \{s+o\},\ si\ Q\ factible,\ |Q|>1$$

Finalmente, la siguiente proposición justifica que los conjuntos generados a través de Definición 4.35 sirven para obtener una solución óptima del JSSP.

Proposicion 4.19. *Alguno de los siguientes enunciados es verdadero:*

1. *s_{CS} es una secuencia óptima.*
2. *$\widetilde{\mathcal{Z}(\mathcal{O})} = \{s\}$, con s una secuencia completa, ordenada y óptima.*

Demostración. Observemos que $\forall Q \subseteq \mathcal{O} : \mathcal{Z}(Q) \subseteq Y(Q)$. En efecto, si $|Q| = 1$, tenemos que

$$\overset{(CS)}{\mathcal{Z}(Q)} = \overset{(CS)}{Y(Q)}$$

Por otro lado, del Corolario 4.2 se infiere que

$$\widetilde{\mathcal{Z}(Q\backslash\{o\})} = \overset{(\preceq)}{\overset{(CS)}{\mathcal{Z}(Q\backslash\{o\})}} = \overset{(CS)}{\overset{(\preceq)}{\mathcal{Z}(Q\backslash\{o\})}} \subseteq \overset{(CS)}{\overset{(\preceq)}{\mathcal{Y}(Q\backslash\{o\})}} \subseteq \overset{(\preceq)}{\mathcal{Y}(Q\backslash\{o\})}$$

Mirando cómo se construyen los conjuntos en la Definición 4.35 y utilizando un argumento inductivo se deduce el resultado.

Luego, tendremos que $\widetilde{\mathcal{Z}(\mathcal{O})} \subseteq \overset{(\preceq)}{\mathcal{Y}}(\mathcal{O} = \{s\}$, con s óptima.

En el caso de que $\widetilde{\mathcal{Z}(\mathcal{O})} = \{s\}$, tendremos que la segunda propiedad es verdadera.

Por el contrario, si $\widetilde{\mathcal{Z}(\mathcal{O})} = \emptyset$, entonces debe ser que para algún $k \in \{1, 2, \cdots, nm-1\} : s_{[1,k]} \notin \mathcal{Z}(\widetilde{q(s_{[1,k]})})$. Como $s_{[1,k]} \in \mathcal{Y}(q(\overset{(\preceq)}{s}_{[1,k]}))$, entonces, $s_{[1,k]}$ debe haber sido eliminado por cotas. Luego, tendremos que $CS \leq CI(s_{[1,k]})$.

De aquí y del hecho de que $s \in \overset{\circ}{Comp}(s_{[1,k]})$, por la Proposición 4.17, se deduce que $C_{\text{máx}}(s_{CS}) \leq C_{\text{máx}}(s)$, por lo que s_{CS} es óptima. Luego, vale el primer enunciado $\square$

5. Implementación

En esta subsección implementaremos un algoritmo en base a lo visto en el Capítulo 4. Además, haremos un breve estudio de la complejidad de dicho algoritmo.

5.1. Construcción del algoritmo

Nuestro algoritmo construirá, en forma iterativa, los conjuntos descritos en la Definición 4.35. La Proposición 4.19, garantiza que este algoritmo es exacto.

Básicamente, el algoritmo constará de dos fases principales:

1. La fase de expansión, en donde se expandirán todas las secuencias correspondientes a una misma etapa $\widetilde{\mathcal{Z}_k}$, constuyendo así una nueva etapa de secuencias $\mathcal{Z}_{k+1}$. En esta fase, para cada secuencia $s \in \widetilde{\mathcal{Z}_k}, o \in \widehat{\eta(s)}$, se construirá $s+o$. Al terminar de expandir s con todas las operaciones en $\widehat{\eta(s)}$, s será descartada.

2. Una fase de eliminación, en la cual se procederán a descartar secuencias mediante los criterios de dominancia y cotas. Así, una vez construída la nueva etapa $\mathcal{Z}_{k+1}$ en la fase de expansión, procederemos a computar el conjunto de secuencias $\widetilde{\mathcal{Z}_{k+1}}$. Para ello, se agrupan las secuencias en $\mathcal{Z}_{k+1}$ que correspondan al mismo conjunto de operaciones Q, y se computa $\widetilde{\mathcal{Z}(Q)}$ según la Definición 4.34.

De esta forma, el algoritmo alternará dichas fases desde $k = 1$ hasta $k = nm$, ó se detendrá antes si $\widetilde{\mathcal{Z}_{k+1}}$ resulta vacío.

A continuación expondremos el algoritmo en pseudocódigo.

Algorithm JSSP

Input: Instancia:

$$I = \begin{cases} Cantidad\ de\ jobs = n \\ Cantidad\ de\ máquinas = m \\ Operaciones = [o_1, o_2, \cdots, o_{nm}] \\ Tiempo\ de\ ejecución = [p_1, p_2, \cdots, p_{nm}] \end{cases}$$

Output: Una solución óptima del JSSP.

1: $s \leftarrow []$
2: $S \leftarrow [s]$
3: $solParcial \leftarrow heuristica(S)$
4: $CS \leftarrow tiempoFinal(solParcial)$
5: $k \leftarrow 0$
6: **while** $k < nm$ **do**
7: $\quad L \leftarrow long(S)$
8: $\quad$ **for** $l = 0 : L - 1$ **do**
9: $\quad\quad s \leftarrow S(0)$
10: $\quad\quad \varepsilon \leftarrow \varepsilon(s)$
11: $\quad\quad \psi \leftarrow \psi(s, \varepsilon)$
12: $\quad\quad \eta \leftarrow \eta(s, \psi, \varepsilon)$
13: $\quad\quad \widehat{\eta} \leftarrow \widehat{\eta}(s, \psi, \eta)$
14: $\quad\quad S \leftarrow S + expandir(s, \psi, \widehat{\eta})$
15: $\quad\quad S \leftarrow S - S(0)$
16: $\quad$ **end for**
17: $\quad L \leftarrow long(S)$
18: $\quad l \leftarrow 0$
19: $\quad$ **while** $l < L$ **do**
20: $\quad\quad s \leftarrow S(l)$
21: $\quad\quad \varepsilon \leftarrow \varepsilon(s)$
22: $\quad\quad \psi \leftarrow \psi(s, \varepsilon)$

23: $\eta \leftarrow \eta(s, \psi, \varepsilon)$
24: $cotaInferior(s) \leftarrow actualizarCota(s, \varepsilon, \psi, \eta)$
25: **if** $cotaInferior(s) \geq CS$ **then**
26: $S \leftarrow S - S(l)$
27: $L \leftarrow L - 1$
28: **end if**
29: **end while**
30: $L \leftarrow long(S)$
31: **if** $L > 0$ **then**
32: $S \leftarrow ordenar(S)$
33: $l \leftarrow 0$
34: **while** $l < L$ **do**
35: $s \leftarrow S(l)$
36: $\varepsilon \leftarrow \varepsilon(s)$
37: $\psi \leftarrow \psi(s, \varepsilon)$
38: $\eta \leftarrow \eta(s, \psi, \varepsilon)$
39: $r \leftarrow l + 1$
40: **while** $\big((r < L) \wedge (\varepsilon = \varepsilon(S(r))) \big)$ **do**
41: $s' \leftarrow S(r)$
42: $\psi' \leftarrow \psi(s', \varepsilon)$
43: $\eta' \leftarrow \eta(s', \psi')$
44: **if** $domina(s, s', \varepsilon, \psi, \psi', \eta, \eta')$ **then**
45: $S \leftarrow S - S(r)$
46: $L \leftarrow L - 1$
47: **else if** $domina(s', s, \varepsilon, \psi', \psi, \eta', \eta)$ **then**
48: $S \leftarrow S - S(l)$
49: $L \leftarrow L - 1$
50: $l \leftarrow l - 1$
51: **break**
52: **else**
53: $r \leftarrow r + 1$

```
54:             end if
55:           end while
56:           l ← l + 1
57:         end while
58:     else
59:         break
60:     end if
61:     k ← k + 1
62: end while
63: if long(S) = 0 then
64:     solOptima ← solParcial
65: else
66:     solOptima ← S[0]
67: end if
68: valorOptimo ← tiempoFinal(solOptima)
    return valorOptimo
```

Antes de poder explicar el algoritmo, haremos una breve reseña de lo que hace cada función dentro de este. Nos ahorraremos el pseudocódigo de esta parte ya que creemos que el cómputo de cada función no tiene ninguna dificultad.

1. $\varepsilon(s)$: Toma una secuencia s y calcula el conjunto $\varepsilon(q(s))$.
2. $\psi(s,\varepsilon)$: Calcula para cada operación o en el conjunto ε su tiempo de inicio $\psi(s,o)$.
3. $\eta(s,\psi,\varepsilon)$: Toma una secuencia s, un conjunto de operaciones ε y sus tiempos de inicio ψ y se computa el conjunto de operaciones $\eta(s)$ que expanden a s ordenadamente.
4. $\widehat{\eta}(s,\psi,\eta)$: Toma una secuencia s, el conjunto de operaciones η y sus tiempos de inicio ψ, y computa el conjunto $\widehat{\eta}(s)$.
5. $expandir(s,\psi,\widehat{\eta})$: Toma una secuencia s, el conjuntos de operaciones que no generan retrasos $\widehat{\eta}$ y sus tiempos iniciales ψ, para generar todas las posibles secuencias $s+o$ con $o \in \widehat{\eta(s)}$.
6. $actualizarCota(s,\varepsilon,\psi,\eta)$: Toma una secuencia s, un conjunto de operaciones ε que pueden ser programadas a tiempos iniciales ψ y el conjunto de operaciones ordenadas η y computa la cota inferior $CI(s)$.

7. *ordenar*(S): Toma una lista de secuencias S y la ordena, dejando todas las secuencias con las mismas operaciones programadas juntas. Este proceso se realiza mediante la comparación de los conjuntos $\lambda(s)$ para cada $s \in S$.

8. *domina*$(s, s', \varepsilon, \psi, \psi', \eta, \eta')$: Toma dos secuencias s y s' con el mismo conjuntos de operaciones a programar ε, sus respectivos tiempos iniciales ψ y ψ' y sus conjuntos de operaciones ordenadas η y η' respectivamente, y devuelve $TRUE$ si y solo si $s \preceq s'$.

Explicaremos el algoritmo brevemente, dividiendo la explicación en los bloques significativos en donde se ejecutan los pasos básicos.

- Líneas 1 a 2: Se define la lista inicial con la secuencia nula como su único elemento.

- Lineas 3 a 4: Se procede a correr una heurística para calcular soluciones factibles y mejorar la cota superior obtenida. Dejaremos por ahora de lado la metodología de estas heurísticas, ya que se explicarán en detalle en el próximo capítulo.

- Líneas 5 a 62: Se fija el valor 0 para k. Luego se itera sobre el número de etapas del problema, hasta llegar a obtener secuencias completas o salir del bucle porque la lista es vacía. Siempre que la lista resulta ser vacía, entonces la solución óptima será la solución parcial almacenada en la variable $solParcial$, y se procederá al final del algoritmo.

- Lineas 7 a 16: Este bloque de líneas se corresponde con la fase de expansión de la etapa k-ésima, dando lugar a una nueva etapa. Para cada $s \in S$, primero se procede a calcular $\widehat{\eta(s)}$, lo que se hace en la línea 13. Luego, se generan las secuencias *hijas* de s, las cuales se apendizan al final de la lista. Finalmente, se descarta s.

- Líneas 17 a 60: En este bloque de líneas se ejecuta la fase de eliminación del algoritmo. En primer lugar, se eliminan las secuencias por cotas inferiores, para finalmente proceder a la eliminación por dominancia.

- Líneas 19 a 29: Eliminación por cotas: para cada secuencia $s \in S$, se calcula y actualiza la cota inferior, lo que se hace en la línea 24. Luego, si la nueva cota resulta ser mayor o igual que la cota superior CS, se procede a descartar la secuencia.

- Línea 32: En caso de resultar la lista diferente de vacía, se ordena la lista de forma que las secuencias que tengan las mismas operaciones programadas queden juntas. Esto se hace mediante la comparación de los conjuntos $\lambda(q(s))$ para cada secuencia s en la lista S.

- Líneas 40 a 55: Eliminación por dominancia. Fijada s, se itera sobre las secuencias s' que tengan las mismas operaciones programadas. Si $s \preceq s'$, entonces se descarta s'. Si, por el contrario $s' \preceq s$, se elimina s, dando lugar a la evaluación de la próxima secuencia.

- Lineas 63 a 67: Una vez terminado el ciclo, se procede a verificar si la lista es vacío. Si la lista no es vacío, entonces la única secuencia en la lista es la secuencia óptima, por lo que $solOptima \leftarrow S[0]$. En caso de que la lista sea vacía, entonces la secuencia óptima será la solución parcial. Luego, $solOptima \leftarrow solParcial$

- Linea 68: Se retorna el valor de la secuencia óptima para el algoritmo.

5.2. Complejidad del algoritmo

En esta subsección estimaremos la complejidad algorítmica en términos de las dimensiones de la instancia. Recalcamos que esta estimación se hará de manera intuitiva e informal.

5.2.1. Supuestos previos

Nuestro algoritmo almacena, para cada secuencia parcial, más información de la necesaria: hemos guardado la secuencia de operaciones que llamaremos $secuenciaOperaciones(s)$, así también como la secuencia de tiempos en que finaliza cada operación, que denominaremos $secuenciaTiemposFinales(s)$. También, hemos guardado el conjunto correspondiente a $\lambda(q(s))$, al que nos referiremos como $\lambda(s)$. Por último, para cada secuencia s guardamos la cota inferior para cada máquina, a las que llamaremos $cotasInf(s)$. Todos estos conjuntos son guardados como arreglos, por lo que supondremos que acceder a cualquier valor en ellos tiene complejidad 1. De esta forma, al contar con esta información adicional algunas funciones se calcularán más rápidamente.

Observemos que, con los datos iniciales y la $secuenciaOperaciones(s)$ podríamos calcular las demás variables mencionadas en el párrafo anterior. Luego, la información minimal requerida será la correspondiente a la variable $secuenciaOperaciones(s)$. No obstante, hemos estimado que la cantidad en que aumenta el uso de la memoria para cada secuencia parcial esta acotada superiormente por 3 veces la memoria utilizada para guardar

secuenciasOperaciones(s). Dado que este factor no es muy grande, no consideramos que este aumento impacte directamente en la capacidad del algoritmo para resolver alguna instancia, ya que, como veremos a continuación, la complejidad resulta exponencial (esto significa que el crecimiento en cantidad de secuencias parciales excede por mucho a cualquier función lineal, por lo que el impacto computacional del aumento del espacio utilizado será despreciable).

5.2.2. Parámetros y variables

Se hace necesario establecer una referencia acerca de cada variable y parámetro utilizados en el cálculo de la complejidad. A continuación haremos una breve descripción de cómo se almacenarán los datos.

Parámetros:

- n representará el número de jobs.
- m el número de máquinas.
- *ListaOperaciones* será un arreglo de nm coordenadas, en donde la coordenada k guardará los datos de la operación o_k.

Variables:

- $\lambda(s), \varepsilon(s), \psi(s,\varepsilon), \eta(s,\psi,\varepsilon), \widehat{\eta}(s,\psi,\eta)$ serán arreglos de tamaño n, donde la coordenada j se referirá a la operación correspondiente al job j. Además, en todos ellos el valor 0 se corresponde con la ausencia lógica de un valor. Por ejemplo, si $\lambda(s)[j] = 0$, entonces significa que la primera operación correspondiente al job j, o_j, no se ha programado todavía.
- $secuenciasOperaciones(s), secuenciasTiemposFinales(s)$ serán arreglos de tamaño variable $d(s)$. Así, $secuenciasTiemposFinales(s)[k]$ será el tiempo de finalización de la operación programada almacenada en $secuenciasOperaciones(s)[k]$.
- $cotasInf(s)$ será un arreglo de m posiciones, que guardará en la coordenada i la cota inferior $CI(s,i)$ para la máquina i-ésima.

Observemos que una vez obtenidos cualquiera de los arreglos, el acceso a la información almacenada en ellos, constituye una operación elemental. Por ende, mediante diferentes combinaciones de acceso entre el parámetro *ListaOperaciones*, *secuenciasOperaciones*(s) y $secuenciasTiemposFinales(s)$, podremos obtener con complejidad $O(1)$:

- $C_{\text{máx}}(s)$
- La máquina ó el job de una operación en $secuenciasOperaciones(s)$. Llamaremos m^* a la máquina correspondiente a la última operación de s.

5.2.3. Análisis de complejidad

Para empezar estimaremos la complejidad de las funciones $\varepsilon(s)$, $\psi(s,\varepsilon)$, $\eta(s,\psi,\varepsilon)$, las cuales se utilizarán recurrente en las demás funciones.

1. $\varepsilon(s)$: Calcularemos en base a $\lambda(s)$ cada operación en $\varepsilon(s)$, siguiendo la siguiente regla:

   a) Si $\lambda(s)[j] + n < nm - n \Rightarrow \varepsilon(s)[j] = \lambda(s)[j] + n$.

   b) En caso contrario, $\varepsilon(s)[j] = 0$.

   Este proceso se puede realizar en n iteraciones más. Luego, la complejidad final de computar $\varepsilon(s)$ es $O(n)$.

2. $\psi(s,\varepsilon)$: Para cada operación $\varepsilon[j]$, buscaremos la última operación en $secuenciaOperaciones(s)$ que utilice la misma máquina i ó tenga el mismo job j. Para esto, iteraremos en el arreglo $secuenciaOperaciones(s)$ desde el final hasta el principio, y detendremos el proceso de búsqueda apenas encontremos una operación que cumpla el criterio anterior. Dentro de este escenario, el peor caso resulta ser que no se hayan programado operaciones del mismo job ó máquina. En este caso, recorreremos la totalidad de $secuenciaOperaciones(s)$ y definiremos $\psi(s,\varepsilon) = 0$. Suponiendo que la secuencia s pertenezca a la etapa k (i.e, $d(s) = k$), se deberá iterar k veces. No obstante, observemos que en la práctica es muy díficil que suceda este caso para secuencias con varias operaciones.

   Si la operación encontrada ocupa la l-ésima posición, el tiempo inicial será $secuenciasTiemposFinales(s)[l]$.

   Como ejecutaremos el mismo proceso para cada operación, la complejidad será $O(kn)$, donde $d(s) = k$.

3. $\eta(s,\psi,\varepsilon)$: Para cada operación de $\varepsilon[j]$ seguiremos el siguiente procedimiento:

   a) Si $\varepsilon[j] = 0 \Rightarrow \eta(s,\psi,\varepsilon)[j] = 0$

   b) Si $\varepsilon[j] \neq 0$:

i) Si $\psi[j] + p(\varepsilon[j]) < C_{\text{máx}}(s) \vee \big(\psi[j] + p(\varepsilon[j]) = C_{\text{máx}}(s) \wedge m^* > m(\varepsilon[j])\big) \Rightarrow \eta(s, \psi, \varepsilon)[j] = 0$.

ii) En caso contrario, $\eta(s, \psi, \varepsilon)[j] = \varepsilon[j]$.

La complejidad será $O(n)$, dado que todas las comparaciones, sumas y acceso a la información tienen complejidad $O(1)$.

4. $\widehat{\eta}(s, \psi, \eta)$: Para cada operación $o \in \eta$, debemos ver si o esta en $\widehat{\eta(s)}$. Para esto, verificaremos que se cumplan las condiciones de la Definición 4.25 para cada una de las operaciones $o' \in \eta, o' \neq o$.

   Teniendo en cuenta que ya fueron computados los tiempos iniciales de todas las operaciones en ε, y que además saber la máquina de ejecución de las operaciones es $O(1)$, no es díficil ver que estas condiciones pueden ser validadas en $O(1)$ para cada operación $o' \neq o$.

   De lo anterior se desprende que la complejidad de $\widehat{\eta}(s, \psi, \eta)$ es $O(n^2)$, en el caso que η tenga n operaciones.

Seguiremos, ahora, por estimar la complejidad de los procesos antes descritos.

1. $heuristica(S)$: Esta función se ejecuta por única vez, sobre la primera etapa de operaciones. No entraremos en detalles, pero como su nombre lo indica, su complejidad resultará polinomial, y no influirá en el resto del análisis.

2. $expandir(s, \psi, \widehat{\eta})$: Para expandir una secuencia, se genera una nueva secuencia agregando una operación. Se copia la secuencia vieja, por lo que se debe iterar sobre cada arreglo de la secuencia s, siendo en general los más largos $secuenciasOperaciones(s)$ y $secuenciasTiemposFinales(s)$. Además, se debe agregar la nueva operación.

   En el peor de los casos, cuando $\varepsilon(q(s)) = \widehat{\eta(s)}$, se tendrán que generar n copias, pues todas las operaciones expanden s ordenadamente y no generan retrasos. Por lo tanto, la complejidad de la función es $O((k+1)n)$ ó, equivalentemente, $O(kn)$.

3. $ordenar(S)$: Esta función ordena la lista de secuencias parciales, utilizando los conjuntos $\lambda(s)$ para cada secuencia. En principio, para poder comparar dos secuencias s y s' se deberán comparar $\lambda(s)$ con $\lambda(s')$. Esto se puede hacer en $O(n)$.

   A continuación, habrá que ordenar la lista. Hemos utilizado la función *stable sort* que viene implementada en la librería standard de

C++. Para esta función, la complejidad en el peor de los casos es $O(L(k)log_2(L(k))^2)$, donde $L(k)$ es la longitud de la etapa k. Por lo tanto, la complejidad final de este proceso se puede estimar como $O(nL(k)log_2(L(k))^2)$. Próximamente, acotaremos esta longitud en base a los parámetros de entrada.

4. $actualizarCota(s, \varepsilon, \psi, \eta)$: No explicaremos con mayor detalle cómo computa esta función las cotas para cada máquina i, pues resultaría tedioso y confuso. No obstante, observemos que dada la secuencia parcial s y las variables ε,ψ y η , para computar tanto $CA(s, i), TR(s, i)$ y $CO(s, i)$ necesitaremos realizar al menos una iteración sobre todas las operaciones no programadas en s, para luego poder calcular los mínimos y máximos de cada job para la máquina i. En el peor de los casos, esto tendrá complejidad $O(nm)$, siendo nm el total de las operaciones.

5. $domina(s, s', \varepsilon, \psi, \psi', \eta, \eta')$: Esta función compara las operaciones en ε según los criterios de la definición 4.18. Básicamente:

   a) Si $\varepsilon[j] \neq 0$, se computan $\xi(s, \varepsilon[j])$ y $\xi(s', \varepsilon[j])$, utilizando los arreglos ψ, ψ', η y η'.

   b) Se proceden a hacer las comparaciones entre $\xi(s, \varepsilon[j])$ y $\xi(s', \varepsilon[j])$ para definir si $s \preceq s'$.

   De lo anterior no es díficil ver que la complejidad de esta función es $O(n)$.

Una vez hallada la complejidad de estas funciones, ya estamos en condiciones de estimar la complejidad del algoritmo exacto. Para ello, supondremos que, en el peor de los casos, ninguna secuencia es eliminada mediante cualquiera de los criterios.

- Líneas 5 a 62: El número de etapas total será nm.

- Lineas 7 a 16: Para cada secuencia s de la etapa k-ésima se computarán sus posibles expansiones ordenadas. Para ello se ejecutarán $\varepsilon(s)$, $\psi(s, \varepsilon)$, $\eta(s, \psi, \varepsilon)$, $\widehat{\eta}(s, \psi, \eta)$ y $expandir(s, \psi, \widehat{\eta})$. De todas éstas, la complejidad máxima es $O(kn)$, y corresponde a $expandir(s, \psi, \widehat{\eta})$.

  Luego, la complejidad de este bloque será $O(L(k)kn)$.

- Líneas 19 a 29: Se calculan para cada secuencia de la etapa $\varepsilon(s)$, $\psi(s, \varepsilon)$, $\eta(s, \psi, \varepsilon)$ y $actualizarCota(s, \varepsilon, \psi, \eta)$. Como de todas estas, la que tiene mayor complejidad es $actualizarCota(s, \varepsilon, \psi, \eta)$, este bloque de líneas tendrá una complejidad final a $O(L(k)nm)$, donde la $L(k)$ es la cantidad de secuencias parciales en la etapa k.

- Línea 32: Se ejecuta la función *ordenar* sobre la lista de secuencias parciales de la etapa S. Como ya hemos mencionado, esta función tiene complejidad $O(nL(k)log_2(L(k))^2)$.

- Líneas 40 a 55: Fijada s, en este bloque de líneas tendremos que iterar para cada secuencia en el conjunto $\mathcal{Z}(Q)$, para cada Q subconjunto factible. Dentro de cada iteración la función más costosa es $\psi(s, \varepsilon)$ en la gran mayoría de los casos (cuando $k > n$), cuya complejidad es $O(k)$.

  Para acotar de forma sencilla esta parte, supondremos que para cada secuencia s iteraremos sobre la totalidad de la etapa. De esta forma, la complejidad final de este bloque será equivalente a $O(kL(k)^2)$.

  Cabe destacar que esta cota es muy imprecisa, pero no cambiará el resultado del análisis final de complejidad.

Una vez estimadas la complejidad de cada bloque, como suponemos que ninguna secuencia será descartada, $L(k)$ se mantendrá constante cada bloque descrito anteriormente. Luego, estimaremos la complejidad de la etapa k-ésima como la máxima de las complejidades correspondientes de cada bloque.

Es claro que las dos complejidades más grandes corresponden a ordenar la lista y eliminar por dominancia, las cuales son $O(nL(k)log_2(L(k))^2)$ y $O(kL(k)^2)$ respectivamente. Entre estas dos, si consideramos que $k > n$ salvo en $\frac{1}{m}$ parte del problema, y que $log_2(L(k))^2 < L(k)$ para $k > 15$, podemos tomar a $O(kL(k)^2)$ como la máxima de las complejidades de los bloques.

Por lo tanto, la complejidad del problema será calculada como

$$\sum_{k=1}^{nm} kL(k)^2$$

Observemos que de la última expresión, el término que realmente hace que el problema sea intratable es $L(k)$. Si bien esta observación resulta bastante obvia, notemos que si acotamos $L(k)$ por una cantidad fija α, tendremos que la complejidad del problema será

$$\sum_{k=1}^{nm} k\alpha^2 = \alpha^2\left(\frac{(nm+1)nm}{2}\right)$$

lo que resulta polinomial. Esta observación será primordial para justificar nuestro algoritmo heurístico, en el siguiente capítulo.

Finalmente, resta obtener una cota para $L(k)$ en términos de los datos iniciales del problema. Para ello, si fijamos la etapa k, la cantidad de subconjuntos factibles está definida por la cantidad de diferentes vectores λ tales que $\sum_{i=1}^{n} \lambda[i] = k$, donde $0 \leq \lambda[i] \leq m$. Este número esta acotado superiormente por la cantidad de particiones de un número natural k, con la

condición de que cada sumando puede ser nulo. Luego, mediante argumentos combinatorios, esta cantidad resulta ser

$$\binom{n+k-1}{n-1}$$

A su vez, en [6], los autores demuestran que $|\overset{(\preceq)}{\mathcal{X}}(Q)| \leq \frac{(p_{\text{máx}})^n}{\sqrt{n}}$, donde $p_{\text{máx}})$ es el tiempo máximo de ejecución. Como hemos visto que $\widetilde{\mathcal{Z}(Q)} \subseteq \overset{(\preceq)}{\mathcal{X}}(Q)$, entonces dicha cota también valdrá para $\widetilde{\mathcal{Z}(Q)}$.

Por lo tanto, para la etapa $k+1$, generaremos, a lo sumo, una cantidad de secuencias igual a

$$n\frac{(p_{\text{máx}})^n}{\sqrt{n}} = \sqrt{n}(p_{\text{máx}})^n$$

Así, $L(k)$ estará acotada por

$$\binom{n+k-2}{n-1}\sqrt{n}(p_{\text{máx}})^n$$

Si consideramos que el algoritmo empieza con la secuencia nula en la etapa 0, tendremos que la complejidad es

$$\sum_{k=1}^{nm} kL(k)^2 \leq \sum_{k=1}^{nm} k\binom{n+k-2}{n-1}^2 \sqrt{n}^2\left((p_{\text{máx}})^n\right)^2 =$$

$$= n(p_{\text{máx}})^{2n}\sum_{k=1}^{nm} k\binom{n+k-2}{n-1}^2 = n(p_{\text{máx}})^{2n}C(n,m)$$

con $C(n,m) = \sum_{k=1}^{nm} k\binom{n+k-2}{n-1}^2$

Como conclusión, queda claro que la complejidad es exponencial, ya que el término $C(n,m)$ lo es.

6. Heurística para el JSSP

6.1. Introducción

El método implementado en el capítulo anterior, si bien es mucho más eficiente que fuerza bruta, en el fondo sigue siendo un algoritmo exponencial. Ejemplos relativamente pequeños pueden ser resueltos, aunque como mostramos en los resultados expuestos en 7.1, los tiempos de ejecución y el consumo de memoria crecen demasiado rápido con el tamaño del problema, haciendo que instancias de 10×10 (10 jobs y 10 máquinas) resulten virtualmente intratables.

Este hecho induce a ensayar métodos heurísticos que permitan obtener soluciones buenas en un tiempo aceptable. En este capítulo presentamos una variante heurística que desarrollamos a partir de modificaciones en el algoritmo de PD expuesto anteriormente.

Una posible solución al crecimiento exponencial de la memoria utilizada consiste en mantener controlada la cantidad de secuencias parciales por etapa. Con esto queremos decir que dicha cantidad de secuencias se encuentre acotada de alguna forma. En este trabajo, se plantean dos maneras de concretar lo dicho:

1. Manteniendo controlada la cantidad de operaciones posibles con que se pueden expandir ordenadamente las secuencias. Esto acotaría el grado de crecimiento entre etapas.

2. Manteniendo acotada la cantidad de secuencias parciales en cada etapa.

Ambos métodos implican que, al generarse cada etapa, el conjunto de secuencias parciales en ella se encuentre sesgado. En el primer caso se sesga la expansión, mientras que en el segundo caso se eliminan secuencias de la misma etapa. Lamentablemente, a diferencia de los métodos de eliminación expuestos hasta aquí, esta elección no esta justificada por un criterio que asegure no descartar potenciales soluciones óptimas. Por lo tanto, la aplicación de estas técnicas de control tiene como efecto la posible eliminación de caminos óptimos, transformando el algoritmo exacto en un algoritmo heurístico.

En los próximos apartados explicaremos muy básicamente de qué forma podemos aplicar ambas técnicas al JSSP. Finalmente, propondremos una implementación heurística basada en estos criterios y mostraremos los resultados obtenidos experimentalmente sobre las instancias ya mencionadas.

6.2. Reglas de prioridad sobre operaciones

Debido a la dificultad para hallar una solución exacta, los primeros intentos sobre el JSSP fueron lo que se conocen como heurísticas *míopes*.

De forma muy general, dada una solución parcial y las posibles elecciones que se han de tomar sobre esta, una heurística míope decide cuál de ellas seguir en base a la información obtenida de la solución parcial. Las demás posibilidades son descartadas.

Es por esto que estas heurísticas tienen la ventaja de que resultan muy rápidas y casi no consumen recursos. No obstante, en la mayor parte de los casos, las soluciones así obtenidas suelen estar alejadas del óptimo. Por lo tanto, este tipo de heurística se utiliza para obtener soluciones factibles rápidamente, que luego podrán usarse en algoritmos más complejos como soluciones iniciales.

Volviendo al JSSP, el abanico de decisiones a tomar esta constítuido de las posibles operaciones próximas a programarse. Una heurística míope sobre este escenario será, pues, algún criterio de elección que nos permita establecer una jerarquía entre las operaciones utilizadas para la expansión. Así, la secuencia parcial será expandida con la mejor/res operación/nes.

En la literatura estas heurísticas son conocidas como *heurísticas basadas en reglas de prioridad*. Pueden ser resumidas mediante la resolución de dos pasos básicos, que explicaremos informalmente a continuación:

1. Definido un schedule parcial ψ de un subconjunto $Q \subset O$, se listan todas las operaciones $o \in \varepsilon(Q)$ con que se pueda extender ψ a un schedule ψ' factible y activo.

2. Del anterior conjunto de operaciones, se elige una operación o según una regla de prioridad establecida sobre los parámetros de ψ. Por regla de prioridad entendemos una función $r(\psi, o) \in \mathbb{R}$ tal que $\forall o_1, o_2 \in \varepsilon(Q)$: $r(\psi, o_1) \neq r(\psi, o_2) \Leftrightarrow o_1 \neq o_2$. Luego, se genera ψ' schedule parcial con la operación elegida.

El primer paso asegura que la expansión genere un schedule factible y activo, ya que se irá construyendo iterativamente sobre schedules parciales que cumplan dicha condición.

En el segundo paso se decide la elección de las operaciones según reglas de prioridad. Estas reglas son, en definitiva, las que diferencian una heurística de otra. Es por esto que cada heurísticas suelen recibir el nombre de la regla de prioridad que le corresponde.

A continuación, mencionaremos las reglas más conocidas y mayormente usadas:

1. SPT (shortest processing time): seleccionar una operación con el tiempo de procesamiento más corto.

2. LPT (longest processing time): seleccionar una operación con el tiempo de procesamiento más largo.

3. MWR (most work remaining): seleccionar una operación para el job con el mayor tiempo de procesamiento pendiente.

4. LWR (least work remaining): seleccionar una operación para el job con el menor tiempo de procesamiento pendiente.

5. MOR (most operations remaining): seleccionar una operación para el job con el mayor número de operaciones pendientes.

6. LOR (least operations remaining): seleccionar una operación para el job con el menor número de operaciones pendientes.

7. RND (random): seleccionar una operación en forma aleatoria.

Es importante destacar que las heurísticas basadas sobre reglas de prioridad deben no ser ambiguas respecto a la elección de las operaciones, de forma de determinar unívocamente cúal operación tiene prioridad sobre las otras en cualquier situación. Para esto, suelen componerse reglas o criterios que determinen un orden total sobre $\varepsilon(Q)$.

Además, es deseable que estas reglas tengan un sentido lógico, es decir, que establezcan la prioridad de las operaciones teniendo en cuenta que el objetivo final es la minimización del tiempo de finalización.

6.3. Beam Search

El *Beam Search* es una técnica heurística que actua sobre el árbol de búsqueda. Por lo tanto, es aplicable a algoritmos como de PD ó *Branch and Bound*, pues éstos generan dicho árbol. En el caso de complementarse con PD se lo conoce también *Programación Dinámica Acotadada*.

La idea básica del Beam Search es *podar* nodos de una misma etapa del árbol de búsqueda, según algún criterio no exacto. Esta característica tiene como ventaja la eliminación de un gran número de soluciones parciales, y como desventaja la pérdida de la optimilidad del algoritmo, lo cual le otorga su cáracter de heurística.

Para desarrollar el método se deben fijar ciertos parámetros y tomar una serie de decisiones básicas. Haremos una breve descripción del marco general.

Denotaremos al árbol de búsqueda como T. A su vez, denotaremos como T_k a la etapa o nivel k-ésima de dicho árbol, es decir, $T_k = \{t \in T : d(t) = k\}$, con $d(t)$ la distancia desde la solución parcial t al nodo inicial. Además, denotaremos como $N(T, k)$ al número de nodos de la etapa T_k.

En general, es deseable que el tamaño del árbol no crezca demasiado, por lo que hay que tener controlada la cantidad de nodos por etapa. Para esto, se fijará un parámetro que denotaremos como $\alpha(k)$, el cual representará el límite máximo de nodos permitidos para la etapa k. De esta manera, ejecutaremos la poda si $N(T, k) > \alpha(k)$.

En segundo lugar, los nodos serán seleccionados según un criterio a definir. Dicho criterio esta representado mediante una función $\gamma(t)$ que otorga un valor a cada nodo t. Esta función valuará las propiedades de las soluciones parciales, estableciendo una jerarquía que irá desde la mejor hasta la peor.

Ya establecido el marco teórico del método, el desafío es obtener una buena composición de los parámetros para hacer la heurística lo más rápida y efectiva posible.

El parámetro $\alpha(k)$ limitará la máxima cantidad de nodos para cada etapa. Así, al aumentar el $\alpha(k)$, más *ancho* será el árbol de búsqueda, lo que implica un mayor tiempo de ejecución. No obstante, es probable que nuestra precisión también se incremente, ya que descartaremos menos nodos en cada etapa. Por el contrario, si $\alpha(k)$ es muy pequeño, las podas serán mucho más frecuentes, lo que hará que el método sea más rápido pero menos preciso.

Al ejecutarse la poda, los nodos serán evaluados mediante la función $\gamma(t)$. Hay que tener en cuenta que cuánto más compleja resulte la evaluación (más parámetros involucre), en general más costosa será. Si la evaluación es demasiado costosa, necesitaremos que la cantidad de nodos a evaluar sea pequeña, por lo que $\alpha(k)$ debe ser pequeño. En general, para poder llegar obtener un método equilibrado, $\alpha(k)$ y la complejidad de $\gamma(t)$ deben estar relacionadas inversamente.

6.4. Implementación heurística para el JSSP

En esta subsección describiremos nuestra implementación heurística para el JSSP. Comenzaremos con una descripción general del método, para luego exponer el algoritmo y los resultados obtenidos.

6.4.1. Método general

Básicamente, la heurística no es más que una implementación particular del beam search sobre el algoritmo exacto. Además, se ha definido una

regla de prioridad para acotar la cantidad de operaciones utilizadas en la expansión.

En nuestra implementación, $\alpha(k)$ no dependerá del número de etapa, por lo que será una constante que llamaremos α. De esta forma, el árbol de búsqueda estará acotado en cada etapa por el mismo número de nodos.

Para $\gamma(s)$ se han probado gran variedad de opciones que utilizan la cota inferior $CI(s)$. Los mejores resultados fueron obtenidos mediante la siguiente función:

$$\gamma(s) = \frac{\sum_{i=1}^{m} CI(s,i)}{m}$$

En este caso, se toma el promedio de las cotas inferiores por máquina para establecer la jerarquía de las secuencias.

Por último, para mantener acotado el crecimiento del árbol para cada instancia, se ha creado una regla de prioridad, que hemos llamado π. Dicha regla ordena las operaciones en $\widehat{\eta(s)}$ según el orden de criterios expuesto a continuación.

Definiremos para cada operación $o \in \widehat{\eta(s)}$:

$$tiempoRestante(s,o) = \psi(s,o) + \sum_{o' \in (\mathcal{O} \setminus q(s)) : j(o') = j(o)} p(o')$$

Entonces, sean $o^1, o^2 \in \widehat{\eta(s)}$. Luego, $o^1 <_{\pi} o^2$ si se cumple alguna de las siguientes condiciones:

1. $\psi(s,o^1) + p(o^1) < \psi(s,o^2) + p(o^2)$

2. $\psi(s,o^1) + p(o^1) = \psi(s,o^2) + p(o^2)$ y $m(o^1) < m(o^2)$

3. $\psi(s,o^1) + p(o^1) = \psi(s,o^2) + p(o^2)$, $m(o^1) = m(o^2)$ y además $tiempoRestante(o^1) < tiempoRestante(o^2)$

4. $\psi(s,o^1) + p(o^1) = \psi(s,o^2) + p(o^2)$, $m(o^1) = m(o^2)$, $tiempoRestante(o^1) = tiempoRestante(o^2)$ y además $indice(o^1) < indice(o^2)$

El parámetro β será la cantidad máxima de operaciones de $\widehat{\eta(s)}$, ordenadas según π, que utilizaremos para expandir s. Así, el crecimiento del árbol de búsqueda estará acotado para la etapa k-ésima por $\min\{N(T,k-1),\alpha\}\beta$.

6.4.2. Construcción del algoritmo

Debido a que nuestra implementación del método heurístico resulta casi idéntica a la del algoritmo JSSP, sólo expondremos los nuevos bloques

Algorithm JSSP-Beam Search

```
1: ···
13:     η̂ ← η̂(s, ψ, η)
14:     ordenarΠ(ψ, η̂)
15:     S ← S + expandir(s, ψ, η̂, β)
16:     S ← S − S(0)
17: ···
58:             ordenar(S, γ)
59:             S ← S[0 : α]
60:             L ← α
61: ···
70: if long(S) = 0 then
71:     solOptima ← solParcial
72: else
73:     solOptima ← S[0]
74: end if
75: valorOptimo ← tiempoFinal(solOptima)
        return valorOptimo
```

de código. Utilizaremos "$\cdots$"para denotar que las líneas de código son las mismas que en el algoritmo exacto.

Haremos una breve descripción de los bloques nuevos con respecto al algoritmo exacto:

- Líneas 14 a 15: La primera línea de estas dos ordena las operaciones en $\widehat{\eta}$ mediante la regla de prioridad π. La segunda, expande desde la primera operación hasta la operación β de las operaciones en $\widehat{\eta}$.

- Líneas 58 a 60: Se realiza la poda del *beam search*. La línea 58 ordena la lista utilizando la función γ. Luego, las intrucciones $S \leftarrow S[0 : \alpha]$ y $L \leftarrow \alpha$ descartan las secuencias sustituyendo la lista de secuencias adecuadamente.

Como se ve en el anterior algoritmo, la poda de nodos por beam search se realiza lógicamente después de haber eliminado las secuencias parciales por los demás criterios expuestos.

No haremos ninguna cuenta adicional para demostrar que la heurística tiene complejidad polinomial. Sólo observaremos que la complejidad de $ordenar(S, \gamma)$ y $ordenar\Pi(\psi, \widehat{\eta})$ es $O(mL(k)log(L(k))^2)$ y $O(n^2)$ respectivamente. A su vez, como hemos observado puntualmente en el cálculo de la complejidad (Ver subsección 5.2.3), si acotamos la cantidad de secuencias

parciales para cada etapa $L(k)$ podremos obtener un algoritmo polinomial. Dado que para el beam search vale que $\forall k : L(k) \leq \alpha$, entonces queda probado que el método heurístico tiene una complejidad polinómica.

Para finalizar, explicaremos en qué consiste la función $heuristica(S)$, que se ejecuta en la línea 3 tanto del algoritmo JSSP como del JSPP-Beam Search.

Resumidamente, se ejecutará el algoritmo JSSP-Beam Search consecutivamente dos veces, pero sin ejecutarse la función $heuristica(S)$ en ambos casos. Para ambas ejecuciones se ha tomado como función de valuación a γ.

La primera vez se define $\alpha = 500$, y CS será la suma de todos los jobs, que resulta claramente una cota superior del problema. Luego, con la solución factible obtenida se actualizará el valor de la cota superior CS.

La segunda ejecución será similar a la primera, pero con $\alpha = 5000$. Notemos que, para la segunda vez, ya tendremos una cota superior más ajustada y un incremento en el límite de secuencias, por lo que se presupone que se obtendrá un mejor resultado.

Ambas ejecuciones son muy rápidas debido a que los valores de α son bajos en ambos casos.

Una vez descrito el método heurístico, se han realizado una serie de experimentos correspondientes a estas variantes, cuyos resultados pueden verse en la subsección 7.2. En cada experimento, dada una instancia particular del JSSP, se han variado los valores de α y β, a fin de poder comparar qué configuración de los parámetros resulta más efectiva en función de los datos de entrada.

7. Resultados y análisis

7.1. Algoritmo Exacto

Las instancias utilizadas para testear el algoritmo han sido las primeras 20 creadas y utilizadas por Lawrence en su trabajo [9] LA01-LA20, así como también las tres instancias correspondientes a Fisher y Thompson[2]: FT06, FT10 y FT20. Dichas instancias son conocidas en la literatura y han sido utilizadas en diversos trabajos sobre el JSSP para analizar la aplicación de los métodos. Por ser de tamaño pequeño a mediano (no superan las 100 operaciones), constituyen la primera prueba de cualquier algoritmo ó heurística, antes de pasar a instancias mucho más grandes.

Además, se crearon dos instancias nuevas: $N1 = LA11 + LA12$ y $N2 = LA11 + LA12 + FT20$. Ambas instancias son el resultado de añadir los jobs de las instancias correspondientes. Así, estas dos instancias tienen 40 y 60 jobs respectivamente, y 5 máquinas de procesamiento.

El algoritmo fue implementado en C++, y los experimentos computacionales fueron realizados con un procesador AMD Sempron(tm) 145, bajo el sistema operativo Linux Mint 13 y con 3097 MB de memoria RAM.

Los Cuadros 3 y 4 resumen la información de los resultados obtenidos. En el Cuadro 3 se ven: la instancia, su tamaño y el valor óptimo. Los valores indicados con un $*$ corresponden a instancias para las cuales el algoritmo no terminó, por haber consumido completamente los recursos del sistema. Las instancias $N1$ y $N2$ son marcadas con †, dado que para ellas no conocíamos previamente el valor óptimo. En todos los otros casos, el valor obtenido por el algoritmo coincide con el óptimo conocido. Las siguientes columnas corresponden al número de etapas efectivamente generadas por el algoritmo (E) y la cota superior (CS) utilizada para la poda. En la mayor parte de los casos la cota superior coincide con el óptimo, por lo que el algoritmo se detiene rápidamente, mucho antes de construir todas las etapas teóricamente necesarias (es decir: $E < n \times m$). Finalmente, en las últimas columnas se muestran: la cantidad máxima de secuencias efectivamente generadas, el porcentaje que éstas representan respecto del total, y el porcentaje que representan las secuencias eliminadas, separadas de acuerdo al criterio de eliminación: por órdenes con retrasos (R), por cota (C) o por dominancia (D).

El Cuadro 4 está dedicado a los tiempos de ejecución. Allí se muestran los tiempos de ejecución declarados en [6], el tiempo de ejecución total de nuestro algoritmo (Tot.) y el desglose de este tiempo entre el cálculo de la cota superior (CS) y el ciclo principal (CP). Además, se detalla los porcentajes de tiempo dedicados a los ciclos de expansión (Exp.), al cálculo de cotas (Cot.), al ordenamiento (Ord.) y a la eliminación de secuencias dominadas (Dom.).

Para las secuencias que no pudieron ser resueltas estos valores corresponden a los tiempos consumidos hasta el momento en que la memoria de la máquina se vio colapsada.

Inst.	$n \times m$	**Val. Opt.**	**E**	**CS**	**Cant. Sec**		**Elim. (%)**		
					#	%	**R**	**C**	**D**
$LA01$	10×5	666	2	666	44	10	70	20	0
$LA02$		655	2	655	29	7	48	45	0
$LA03$		597	50	603	1105184	13	67	14	6
$LA04$		590	44	590	712155	13	68	13	6
$LA05$		593	2	593	40	7	78	15	0
$LA06$	15×5	926	2	926	30	3	70	27	0
$LA07$		890	2	890	30	3	70	27	0
$LA08$		863	2	863	58	5	79	16	0
$LA09$		951	2	951	119	8	51	41	0
$LA10$		958	2	958	130	8	58	34	0
$LA11$	20×5	1222	2	1222	182	6	55	39	0
$LA12$		1039	2	1039	93	4	74	22	0
$LA13$		1150	2	1150	150	5	58	37	0
$LA14$		1292	2	1292	192	6	58	36	0
$LA15$		1207	2	1207	185	5	70	25	0
$FT20$		1165	11	1165	3101	6	71	21	2
$LA16$	10×10	945∗	38	979	188745116	15	80	1	4
$LA17$		784	100	797	310794987	11	80	5	4
$LA18$		848∗	46	856	306944717	14	80	3	3
$LA19$		842	93	842	173674258	11	80	6	3
$LA20$		902∗	42	945	308104416	14	80	2	4
$FT10$		930∗	31	958	364392613	14	79	3	4
$N1$	40×5	$2249^{\dagger}$	2	2249	834	3	53	44	0
$N2$	60×5	$3191^{\dagger}$	2	3191	1747	2	53	45	0

Cuadro 3: Resultados JSSP-Exacto

Se observan los siguientes hechos:

- Se han podido resolver de forma óptima todas las instancias con menos de 10 máquinas. Respecto de las instancias de tamaño 10×10, sólo se ha podido hallar el valor óptimo para LA17 y LA19, mientras que en todo los demás casos el algoritmo ha agotado los recursos del sistema.

- En la mayor parte de los casos, la cota superior calculada coincide con

Inst.	$n \times m$	Tiempo (s)				Tiempo (%)			
		[6]	Tot.	CS	CP	Exp.	Cot.	Ord.	Dom.
$FT06$	6×6	0	1	1	0	100	0	0	0
$LA01$		1533	2	2	0	100	0	0	0
$LA02$		1961	10	10	0	100	0	0	0
$LA03$	10×5	1206	20	11	9	22	33	22	22
$LA04$		1629	14	8	6	17	33	33	17
$LA05$		965	1	1	0	100	0	0	0
$LA06$			4	4	0	100	0	0	0
$LA07$			4	4	0	100	0	0	0
$LA08$	15×5		4	4	0	100	0	0	0
$LA09$			5	5	0	100	0	0	0
$LA10$			5	5	0	100	0	0	0
$LA11$			8	8	0	100	0	0	0
$LA12$			10	10	0	100	0	0	0
$LA13$	20×5		9	9	0	100	0	0	0
$LA14$			9	9	0	100	0	0	0
$LA15$			8	8	0	100	0	0	0
$FT20$			95	95	0	100	0	0	0
$LA16$				43	2340	10	37	44	10
$LA17$			3797	37	3760	11	34	39	15
$LA18$	10×10			41	3568	10	36	43	11
$LA19$			1948	33	1915	12	37	36	16
$LA20$				41	7554	7	36	41	16
$FT10$				47	8109	9	27	52	12
$N1$	40×5		81	81	0	100	0	0	0
$N2$	60×5		224	224	0	100	0	0	0

Cuadro 4: Tiempos JSSP-Exacto

el valor del óptimo. Dicho de otro modo, la heurística inicial encuentra una solución óptima. Esto hace que el algoritmo exacto se quede sin nodos para expandir en etapas tempranas y reduce enormemente el tiempo de ejecución.

- Aún cuando la cota superior no coincide con el óptimo, es siempre menor al 105 % del valor óptimo. Con excepción de la instancia generada LA11+LA12+FT20, esta cota fue calculada en menos de 2 minutos en todos los casos.

- La mayor parte de las instancias con menos de 10 máquinas se resolvieron en tiempos menores a los 20 segundos, con la excepción de $LA20$, que consumió 95 s, y de las instancias $N1$ y $N2$, que requirieron 81 y 224 segundos respectivamente. Debe tenerse en cuenta que estas últimas contenían gran número de jobs.

- Globalmente, se han eliminado un 87 % de los nodos generados. De este 87 %, un 80 % corresponde a secuencias con retrasos, un 4 % a dominancia y el 3 % restante a desigualdad de cotas. Es importante señalar que estas diferencias se deben mayormente al orden en que se realizan las comparaciones. En experimentos realizados anteponiendo la poda por cotas, ésta produce la mayor parte de los descartes.

- En [6] sólo se exponen resultados de las instancias LA01-LA05 y FT06, con un tiempo de procesamiento total de 7294 segundos (Ver *Tiempo (s) en [6]* en el Cuadro 4). Con nuestro algoritmo, la resolución de dichas instancias se obtiene en un total de 48 segundos (Ver *Tiempo (s) Tot.* en el Cuadro 4).

  Cabe resaltar que las características del sistema utilizado en [6] son mejores que las de la máquina que utilizamos para nuestros experimentos, lo que enfatiza las virtudes de las mejoras incorporadas. Esencialmente: la ampliación de los criterios de detección de operaciones con retrasos y la introducción de la poda por cotas.

- Las instancias $N1$ y $N2$ pudieron resolverse sin problema alguno, pese a la gran cantidad de jobs con respecto a las demás instancias. Además, estas instancias no superaron la etapa 2 del ciclo principal, por lo que se podaron todas las secuencias por desigualdad de cotas.

- El bloque que mayor tiempo ocupó en todas las instancias se corresponde con el ordenamiento de la etapa, con 12201 segundos. Luego, le sigue el tiempo correspondiente a la eliminación por cotas, que consumió 9026 segundos. Prácticamente, la totalidad de éstos se da en las instancias LA16-LA20 y FT10. Observemos que en estos casos la diferencia entre la cota superior y el óptimo es grande, por lo que se eliminan menos secuencias y, por ende, se deben ordenar una mayor cantidad de éstas.

Podemos concluir que el algoritmo resuelve eficazmente instancias con un número de máquinas reducido, pero agota los recursos en instancias con 10 máquinas, aún descartando un 87 % de las secuencias parciales. Esto no hace más que corroborar el cáracter exponencial del problema.

Además, se observa que las instancias en donde la cantidad de máquinas es 5, las secuencias son podadas en las etapa iniciales. En estas instancias no solo la cota alcanza el óptimo, sino que la gran diferencia entre la cantidad de jobs y la cantidad de máquinas tiene como efecto que las cotas inferiores de las secuencias también alcancen este valor. Por esto, la totalidad del árbol es podado apenas comienza el ciclo principal del algoritmo.

Es importante resaltar que la eliminación por cotas no es considerada en [6], sino que es introducida en esta tesis. Los experimentos numéricos resaltan la importancia de este método para mejorar la poda de secuencias parciales no-óptimas. La precisión de las cotas resulta decisiva en los tiempos de procesamiento y capacidad de resolución de todas las instancias, más alla de las pequeñas. Basta ver la comparación de tiempos con el trabajo original para corroborar que la eliminación por cotas reduce los tiempos sustancialmente.

7.2. Heurística

A las instancias estudiadas con el algoritmo exacto se agregaron las instancias $LA21$ a $LA30$, y las instancias $SWV01$, $SWV02$, $YN01$ y $YN02$, para ejecutar los experimentos correspondientes al algoritmo heurístico. En los Cuadros 5 a 8 se muestran los resultados obtenidos para los diferentes valores de α, β. V.O. indica el mejor valor conocido para estas instancias. Para cada α se indican: el valor de la mejor solución encontrada (V), el error relativo (E), y el tiempo de ejecución (T), medido en segundos.

Cabe resaltar que para las últimas cuatro instancias $SWV01$, $SWV02$, $YN01$, $YN02$, el valor de la cota superior inicial se realizó en una sola corrida con ancho de banda 500.

Haremos las siguientes observaciones respecto de los resultados obtenidos:

- En el Cuadro 5 podemos ver cómo el incremento en el ancho de ventana aumenta la precisión del método, así también como dispara los tiempos de ejecución. En particular, para un ancho de ventana de 500000 la heurística tiene un error relativo prácticamente nulo.

- Por otro lado, en los Cuadro 6 a 8, notamos que en algunas instancias un aumento en el ancho de ventana no sirve para acercarnos al mejor valor hasta la actualidad. Es decir: en las instancias en donde el error relativo es muy grande, el incremento del ancho de ventana no logra reducir este error. Entendemos que esto se debe a particularidades de la estructura de las instancias, que hacen que nuestro criterio de selección deje afuera las subsecuencias que conducen al óptimo, incluso para anchos de ventana grandes. A este respecto, podrían evaluarse nuevos criterios de selección.

Inst	$n \times m$	α	**V.O.**	**V**	**E**	**T**
$LA16$	10×10	10000	945	979	0,04	112
$LA17$			784	797	0,02	98
$LA18$			848	853	0,01	111
$LA19$			842	842	0	91
$LA20$			902	912	0,01	118
$LA16$		50000	945	979	0,04	453
$LA17$			784	784	0	371
$LA18$			848	848	0	417
$LA19$			842	842	0	334
$LA20$			902	911	0,01	484
$LA16$		100000	945	975	0,03	882
$LA17$			784	784	0	693
$LA18$			848	848	0	803
$LA19$			842	842	0	608
$LA20$			902	902	0	962
$LA16$		500000	945	947	0	4449
$LA17$			784	784	0	2823
$LA18$			848	848	0	3735
$LA19$			842	842	0	2112
$LA20$			902	902	0	4225

Cuadro 5: Corridas del algoritmo heurístico para $\beta = 5$

Inst	$n \times m$	**V.O.**	**V**	**E**	**T**
$LA21$	15×10	1046	1119	0,07	372
$LA22$		927	1033	0,11	354
$LA23$		1032	1032	0	126
$LA24$		935	958	0,02	346
$LA25$		977	1008	0,03	324
$LA26$	15×10	1218	1262	0,04	758
$LA27$		1235	1324	0,07	770
$LA28$		1216	1287	0,06	788
$LA29$		1152	1345	0,17	678
$LA30$		1355	1411	0,04	733

Cuadro 6: Corridas del algoritmo heurístico para $\beta = 10$ y $\alpha = 10000$.

- Para la instancia $LA27$, se observa que, extrañamente, el mejor valor devuelto se corresponde con el menor ancho de ventana. $\alpha = 10000$. En $LA29$ sucede lo mismo cuando pasamos de $\alpha = 50000$ a $\alpha = 100000$.

Inst	$n \times m$	**V.O.**	**V**	**E**	**T**
$LA21$		1046	1085	0,04	1498
$LA22$		927	994	0,07	1468
$LA23$	15×10	1032	1032	0	127
$LA24$		935	958	0,02	1405
$LA25$		977	1004	0,03	1341
$LA26$		1218	1227	0,01	3212
$LA27$		1235	1386	0,12	3043
$LA28$	15×10	1216	1256	0,03	3318
$LA29$		1152	1303	0,13	2798
$LA30$		1355	1359	0	2993
$SWV06$	20×15	1675	1829	0,09	6616
$SWV07$		1594	1715	0,08	6374
$YN01$	20×20	884	957	0,08	7835
$YN02$		904	1064	0,18	7041

Cuadro 7: Corridas del algoritmo heurístico para $\beta = 10$ y $\alpha = 50000$.

Inst	$n \times m$	**V.O.**	**V**	**E**	**T**
$LA21$		1046	1064	0,02	3332
$LA22$		927	971	0,05	2971
$LA23$	15×10	1032	1032	0	126
$LA24$		935	948	0,01	2827
$LA25$		977	1004	0,03	2762
$LA26$		1218	1223	0	6367
$LA27$		1235	1386	0,12	6110
$LA28$	15×10	1216	1256	0,03	6505
$LA29$		1152	1339	0,16	6096
$LA30$		1355	1359	0	6218

Cuadro 8: Corridas del algoritmo heurístico para $\beta = 10$ y $\alpha = 100000$.

En estos casos, debido al sesgo producido por la elección de α, la comparación entre el promedio de las cotas inferiores de las secuencias se realiza entre un menor número de secuencias. Luego, es posible (aunque improbable) que para un valor de ancho de ventana pequeño, algunas secuencias no sean eliminadas, mientras que para anchos de ventana más grande, sí sean descartadas. De estas secuencias parciales se obtiene un mejor resultado.

- De las instancias de tamaño 15×10, la instancia $LA23$ se resuelve

de forma óptima en muy poco tiempo en comparación con las otras debido a que la cota superior coincide con el óptimo del problema. Nuevamente, esto tiene que ver con la estructura de la instancia, que permite que nuestro criterio de selección encuentre una cota superior que coincide con el óptimo.

- A medida que el tamaño de las instancias aumenta nos vimos obligados a omitir corridas de la heurística, dado que valores altos de α consumían todos los recursos del sistema. Esto pone de manifiesto que la elección de los parámetros condiciona fuertemente la efectividad de la heurística.

- Las instancias $SWV01$, $SWV02$, $YN01$ y $YN02$, a diferencias de las anteriores, poseen una mayor cantidad de máquinas. En éstas, no sólo los tiempos de ejecución son considerables, sino que la cota superior calculada se encuentra muy alejada del mejor valor hasta la actualidad. Además, al aumentar el ancho de ventana hasta 50000, no se ven mejoras considerables.

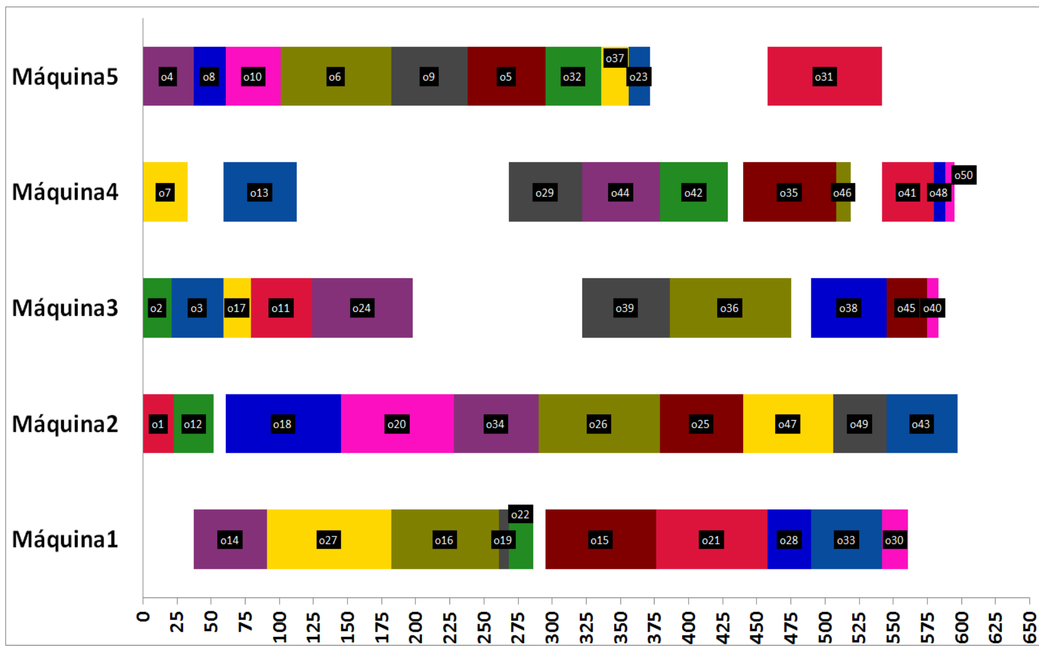
- Recordemos, por último que para la mayor parte de las instancias con menos de 10 máquinas el ciclo heurístico introducido en el algoritmo exacto bastaba para encontrar la solución óptima. Es decir que en todos esos casos la heurística funcionaba bien incluso con anchos muy pequeños de ventana.

En conclusión, la heurística tiene la capacidad de resolver de forma casi óptima las instancias de tamaño inferior a 10×10 expuestas anteriormente. No obstante, conforme se incrementa dicho tamaño, su efectividad disminuye y los tiempos de procesamiento resultan excesivos. Ya en instancias de 15 jobs y 10 máquinas, un parámetro $\alpha = 500000$ provoca que el algoritmo heurístico sea totalmente ineficiente.

Debe tenerse en cuenta que la performance mejora significativamente cuando el número de máquinas es pequeño, aún con muchos jobs.

Es importante aclarar que, más allá del tamaño del problema, algunas instancias tienen una estructura de tiempos de ejecución particularmente *maligna*. En estos casos, el aumento del parámetro α no mejora significativamente la cota superior inicial, ocasionando un importante incremento de los tiempos de ejecución, que no redunda en una disminución apreciable del error relativo.

Por último, en la Figura 18 se pueden visualizar una solución óptima y la solución heurística que nos da la cota superior para la instancia LA03. Al comparar ambos schedules, resulta interesante ver que schedules tan diferentes pueden resultar tener tiempos de finalización similares.



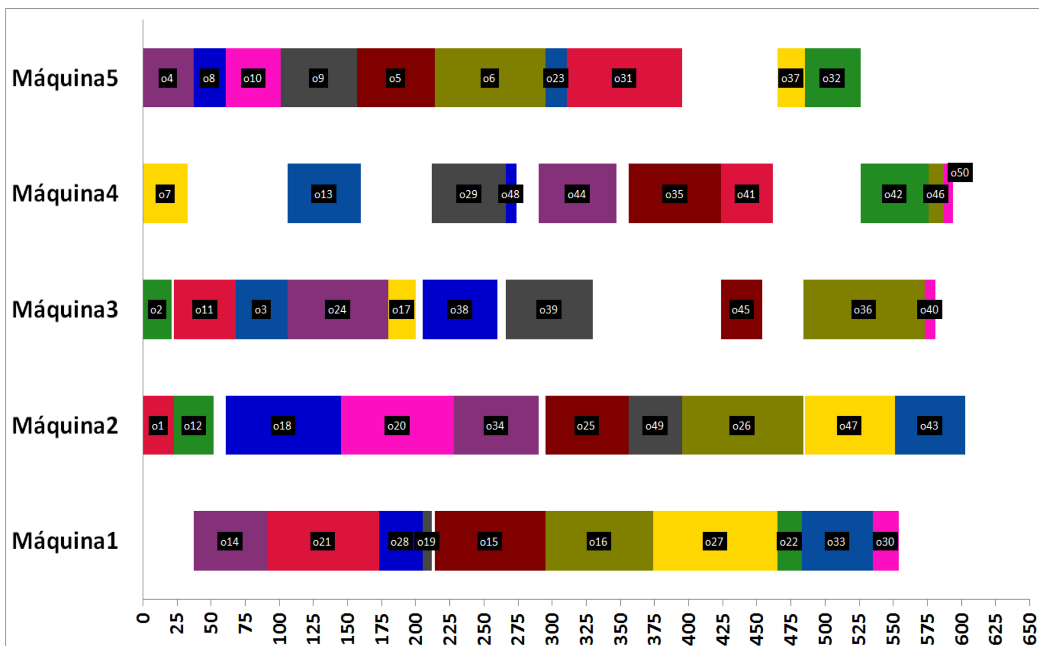


Figura 18: Schedule óptimo (arriba) vs. Schedule Heurístico(abajo)

8. Conclusiones y comentarios finales

En este trabajo se ha desarrollado un algoritmo exacto y un algoritmo heurístico en base a la idea propuesta en [6].

Respecto del algoritmo exacto, se ha logrado demostrar su validez formalmente, enmendando, en colaboración con J. Gromicho y J. van Hoorn, algunos errores que se encontraron en las demostraciones originales. Además, se ha estimado una cota para su complejidad que corrobora que el JSSP es un problema exponencial. Los resultados experimentales realizados confirman este hecho, aunque los criterios de poda utilizados reducen considerablemente el número de secuencias parciales analizadas con respecto al análisis del peor caso.

Respecto del algoritmo heurístico, se ha logrado hallar una función de valuación aceptable. En instancias de menos de 10 máquinas, la composición de parámetros $\alpha = 500000, \beta = n$ ha logrado hallar los óptimos en casi la totalidad de los casos, siendo el máximo error relativo 0,9 %. No obstante, si aumentamos el tamaño de las instancias, el rendimiento del algoritmo comienza a decaer, tanto en precisión como en efectividad.

Cabe mencionar que en la actualidad existen métodos heurísticos mucho más precisos y efecientes. Estas heurísticas, a partir de una solución inicial, generan un número soluciones factibles mediante diferentes metaheurísticas, como ser búsqueda tabú, recocido simulado, algoritmos genéticos, entre otras. De esta forma, se pueden construir un gran número de soluciones sin incurrir en grandes costos computacionales, lo que permite disponer todos los recursos en la precisión de la búsqueda.

Por poner un ejemplo reciente, en [5] los autores proponen una heurística llamada *BRKGA-JSP*, que como base combina un algoritmo genético con un método gráfico para obtener excelentes resultados. BRKGA-JSP resuelve las mismas instancias que el algoritmo que aquí estudiamos de forma óptima en un máximo de 14 segundos. Además, con BRKGA-JSP se han reducido los mejores valores para instancias de tamaño grande (40×20 ó 50×20) en tiempos aceptables. En comparación con este tipo de algoritmos, el algoritmo propuesto no resulta para nada competitivo.

En contraposición, nuestro algoritmo debe generar un conjunto de secuencias parciales para cada etapa, desde secuencias con una operación programada hasta secuencias completas. Así, los resultados obtenidos dependen en gran medida de la capacidad de reducir el árbol de búsqueda en el mayor grado posible sin perder demasiada precisión. Esto requiere hallar una función de valuación mucho más eficiente que la propuesta en este trabajo.

Es por este motivo que la mayor parte de las técnicas heurísticas propuestas en los últimos años para resolver el JSSP evitan la generación de solucio-

nes parciales, enfocándose en encontrar buenas soluciones dentro del espacio de búsqueda. Algoritmos como el BRKGA-JSP se constituyen a través de la variación y combinación de las mejores prácticas conocidas, a la vez que la gran cantidad de componentes hace que la implementación resulte bastante más compleja que el algoritmo heurístico propuesto. Desde este punto de vista, se podría decir que esta clase de algoritmos se encuentran en otro nivel de madurez.

En conclusión, las variantes heurísticas del algoritmo de [6] que aquí estudiamos resultan obsoletas en comparación con la mayoría de los algoritmos utilizados actualmente, aunque abren un abanico de caminos y variantes aún no explorados.

A. Contraejemplo

Tanto la formulación del problema de job-shop scheduling como un problema de secuenciamiento, como el algoritmo de programación dinámica basado en este planteo son desarrollados en [6].

Para probar que el algoritmo encuentra efectivamente una solución óptima, en el artículo se introducen diversos conceptos (algunos de los cuales se retoman en esta tesis) y se prueban varios resultados preparatorios.

Sin embargo, hemos encontrado algunos problemas que llevan a la conclusión de que varios de los resultados preliminares de [6] resultan ser falsos. Esto invalida la argumentación propuesta en el artículo para la demostración de corrección del algoritmo.

En este apéndice exponemos un ejemplo que muestra los principales errores de la demostración propuesta en [6]. No reproducimos aquí el conjunto de los argumentos del artículo, sino que nos limitamos a comentar los puntos esenciales para poner de relieve sus principales fallas.

En la Sección 4 introdujimos el conjunto $\mathring{S}(Q)$: dado $Q \subset \mathcal{O}$, $\mathring{S}(Q)$ contiene todas las secuencias ordenadas de operaciones en Q. En [6] se definen además dos subconjuntos de $\mathring{S}(Q)$. El primero, $\widehat{S}(Q)$ está formado por todas las secuencias de $\mathring{S}(Q)$ que no son dominadas por ninguna otra secuencia de $\mathring{S}(Q)$. Por otro lado, $\overset{\triangle}{S}(Q)$ contiene todas las secuencias $s \in \widehat{S}(Q)$ tales que todas las subsecuencias de s pertenecen a $\widehat{S}(Q')$ para el correspondiente conjunto Q'. En otras palabras: $\overset{\triangle}{S}(Q)$ es el conjunto de secuencia *nunca* dominadas por *ninguna otra secuencia ordenada*.

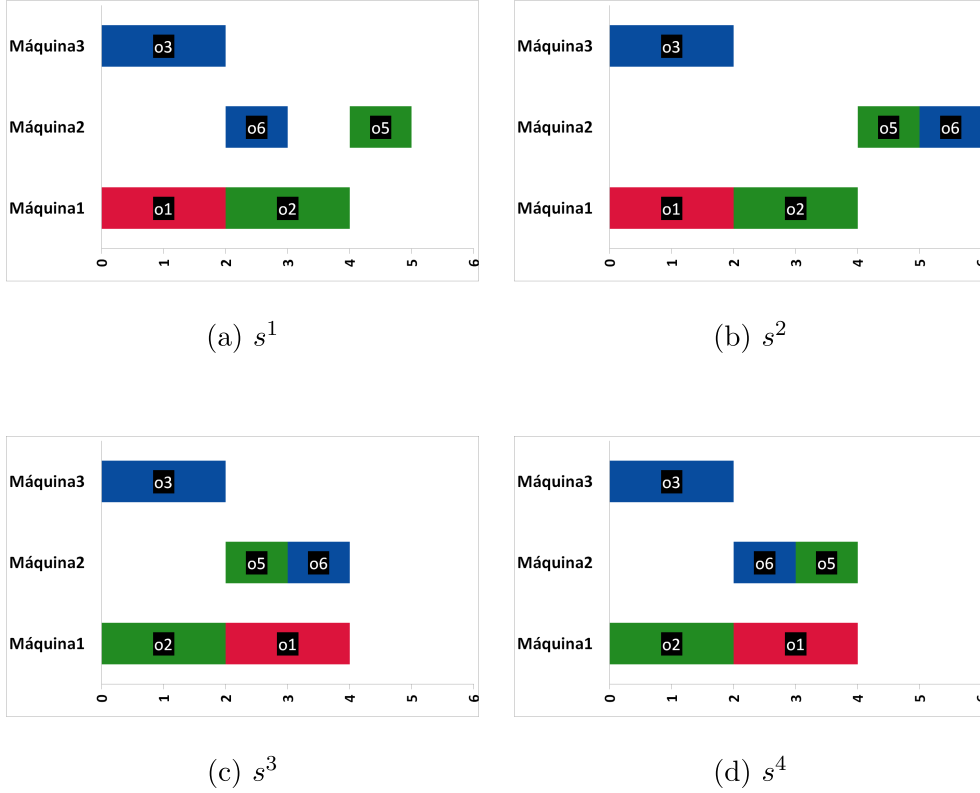
La Proposición 3 de [6] establece que $\overset{\triangle}{S}(Q)$ no es vacío para ningún Q. A partir de este resultado, la Proposición 4 concluye que los conjuntos efectivamente generados por el algoritmo son exactamente los conjuntos $\overset{\triangle}{S}(Q)$. Ambos resultados son falsos.

Nuestro ejemplo está dado precisamente por la instancia $\mathcal{I}$, introducida en la Sección 4. Recordemos que esta instancia está dada por la tabla:

Operaciones	o_1	o_2	o_3	o_4	o_5	o_6	o_7	o_8	o_9
$p(o)$	2	2	2	4	1	1	1	3	3
$m(o)$	1	1	3	3	2	2	2	3	1

Consideremos, para esta instacia, el conjunto $Q = \{o_1, o_2, o_3, o_5, o_6\}$. Es fácil verificar que $\mathring{S}(Q)$ está dado por las cuatro secuencias:

$$s^1 = (o_1, o_3, o_6, o_2, o_5), \quad s^2 = (o_1, o_3, o_2, o_5, o_6),$$
$$s^3 = (o_2, o_3, o_5, o_1, o_6), \quad s^4 = (o_2, o_3, o_6, o_1, o_5),$$



(a) s^1 (b) s^2

(c) s^3 (d) s^4

Figura 19: Secuencias en el conjunto $\mathring{S}(Q)$

representadas por los siguientes diagramas:

Para estas secuencias, tenemos:

$$\begin{array}{llll} \xi(s^1, o_4) = 6 & \xi(s^2, o_4) = 6 & \xi(s^3, o_4) = 8 & \xi(s^4, o_4) = 8 \\ \xi(s^1, o_8) = 8 & \xi(s^2, o_8) = 8 & \xi(s^3, o_8) = 6 & \xi(s^4, o_8) = 7 \\ \xi(s^1, o_9) = 7 & \xi(s_2, o_9) = 9 & \xi(s^3, o_9) = 7 & \xi(s^4, o_9) = 7 \end{array}$$

De modo que concluimos que s^1 domina a s^2 y s^3 domina a s^4. Es fácil verificar que las subsecuencias de s^1 y s^3 nunca son dominadas, por lo que pertenecen a los correspondientes conjuntos $\widehat{S}(Q')$, lo que permite concluir que

$$\overset{\triangle}{S}(Q) = \{s^1, s^3\}.$$

Ahora consideremos el conjunto $Q \cup \{o_9\}$. El algoritmo puede generar secuencias con este conjunto expandiendo apropiadamente secuencias con distintos conjuntos de cardinal $|Q|$. Particularmente los posibles conjuntos son Q, $(Q \cup \{o_9\}) \setminus \{o_5\}$, $(Q \cup \{o_9\}) \setminus \{o_6\}$ and $(Q \cup \{o_9\}) \setminus \{o_1\}$. Sin embargo, es sencillo hacer un análisis eshaustivo para comprobar que sólo Q admite

expansiones en secuencias ordenadas. En conclusión, el algoritmo sólo generará las secuencias: $s^1 + o_9$ y $s^3 + o_9$.

Para estas secuencias tenemos que:

$$\begin{array}{cc} \xi(s^1 + o_9, o_4) = 11 & \xi(s^3 + o_9, o_4) = 8 \\ \xi(s^1 + o_9, o_8) = 8 & \xi(s^3 + o_9, o_8) = 10, \end{array}$$

lo que significa que ninguna de estas secuencias será descartada por el algoritmo.

Sin embargo, la secuencia $s^4 + o_9$ es ordenada y por lo tanto pertenece a $\mathring{S}(Q \cup \{o_9\})$. El cómputo del funcional ξ para $s^4 + o_9$ da como resultado:

$$\begin{array}{c} \xi(s^4 + o_9, o_4) = 8 \\ \xi(s_4 + o_9, o_8) = 7. \end{array}$$

Es decir que $s^4 + o_9$ domina tanto a $s^1 + o_9$ como a $s^3 + 9$. Dos grandes conclusiones pueden sacarse de este hecho: La primera es que $\overset{\triangle}{S}(Q \cup \{o_9\})$ es vacío, lo que contradice la Proposición 3 de [6]. La segunda es que el algoritmo *no* generará el conjunto $\overset{\triangle}{S}(Q)$, como declara la Proposición 4: como s^4 no será expandida, $s^4 + o_9$ nunca será generada y, por lo tanto, no estará disponible para realizar comparaciones y $s^1 + o_9$ y $s^3 + o_9$ no serán descartadas. En resumen, tenemos que el algoritmo generará el conjunto: $\{s^1 + o_9, s^3 + o_9\}$ y continuará expandiendo estas secuencias, pese a que $\overset{\triangle}{S}(Q \cup \{o_9\}) = \emptyset$.

Como señalamos anteriormente, este ejemplo invalida el camino utilizado en [6] para probar que el algoritmo propuesto es correcto. La argumentación desarrollada en esta tesis, que aparecerá publicada en [7], salva estos inconvenientes.

Referencias

[1] Bellman R. *The theory of Dynamic Programming.* The Rand Corporation, Santa Monica, California.

[2] Fisher, H., Thompson, G. *Probabilistic learning combinations of local job-shop scheduling rules*, in: Muth, J., Thompson, G. (Eds.), Industrial Scheduling. Prentice Hall, Englewood Cliffs, New Jersey, pp. 225–251.

[3] Garey, M.R., Johnson, D.S. *Computers and Intertracability: A Guide to the Theory of NP-Completeness*, W.H. Freeman, San Francisco, California.

[4] Gholami O., Sotskov Y.N., Werner F. *Fast Edge-Orientation Heuristics for Job-Shop Scheduling Problems with Applications to Train Scheduling.* International Journal of Operational Research Nepal 2(1):19-32.

[5] Gonçalves, J.F., Resende, M.G.C. *An extended Akers graphical method with a biased random-key genetic algorithm for Job-Shop Scheduling.* International Transactions in Operational Research(2013), DOI: 10.1111/itor.12044

[6] Gromicho J., Saldanha da Gama F., Timmer, G., van Hoorn, J. *Solving the job-shop scheduling problem optimally by dynamic programming.* Computers and Operations Research (2012), DOI:10.1016/j.cor.2012.02.024.

[7] Gromicho, J., Nogueira, A., Ojea, I, van Hoorn, J.; *A note on the paper: "Solving the job-shop scheduling problem optimally by dynamic programming."*. En redacción.

[8] Held M., Karp R.M. *A dynamic programming approach to sequencing problems.* SIAM Journal of Applied Mathematics 1962;10:196-210.

[9] Lawrence, S. *Resource constrained project scheduling: an experimental investigation of heuristic scheduling techniques* (Supplement).

[10] Sabuncuoglu I., Bayiz M. *Job shop scheduling with beam search.* European Journal of Operational Research 118 (1999) 390-412.

[11] Sprecher, A., Kolisch, R., Drexl A.; *Semi-active, active and non-delay schedules for the resource- constrained proyect scheduling problem* (1995); Eur. J. of Op. Res., 80, p. 94-102.