Algoritmos heurísticos y el problema de job shop scheduling

Algoritmos heurísticos y el problema de job shop scheduling

Profesores Dr Fabio Vicentini Dra Susana Puddu
Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Bs As, enero 2003

Germán Mailing

Introducción

Este texto aborda la problemática de la optimización combinatoria. Esta es una rama de la matemática discreta orientada a la resolución de problemas de la vida real que involucren hallar un óptimo de una función de costo sobre un conjunto discreto.

La optimización combinatoria, nacida a mediados del siglo XX, se ha desarrollado enormemente hasta la actualidad y hoy tiene importantes aplicaciones en el nivel industrial, informático, científico, etc. y en todos los ámbitos en que se presenten situaciones en que se requiera minimizar costos, demoras, recorridos, etc.

El estudio de los problemas de optimización combinatoria se centra indudablemente en hallar métodos o algoritmos que procuren la resolución efectiva de los mismos. Pero surge inmediatamente la pregunta ¿es siempre posible encontrar algoritmos que en circunstancias del mundo real, en el aquí y ahora, posean la facultad de concedernos soluciones óptimas?

La respuesta no es muy alentadora, muchos problemas de optimización combinatoria se encuentran contenidos en la clase NP-hard o problemas en los que el tiempo de cómputo necesario para resolverlos crece desproporcionadamente conforme aumenta el tamaño del problema. Para hacernos una idea acerca de este comportamiento, diremos que se traduce en inconvenientes severos a la hora de conciliar el tratamiento de problemas de mediana envergadura y la aspiración de conocer la solución en el término de nuestra vida natural.

Esto nos obliga a buscar un nuevo ángulo para el tratamiento y el que se presenta como más satisfactorio es aquel que parece venir inspirado en una analogía con el problema de hallar raíces de polinomios. Como es sabido, hallar raíces exactas es harto laborioso pero aproximarlas puede hacerse mucho más sencillamente mediante procedimientos iterativos.

La idea entonces, es comprometer la calidad de las soluciones aspirando a soluciones cuasi-óptimas en pos de asegurar la posibilidad de alcanzarlas en intervalos de tiempo admisibles. Por lo tanto, reformulamos la pregunta anterior de la siguiente manera, ¿es posible encontrar algoritmos que en circunstancias reales, encuentren “buenas” soluciones?

Aquí, la respuesta es afirmativa. En primer lugar, debemos destacar que este nuevo enfoque que hemos ensayado se ha consolidado como campo de estudio científico bajo el nombre de *heurística* y ha traído aparejado el desarrollo de nuevos, potentes y más sofisticados algoritmos y a la vez, de nuevas herramientas que posibilitasen medir y analizar los parámetros emanados de esos últimos.

Hasta nuestros días, perduran los esfuerzos constantes por experimentar algoritmos más veloces, que aporten soluciones más precisas a problemas cada vez más complejos que reportan continuamente las diversas actividades humanas.

Nuestro itinerario consta de varias partes. En el capítulo I se da una visión global del tema: historia, problemas arquetípicos, algoritmos para resolverlos, análisis de complejidad y eficiencia. El capítulo II se centra en un problema en particular: el *job shop scheduling* y se esboza la base teórica para su tratamiento. En el capítulo III se presenta un algoritmo de búsqueda local iterativa para manipular el problema expuesto en el capítulo precedente. En el capítulo IV se introduce un algoritmo alternativo de tipo constructivo con el mismo fin. Finalmente, el capítulo V se basa en el estudio estadístico de los algoritmos anteriores.

Capítulo I

1.1 Generales

En esta sección nos ocuparemos de problemas de optimización combinatoria. Tales problemas aparecen en situaciones que involucran elecciones entre discretas alternativas y solucionarlos equivale a encontrar una solución óptima entre esas opciones. Optimalidad que debe ser entendida de acuerdo a cierto criterio de costo que nos provea de una medida cuantitativa de la calidad de cada solución.

Es decir, estos problemas residen en encontrar en un conjunto de esquemas permitidos, aquel que tenga un costo asociado óptimo.

La naturaleza de estos problemas hace que rara vez pueden ser tratados teóricamente. Por lo tanto, nos vemos motivados a eludir el cálculo directo y considerar la utilización de ordenadores y algoritmos específicos que nos auxilien en nuestra tarea.

Pero aquí aparece una dificultad imprevista, la cual radica en el tiempo que demora el ordenador en encontrar la solución puesto que, por desgracia, muchos problemas de optimización combinatoria pertenecen a la clase NP-hard (Garey y Johnson, 1979). Es decir, el tiempo de cómputo que se requiere para resolver uno de estos problemas se incrementa conforme crece el tamaño del problema presentando una dependencia funcional tal que no admite ser acotada por un polinomio.

En consecuencia, nuestra esperanza se centra en torno a algoritmos que encuentran soluciones cuasi-óptimas en tiempos de aplicación razonables. Es decir, pactamos comprometer la calidad de la solución a cambio de obtener el acceso a un resultado en tiempo viable.

La literatura distingue entre dos clases principales de algoritmos de aproximación: *constructivos (constructives)* y *de búsqueda local (local search)*. Nos detendremos principalmente sobre estos últimos.

Hay muchas aplicaciones particulares de búsqueda local pero todas descansan sobre la utilización del concepto de *entorno (neighborhood)*, el cual es usado para guiar la búsqueda iterativa hacia buenas soluciones.

El uso de la búsqueda local en problemas de optimización combinatoria se remonta a 1950 cuando surge el primer algoritmo de intercambio de arcos (edge-exchange) para resolver el problema del viajante (travelling salesman problem). En los años subsiguientes el alcance de la búsqueda local se amplió hasta incluir aplicaciones a problemas de toda índole.

Muchas variantes de búsqueda local han sido propuestas basadas en analogías con procesos de la naturaleza, como *simulated annealing (templado simulado)*, *genetic algorithms (algoritmos genéticos)* y algunos ejemplos de *neural networks (redes neuronales)*. El incremento de recursos de cómputo de los ordenadores y la sofisticación de las estructuras de datos han posibilitado empleo de estos algoritmos para resolver complejos problemas de la vida real. Se han modelizado matemáticamente y obtenido consistentes resultados teóricos en cuanto a su complejidad y eficiencia. El ejemplo más difundido es el simulated annealing, modelizado a partir de cadenas de Markov (Aarts y Korst, 1989).

1.2 Formulación y modelo

Definición: Un problema de optimización combinatoria queda definido por un conjunto de instancias del problema y una prescripción de maximizar o minimizar.

Consideraremos sin pérdida de generalidad únicamente problemas de minimización. De lo contrario, si f es la función de costo y S el conjunto de soluciones usamos que

$$\max \{f(x) : x \in S\} = -\min \{-f(x) : x \in S\}$$

Definición: Una instancia de un problema de optimización combinatoria es un par (S,f) donde S es el conjunto de soluciones factibles y $f:S\rightarrow\mathbb{R}$ es la función de costo. El problema reside en encontrar una solución óptima global, esto es, un $i^*\in S$ tal que $f(i^*) \leq f(i)$ para todo $i\in S$. Además, si un $f^*=f(i^*)$ entonces $S^*=\{i\in S: f(i)= f^*\}$ denota el conjunto de soluciones óptimas.

Una instancia generalmente no está dada explícitamente, esto es, expuesta llanamente como una lista de soluciones con sus costos respectivos. Usualmente tenemos una representación compacta de los datos formada por un algoritmo de tiempo polinomial para verificar si una solución pertenece a S y computar su costo (Garey y Johnson, 1979).

Definición: Sea (S,f) una instancia de un problema de optimización combinatoria. Una función entorno es una aplicación $N:S\rightarrow\mathcal{P}(S)$ que define para cada solución $i\in S$ un conjunto $N(i)\subseteq S$ de soluciones que están, en algún sentido, cerca de i. El conjunto N(i) es el entorno de la solución i, y cada $j\in N(i)$ es un vecino de i. Asumiremos que $i\in N(i)$ para todo $i\in S$.

En líneas generales, un algoritmo de búsqueda local comienza con una solución inicial y trata continuamente de encontrar solución óptima revisando entornos. Una versión básica de búsqueda local es el *iterative improvement (progreso iterativo)* que comienza con una solución inicial y busca en su entorno una solución de menor costo. Si una tal es encontrada se reemplaza la solución actual y continúa la búsqueda. De otra manera el algoritmo devuelve la solución actual, la cual resulta un óptimo local.

Definición: Sea (S,f) una instancia de un problema de optimización combinatoria y sea N la función entorno. Una solución $i\in N$ es un óptimo local con respecto a N si $f(i)\leq f(j)$ para todo $j\in N$. Denotamos S_{loc} al conjunto de óptimos locales. Además, si $S_{loc}\subseteq S^*$ decimos que S es exacta.

El método iterative improvement puede aplicar *first improvement (primer progreso)* en el cual la solución actual es reemplazada por la primera solución del entorno encontrada que mejore el costo, o *best improvement (mayor progreso)* en la cual la solución actual es reemplazada por la mejor solución del entorno.

El proceso de búsqueda local puede verse como un recorrido por un grafo dirigido cuyos vértices son etiquetados de acuerdo al costo y están dados por elementos de S y cuyos arcos están dados por los pares de la forma (solución, vecino). Un ejemplo ilustrativo de búsqueda local es el algoritmo simplex (Dantzig, 1951, 1963) donde el politopo determina el grafo de entornos y las sucesivas aplicaciones de las reglas de pivote determinan el recorrido.

1.3 Entornos

Los entornos dependen del problema considerado y encontrar funciones entorno eficientes que guíen hacia óptimos locales de alta calidad se plantea como uno de los desafíos de la búsqueda local. Hasta el momento, no hay disponibles reglas generales y cada situación debe ser apreciada separadamente. La literatura ofrece muchos ejemplos de funciones entorno y a menudo, hay disponibles diversas posibilidades para un mismo problema. Para ordenar esta sección, presentaremos inicialmente una clase de funciones entorno basadas en un intercambio simple. Luego, elaborando este principio introduciremos una clase de funciones entorno más compleja.

1.3.1 Entornos de intercambio simple

En muchos problemas de optimización combinatoria las soluciones pueden ser representadas como sucesiones o particiones. Esta representación permite el uso de entornos de k-intercambios, esto es, entornos que se obtienen intercambiando k pares de elementos de una secuencia o partición dada.

Definición: Sea S un conjunto de permutaciones de una sucesión. Para todo $\pi\in S$, el entorno de k-intercambio de π, notado $N_k(\pi)$, consiste en el conjunto de soluciones que se obtienen a partir de π través de un k-intercambio. Un k-intercambio es una función $h:S\rightarrow S$ que a partir de una solución obtiene

una nueva seleccionando k pares distintos de elementos e intercambiando las posiciones de cada par ordenadamente. Es decir, para cada uno de los k pares (a,b), si i y j son tales que $\pi(i)=a$ y $\pi(j)=b$ entonces luego del intercambio tenemos $\pi(i)=b$ y $\pi(j)=a$.

Definición: (Sorting problem) Sea dado un conjunto $A=\{a_0,...,a_{n-1}\}$ de n números positivos. Encontrar un orden no creciente de los elementos de A.

En el problema de ordenamiento, S puede ser elegido como el conjunto de todas las permutaciones π de los elementos de A, donde $\pi(i)$ denota el número en la posición i-ésima del orden. La función de costo se define como

$$f(\pi) = \sum_{j=0}^{n-1} j\pi(j)$$

para todo $\pi \in S$. El entorno de k-intercambios se define como sigue. Para todo $\pi \in S$, $N_k(\pi)$ consiste en el conjunto de permutaciones en S que se obtienen de π seleccionando k pares distintos de números e intercambiando las posiciones de los elementos de cada par, esto es, para cada uno de los k pares (a,b) si p y q son tales que $\pi(p)=a$ y $\pi(q)=b$ entonces luego del intercambio tenemos $\pi(p)=b$ y $\pi(q)=a$. Puede verse que el entorno definido por k-intercambios es exacto. Asimismo, el entorno definido por un intercambio simple sobre pares adjacentes es exacto y su correspondiente algoritmo resulta el conocido *bubble sort* (Knut, 1983).

Definición: (Uniform graph partitioning problem) Dados un grafo no dirigido G=(E,R) con #V=2n y para cada rama $e \in E$ un peso $w_e \in \mathbb{R}_+$. Encontrar una partición de V en dos subconjuntos V_1 y V_2 con $\#V_1=\#V_2=n$ tal que la suma de los pesos de las ramas que tienen un extremo en V_1 y el otro extremo en V_2 sea mínima.

Aquí S puede elegirse como el conjunto de todas las particiones (V_1,V_2) de V en dos subconjuntos de igual tamaño y la función de costo se define como

$$f(V_1, V_2) = \sum_{\substack{ij \in E \\ i \in V_1 \\ j \in V_2}} w_{ij}$$

para todo $(V_1,V_2) \in S$.

Un entorno de k-intercambios se define como sigue. Para todo $(V_1,V_2) \in S$, se describe $N_k(V_1,V_2)$ como el conjunto de particiones de S que se obtienen de (V_1,V_2) seleccionado k pares distintos de vértices (u,v) con $u \in V_1$, $v \in V_2$ e intercambiándolos, esto es, moviendo u a V_2 y v a V_1.

Definición: (Traveling salesman problem) Sean dados un conjunto de n ciudades y una matriz de distancias $D \in \mathbb{R}^{n \times n}$ donde d_{ij} denota la distancia desde la ciudad i hasta la ciudad j para todo i,j=0,...,n−1. Encontrar un itinerario de mínima longitud que visite cada ciudad exactamente una vez.

En este caso, S puede ser elegido como el conjunto de todos los ciclos hamiltonianos C en el grafo completo K_n cuyos vértices corresponden a las ciudades y cuyos ramas uv tienen un peso d_{ij}. La función costo se define como

$$f(C) = \sum_{ij \in C} d_{ij}$$

para todo $C \in S$. Un entorno de k-intercambios se define como sigue. Para todo $C \in S$, se describe $N_k(C)$ como el conjunto de ciclos hamiltonianos en S que se obtienen removiendo k ramas de C y reemplazándolas con otras k ramas de K_n.

Notemos que el cardinal de los entornos de k-intercambios dados en los anteriores ejemplos es del orden $O(n^k)$ ya que en general se trata de elegir k ítems entre n. En el caso del problema del viajante hay un factor exponencial adicional originado por el número de maneras diferentes de reconectar los k caminos que quedaron vacantes luego de remover los k arcos del ciclo hamiltoniano.

Es importante resaltar la fuerte conexión entre los grafos de entornos de los distintos ejemplos, lo cual evidencia cómo una solución se alcanza a partir de otra. Este hecho es una propiedad de gran utilidad en el análisis de convergencia de los algoritmos.

De igual manera que en el caso del algoritmo de iterative improvement, cualquier algoritmo de búsqueda local tiene tres pasos básicos: generación de una solución inicial, generación de las soluciones del entorno y cálculo de los valores de costo. Para los ejemplos discutidos con anterioridad, estos pasos involucran operaciones simples que pueden ser realizadas por algoritmos eficientes. Pero esto no es generalmente cierto. Algoritmos eficientes requieren a menudo de tareas intrincadas de cómputo y pueden incluso no existir. Por ejemplo, encontrar una solución factible inicial para el problema del viajante se evidencia como un problema NP-hard. También el cálculo de costos puede resultar seriamente dificultoso.

Con el fin de ilustrar algunas de las complicaciones que pueden aflorar en el cómputo de entornos, discutiremos el job shop scheduling problem (Conway, Maxwell y Miller, 1967).

Definición: (Job shop scheduling problem) Sean dados un conjunto J de trabajos, un conjunto M de máquinas y un conjunto O de operaciones con N integrantes. Para cada operación $i \in O$ tenemos ligadas una trabajo $j_i \in J$ a la que pertenece y una máquina $m_i \in M$ en la que debe realizarse consumiendo un tiempo $p_i \in \mathbb{R}_+$. Además, es dada una relación de precedencia binaria $\prec$ que descompone O en cadenas, una para cada trabajo. Encontrar un tiempo de comienzo s_i para cada operación $i \in O$ tal que se minimiza el makespan, definido como $\max_{i \in O} (s_i + p_i)$, sujeto a las siguientes restricciones

i)	$s_i \geq 0$	para todo $i \in O$
ii)	$s_j \geq s_i + p_i$	para todo $i,j \in O$ con $i \prec j$
iii)	$s_j \geq s_i + p_i$ o $s_i \geq s_j + p_j$	para todo $i,j \in O$ con $m_i = m_j$

La restricción i) implica que ninguna máquina está disponible antes del instante 0.

La restricción ii) da cuenta de la relación de precedencia, esto es, si $i \prec j$ entonces j no puede comenzar antes que i finalice. Finalmente la restricción iii) implica que ninguna máquina puede procesar dos operaciones al mismo tiempo.

1.3.2 Entornos de intercambios complejos

Un punto central en lo que concierne a la búsqueda local es la observación del hecho desafortunado de que los algoritmos pueden quedar atascados en un óptimo local pobre. Es ampliamente conocido que para pequeños valores de k, los entornos de k-intercambios pueden ser explorados fácilmente pero producen, en promedio, soluciones de baja calidad. Por otro lado, para valores grandes de k se puede esperar mejores soluciones pero el tiempo de $O(n^k)$ necesario para verificar la optimalidad local puede contrarrestar ese beneficio. Debido a esto, la investigación se ha concentrado en el desarrollo y construcción de entornos más elaborados que puedan ser explorados sin un esfuerzo computacional excesivo. A continuación discutiremos algunos pormenores en ese sentido.

Uno de los mayores logros es la clase de algoritmos *variable depth search (búsqueda de profundidad variable)*. Estos algoritmos fueron introducidos por Kernighan y Lin (1970) para el uniform graph partitioning problem y luego para el traveling salesman problem (Kernighan y Lin, 1973). Sin embargo muchos otros problemas han sido subsecuentemente tratados con el mismo tipo de procedimiento. Aquí, discutiremos el uniform graph partitioning problem que ilustra más claramente las características típicas de este método.

Recordemos su definición. Para una partición uniforme (V_1,V_2) de V, la ganancia g(a,b) que se obtiene intercambiando los nodos $a\in V_1$ y $b\in V_2$ está dada por

$$g(a,b) = \sum_{\substack{aj\in E \\ j\in V_2-\{b\}}} w_{aj} - \sum_{\substack{ai\in E \\ i\in V_1}} w_{ai} + \sum_{\substack{bi\in E \\ i\in V_1-\{a\}}} w_{bi} - \sum_{\substack{bj\in E \\ j\in V_2}} w_{bj}$$

Notemos que puede ser positiva, negativa o cero.

El algoritmo variable depth search reemplaza intercambios dobles por una secuencia seleccionada de k-intercambios dobles usando la ganancia de un intercambio para guiar la búsqueda. Más aún, el valor de k puede variar de iteración en iteración. Para una partición dada (V_1,V_2) de V con $\#V_1=\#V_2=n$ el algoritmo define una solución vecina siguiendo los siguientes pasos:

i) Elige dos nodos $a\in V_1$ y $b\in V_2$ tales que g(a,b) es máxima, aún si ese máximo es no positivo.

ii) Realiza un intercambio tentativo de a y b.

iii) Repite los pasos i) y ii) n veces, donde un nodo no puede ser elegido para ser intercambiado si ya lo ha sido en una iteración previa de i) y ii). La secuencia de intercambios dobles define una sucesión $g_1,\ldots, g_n$ de ganancias donde g_i corresponde al intercambio de los nodos $a_i\in V_1$ y $b_i\in V_2$ en la i-ésima iteración. La ganancia total luego de un k-intercambio equivale a

$$G(k) = \sum_{i=1}^{k} g_i$$

con $0\le k\le n$. Notemos que para k=n, V_1 y V_2 han sido enteramente intercambiados, resultando una ganancia total G(n)=0.

iv) Escoger un valor de k para el cual G(k) es máximo. Si G(k)>0 la solución vecina se reduce a la partición que se obtiene de (V_1,V_2) efectuando un intercambio determinado de $\{a_1,\ldots, a_k\}$ y $\{b_1,\ldots, b_k\}$. Si $G(k)\le 0$ entonces (V_1,V_2) tiene entorno vacío.

En la función de entorno anterior cada solución tiene como máximo una solución vecina. La solución final, obtenida por la repetición del procedimiento hasta ya no encontrar un conjunto de k-intercambios favorable, es óptimo local con respecto a la función de entorno aplicado.

Nuestro ambicioso procedimiento de encontrar k-intercambios para algún $k\le n$ evita el crecimiento exponencial del tiempo de aplicación que una búsqueda completa requeriría. Es más permisivo que otros algoritmos, en lo que respecta a la admisión de k'-intercambios interinos (k'<k) que no son favorables en lo inmediato. La idea es que unos pocos pequeños pasos en la dirección equivocada pueden ser redimidos en última instancia por grandes pasos en la dirección correcta.

Esta estrategia puede ser aplicada a otros problemas. En particular ha sido utilizada para el traveling salesman problem (Kernighan y Lin, 1973). Dos componentes cruciales específicos del problema en este enfoque son una apropiada función de ganancia y una regla de cierre. La regla de cierre asegura que sólo una secuencia de longitud polinomial de subintercambios puede ser realizada antes que ningún subintercambio sea posible. Por desgracia, el diseño de estos dos componentes con propósito general está revestido de asperezas.

Otra estratagema para modificar el paradigma de intercambios simples es reducir el entorno de tal manera que se incremente su eficiencia sin afectar la calidad del resultado. Un ejemplo de tal modificación para el problema de uniform graph partitioning consigna la división de un 2-intercambio en dos 1-intercambio constituidos en seleccionar el mejor nodo para trasladar de V_1 a V_2 y luego elegir el mejor nodo para trasladar de V_1 a V_2, excluyendo el nodo recientemente cambiado (Fiduccia y Matheyses, 1982). Para el traveling salesman problem puede reducirse el entorno de un 2-intercambio cuadrático a uno de categoría lineal mediante la astucia de requerir que uno de los nuevos arcos ligue una ciudad a una de sus vecinos más cercanos (Lin y Kernighan, 1973; Bonomi y Lotton, 1984: Bentley, 1990; Reinelt, 1992; Fredman, 1995).

1.4 Análisis y complejidad

A menos que el algoritmo de búsqueda local emplee una función entorno exacta, generalmente no es posible dar cotas no triviales del valor en que el costo de un óptimo local se aparta del costo óptimo. Aún más, en general no pueden ser halladas cotas no triviales para la complejidad temporal de la búsqueda local. Sin embargo, en la práctica muchos ejemplos de algoritmos de búsqueda local convergen rápidamente y encuentran soluciones de alta calidad. En esta sección discutiremos la performance de la búsqueda local en detalle, concentrándonos en aspectos teóricos y prácticos.

En el análisis de performance de un algoritmo combinatorio uno debe distinguir usualmente entre *average case (caso promedio)* y *worst case (peor caso)*. Es común efectuar un análisis de desempeño desde un punto de vista teórico-empírico, con el peor caso empírico tomado para estudiar la robustez de ese método en la práctica. La performance de un algoritmo aproximador puede ser cuantificada por el tiempo que demora su ejecución y por la calidad de las soluciones suministradas. El tiempo de ejecución es usualmente dado en segundos de CPU que el algoritmo requiere para encontrar una solución final en un ordenador y sistema operativo específico. La calidad de la solución es típicamente medida por la razón entre su valor de costo y el de una solución óptima o alguna cota fácilmente calculable del valor óptimo.

1.4.1 Resultados empíricos

A lo largo de los años muchos resultados han sido publicados sobre la performance práctica de la búsqueda local. Una conclusión general es que la búsqueda local es capaz de encontrar buenas soluciones para muchos problemas de interés en tiempos de ejecución viables.

Para el uniform graph partitioning, Kernighan y Lin (1970) observaron que el variable depth search encuentra soluciones cercanas al óptimo salvo un pequeño error relativo mientras reduce la tasa de crecimiento del tiempo de ejecución promedio a $O(n^{2,4})$. La implementación de Fiduccia y Mattheyses redujo el tiempo de ejecución a O(n log n) sin una pérdida substancial de calidad en la solución final. Ulteriores investigaciones de Johnson (1989) ahondan en la performance de la búsqueda local para el uniform graph partitioning.

Para el job shop scheduling, Van Laarhoven, Aarts y Lenstra (1992) obtienen que el algoritmo de intercambios simples encuentra soluciones finales para casos de más de 15 tareas y 10 máquinas que distan del óptimo en un 15% de éste con elevada frecuencia, lo cual es substancialmente mejor que la clásica construcción heurística basada en el dispatching rules algorithm. El tiempo de ejecución para los algoritmos de intercambio es considerablemente más largo que para los algoritmos constructivos, Applegate y Cook (1991) fueron capaces de mejorar los resultados de Van Laarhoven, Aarts y Lenstra (1992) tanto en términos de calidad de solución como en tiempo de ejecución implementando una combinación del shifting bottleneck procedure de Adams, Balas y Zawack (1988), limited back tracking y uso de entornos basados en revisión del schedule sobre múltiples máquinas. Otros autores mejoraron esos resultados con otros algoritmos de búsqueda local basados en simulated annealing, algoritmos genéticos y otros métodos.

Actualmente, los mejores algoritmos de aproximación para el job shop scheduling son el *tabu search algorithm* de Nowicki y Smutnicki (1996) que se vale de entornos de intercambios simples, la estrategia de back tracking y la clasificación de intercambios en categorías de promisorios o vedados y el *guided local search algorithm* de Balas y Vazacopoulos (1994) que combina el *shifting bottleneck procedure* con un variable-depth search basado en amplios entornos de intercambios. Este algoritmo puede encontrar soluciones con errores de 0,5% del óptimo en pocos minutos de ejecución para casos de más de 100 tareas y 20 máquinas.

El traveling salesman problem es tal vez el problema mejor analizado bajo la órbita de la búsqueda local y hasta el momento uno de sus más grandes éxitos. Reiter y Sherman (1965) fueron los primeros en efectuar un extenso estudio concerniente a algoritmos de intercambio simple. Hallaron soluciones óptimas para casos de más de 57 ciudades en menos de 20 segundos de ejecución. Desde aquel momento diversas conclusiones han sido publicadas en respuesta al desafío impuesto por manejar problemas que involucren un gran incremento del número de instancias. En resumen, tenemos que algoritmos basados en entornos de 2-intercambios y 3-intercambios hallan soluciones dentro del 3-6% del

óptimo, mientras que el variable depth search de Kernighan y Lin (1973) puede obtenerlas en un rango de 2%.

Aún más, las modernas implementaciones de estos algoritmos han explotado las ventajas que brindan las estructuras de datos y las técnicas de poda de entornos (neighborhood pruning) para obtener tiempos de ejecución sorprendentemente bajos. Tanto es así, como para permitir al algoritmo más lento, Kernighan y Lin, tomar menos de 1 hora para manejar 1 000 000 de ciudades. Actualmente, el método aproximante de mejor desempeño para el traveling salesman es el iterated Lin-Kernighan procedure de Johnson y colaboradores (Johnson, 1990; Fredman, 1995) que puede encontrar soluciones a 0,5% del óptimo en exiguos minutos de tiempo de corrido para instancias generadas aleatoriamente con más de 1 000 000 de ciudades. Resultados similares han sido manifestados por Verhoven, Swinkels y Aarts (1995) para instancias de la vida real con más de 100 000 ciudades.

1.4.2 Resultados teóricos

En la sección previa, hemos argüido que la búsqueda local promedio se comporta suficientemente bien en la práctica. Soluciones de alta calidad son generalmente encontradas en tiempos de ejecución de bajo orden polinomial. Ahora discutiremos un pequeño número de resultados teóricos, exponiendo las limitaciones de la búsqueda local analizando cómo se desenvuelve a través de un caso patológico (worst-case annalysis).

Se sabe desde algún tiempo que existen problemas para los que la búsqueda local requiere de un número exponencial de pasos para converger a la solución final. Probablemente el ejemplo más conocido que ilustra tal situación es el de Klee-Minty (1972) para el algoritmo simplex. Sin embargo, el algoritmo simplex usa entornos exactos y cabría esperar que encontrar una solución óptima local es más fácil en aquellas situaciones donde el óptimo local no necesariamente es global y en consecuencia hay muchos de estos óptimos locales presentes. Aún más, debería ser posible encontrar soluciones locales óptimas en tiempo polinomial.

Weiner, Savage y Bagchi (1976) probaron que entornos exactos para el traveling salesman problem deben ser de extensión exponencial siempre que no dependa de las instancias. Este resultado no se riñe con la posibilidad de encontrar en tiempo polinomial una solución próspera dentro de un entorno exacto de extensión exponencial. Sin embargo, Papadimitriou y Steiglitz (1977) verificaron que entornos exactos examinables polinomialmente, según el sentido anterior, no pueden existir sin valer P=NP. También construyeron ejemplos de instancias con una única solución óptima y un número exponencial de segundas mejores soluciones que son óptimos locales con respecto a entornos de k-intercambios (para $k<3/8n$) mientras que sus costos son arbitrariamente lejanos del óptimo (Papadimitriou y Steiglitz, 1978).

Más aún, incluso el entorno inexacto más familiar sucumbe frente a un caso patológico. Lueker (1975) construyó instancias y soluciones iniciales del traveling salesman problem para las cuales se revela que algoritmos de 2-intercambios requieren un número exponencial de pasos. Ejemplos de comportamiento indeseado sobre circunstancias particularmente hoscas se han manifestado en otros problemas como el independent set problem (Rödl y Tovey, 1987).

Kern (1989) discurrió a cerca de los aspectos teóricos y probabilísticos del comportamiento de la búsqueda local y advirtió que el algoritmo de 2-intercambio para instancias euclídeas del traveling salesman problem supeditado a n puntos insertos en el cuadrado unidad converge con alta probabilidad conforme a $O(n^{18})$ pasos. Estos guarismos han sido posteriormente superados por Chandra, Karloff y Tovey (1997).

Modelos teóricos alternativos para búsqueda local han sido estudiados por Tovey (1985, 1986). El consideró clases de grafos de entorno coordinados por una estructura regular, como por ejemplo el hipercubo. Para variados tipos de funciones de costo aleatorias verificó tiempos de ejecución de bajo orden polinomial para casos promedio. Por otro lado, casos patológicos eran indubitablemente sombríos.

Johnson, Papadimitriou y Yannakakis (1988) formalizaron la cuestión relativa al comportamiento de algoritmos de búsqueda local en casos anómalos introduciendo la noción de complejidad clase PLS de los problemas cuya resolución a través de la búsqueda local exige tiempo polinomial. Esta clase y su marco teórico subyacente se constituyeron en una invaluable herramienta del análisis de desempeño de la búsqueda local. Informalmente hablando, la clase PLS contiene aquellos

problemas de búsqueda local para los cuales la optimalidad local puede ser verificada en tiempo polinomial. Aún más, así como los problemas NP-completos se definen como los problemas de resolución más duros, los problemas PLS-completos involucran los problemas de búsqueda local más ríspidos. Ninguno de esos problemas se sabe soluble en tiempo polinomial y si meramente uno sólo de ellos lo fuera, lo serían todos los demás en la clase PLS.

Demostrar que un problema es PLS-completo se traduce en i) probar que es PLS y ii) existe un problema que a saber incontrovertible pertenezca a la clase PLS-completo y que pueda ser PLS-reducido a él. Expliquemos esto último, una PLS-reducción de un problema A a un problema B consiste en dos funciones coordinadas computables en tiempo polinomial. La primera función f aplica instancias X de A en instancias f(X) de B. La segunda función g aplica un óptimo local de f(X) de B al óptimo local para X en A. En términos generales, una PLS-reducción no sólo mapea instancias de A sobre aquellas de B tales que una solución de A corresponde uno a uno con soluciones de B, del mismo modo en que se desempeñan las NP-reducciones, sino que también mapea por entero la arquitectura del grafo de entorno de tal manera que la topología de ambas estructuras con respecto a la optimalidad local se conservan.

El problema PLS-completo genérico es Flip, definido como a continuación. Queda determinado un circuito booleano de compuertas AND, OR y NOR con m inputs ordenados y n outputs ordenados, donde el costo de una solución (esto es, un input) es el número binario codificado por su correspondiente output. El problema es encontrar un input cuyo costo no pueda ser decrementado mediante un cambio en el valor de una única variable.

Desde su aparición, la clase PLS ha recibido considerable atención. Un gran número de problemas de búsqueda local han sido proclamados PLS-completos, incluyendo el uniform graph partitioning con la función de entorno de profundidad variable de Kernighan y Lin (Johnson, Papadimitriou y Yannakakis, 1988), satisfiability y 2-Sat con el análogo del anterior entorno Flip (Schäffer y Yannakakis, 1991) y una variante de la función de entorno de Lin Kernighan para el TSP (papadimitriou, 1992). También cabe mencionar importantes contribuciones hechas por Krentel (1990).

1.5 Estrategias avanzadas de búsqueda

Muchos de los algoritmos de búsqueda local discutidos antes tienen la ventaja de ser flexibles o aplicables con gran generalidad, esto indica que sólo requieren de una serie de pautas: especificaciones de solución, una función de costo, una función entorno y un método eficiente para explorar entornos. Todas ellas pueden procurarse sencillamente a partir de los problemas en la gran mayoría de las circunstancias.

No obstante, puede darse con óptimos locales en muchos casos. Para remediar este inconveniente dentro del paradigma de búsqueda de entornos, se ha estado investigando las posibilidades de extender el alcance de la búsqueda local para obtener algoritmos de búsqueda por entornos que sean capaces de arribar a soluciones de alta calidad en tiempos convenientemente bajos.

Un procedimiento directo para extender la búsqueda local podría ser el desplegar un algoritmo de búsqueda local un número de veces usando diferentes soluciones iniciales y quedarse con la mejor solución encontrada como la solución final. Muchos autores han investigado esta estrategia *multi-start (multi-arranque)* pero no se han reportado éxitos relevantes. Ejemplo de ello son los trabajos de Aarts y Laarhoven (1985) y Johnson (1990) para el traveling salesman problem y Aarts (1994) para el job shop scheduling problem.

En vez de reiniciar desde una solución arbitraria, uno podría considerar una estrategia que aplique múltiples ejecuciones de la búsqueda local combinando varios entornos, esto es, recomenzando la búsqueda en un entorno con una solución seleccionada de otro entorno. Tales métodos son a menudo llamados *multi-level (multi-nivel)*. Un ejemplo de esto es la *búsqueda local iterada (iterated local search)*, en la cual las soluciones iniciales de búsquedas subsecuentes son aprehendidas modificando el óptimo local de una corrida anterior. Un ejemplo para el traveling salesman problem son el large step Markov chains (Martin, Otto y Felten, 1922) y el iterated Lin-Kernighan (Johnson, 1990) inspirado en el anterior.

Este último consiste en múltiples corridas del variable-depth search algorithm Lin-Kernigan con un 4-intercambio aleatorio con el fin de apropiarse de una nueva solución. Es actualmente el mejor algoritmo de búsqueda local para el traveling salesman problem.

Además, hay muchos métodos que toman ventaja de estrategias en las que se aceptan soluciones que deterioran el costo. Mencionamos anteriormente simulated annealing, tabu search, genetic algorithm y neural networks. Ahora, daremos una somera descripción de cada uno de ellos.

Simulated annealing

El simulated annealing (Kirkpatrick, Gelatt y Vecchi, 1983; Cerny, 1985) se basa en una analogía con el proceso físico de templado, en el cual se forma una estructura reticular en un sólido calentándolo en un baño caliente hasta ablandarlo y luego enfriándolo pausadamente hasta solidificarse en un estado de baja energía.

Desde el punto de vista de la optimización combinatoria, el simulated annealing es un algoritmo aleatorio de búsqueda local. Las soluciones vecinas de mejor costo son siempre aceptadas e incluso se aceptan las soluciones de peor costo, aunque con una probabilidad que decrece gradualmente durante el transcurso de la ejecución del algoritmo. Esta reducción de la tolerancia o probabilidad de aceptación es controlada por un conjunto de parámetros cuyos valores son determinados por un esquema de enfriamiento prescrito.

El simulated annealing ha sido vastamente aplicado con considerable éxito. Su naturaleza aleatoria permite convergencia asintótica a la solución óptima bajo condiciones moderadas. Desafortunadamente, la convergencia requiere típicamente de tiempo exponencial, convirtiendo el simulate annealing en impráctico como instrumento para procurarse soluciones óptimas. En cambio, como muchos algoritmos de búsqueda local, resulta eficiente como método de aproximación donde presenta una tasa de convergencia más indulgente (Aarts y Korst, 1989).

Taboo search

La búsqueda tabú (Glover, 1990) combina un algoritmo determinístico de avance iterativo con la posibilidad de aceptar soluciones que incrementen el costo. De esta manera, la búsqueda es dirigida fuera de óptimos locales y otras partes del espacio de soluciones puede ser explorado. Cuando es visitada una nueva solución, es considerada un vecino legítimo de la solución actual aún cuando empeore el costo. Notemos la similitud con el paradigma de elección del variable depth search algorithm discutido anteriormente.

El conjunto de vecinos legítimos queda representado por una lista tabú, cuyo objetivo es restringir la elección de soluciones para prevenir el regreso a puntos recientemente visitados. La lista tabú es actualizada dinámicamente durante la ejecución del algoritmo y define soluciones que no son aceptables en unas pocas siguientes iteraciones. Sin embargo, una solución de la lista tabú puede ser aceptada si su calidad, en algún sentido, elevada lo justifica. En este caso diremos que se ha alcanzado cierto nivel de aspiración (aspiration level).

El tabú search ha sido aplicado en una gran variedad de problemas con considerable éxito ya que presenta un esquema de gran adaptabilidad que se ajusta a los detalles del problema considerado. Por otro lado, existen vagos conocimientos teóricos que guíen ese proceso de ajuste y cada usuario debe recurrir a la información práctica disponible y a su propia experiencia.

Genetic algorithm

Esta estrategia de búsqueda se basa sobre un tema biológico y fue introducido por Holland (1975) y posteriormente Goldberg (1989). Holland utilizó conceptos de población, genética y teoría de evolución para construir algoritmos que traten de optimizar la aptitud de una población de elementos a través de recombinaciones y mutaciones de sus genes.

Existen actualmente en la literatura muchas variaciones conocidas de algoritmos que siguen esa impronta. Como un ejemplo, discutiremos una búsqueda local genética desarrollada por Mühlenbein, Gorges, Schleuter y Krämer (1988). La idea general está mejor explicada un el siguiente esquema:

i) *Iniciar*. Construir una población inicial de n soluciones.
ii) *Mejorar*. Use búsqueda local para reemplazar las n soluciones en la población por n óptimos locales.
iii) *Recombinar*. Aumentar la población añadiendo m soluciones que constituyen la progenie de las n anteriores.
iv) *Mejorar*. Usar búsqueda local para reemplazar los m retoños por m óptimos locales.
v) *Seleccionar*. Reducir la población a su amplitud original seleccionando n soluciones de la población actual.
vi) *Evolucionar*. Repetir los pasos iii), iv) y v) hasta que se satisfaga algún criterio estipulado.

Evidentemente, el paso de recombinación es un punto trascendente, puesto que se trata de tomar ventaja del hecho de disponer de varios óptimos locales.

El esquema anterior ha sido aplicado a un piélago de problemas donde han aflorado buenos resultados. Los algoritmos genéticos conforman una extensa clase de la cual emergen muchos enfoques muy disímiles entre sí.

Neural networks

Los avances en el diseño y producción de circuitos integrados han traído aparejados la computación en paralelo y el consiguiente aumento del interés en modelos computacionales que permitan la explotación de los recursos que brindan las redes. Los modelos de enlace (connectionist models) (Feldman y Ballard, 1982) son modelos inspirados en una analogía con las redes neuronales del cerebro humano, donde el paralelismo es considerado esencial.

Una red neuronal consiste en un conjunto de nodos elementales o neuronas que están enlazados a través de conexiones ponderadas. Los nodos representan unidades operativas capaces de desempeñar un cómputo simple que consiste en sumar entradas ponderadas seguidas de la adición de una constante llamada umbral (threshold) o sesgo (bias) y la aplicación de una función de respuesta no lineal. El resultado del cómputo de unidades constituye la salida. Esta es usada como entrada para los nodos que están enlazados una conexión saliente de la primera unidad. La tarea principal de la red es alcanzar una cierta configuración, por ejemplo, una requerida relación entrada-salida, a través del cómputo colectivo de los nodos. Este proceso es a menudo llamado auto-organización (self-organization).

A lo largo de los años, una extensa variedad de redes neuronales han sido exhibidas, que difieren tanto en la arquitectura o topología de la red como en el modelo computacional usado para la auto-organización. Postreramente, muchos investigadores han indagado sobre el uso de redes neuronales para resolver problemas de optimización combinatoria. La mayoría de los trabajos, motivados por el aumento potencial de la velocidad proporcionado por el cómputo masivo en paralelo. Los iniciadores en esta área son Hopfield y Tank (1985) y Baum (1986).

En muchos modelos de redes neuronales la auto-organización de la red tiende a encontrar una configuración estable de la misma. Para muchos modelos como *Hopfield network* (Hopfield, 1982) o *Boltzmann machine* (Hinton y Sejnowski, 1986) la auto-organización es susceptible de modelizarse en términos de búsqueda local.

Aquí, el procedimiento de escape del óptimo local en nuevamente de principal importancia. En un respetable número de casos se resuelve aplicando quehaceres probabilísticos, como en el simulated annealing.

En las tres últimas décadas la búsqueda local ha crecido desde una simple idea heurística hasta devenir en una potente herramienta de la investigación operativa y constituirse en campo floreciente de la optimización combinatoria. Proporciona la base sobre la que se desarrollan nuevos algoritmos, incluyendo muchos de los más populares y anteriormente citados.

Capítulo II
Job shop scheduling

2.1 Introducción

Con el término *programación de tareas (scheduling)* se entiende una vasta clase de problemas de optimización combinatoria, muy distintos entre sí por complejidad y estructura. Todos ellos se presentan como formas tangibles en la optimización de procesos de producción industrial. Muchos autores han intentado una sistematización del cuerpo de problemas y algoritmos que giran en torno a ellos. Todavía, a diferencia de cuanto acaece en otras áreas de optimización combinatoria, para los problemas de programación más difíciles no es posible indicar un enfoque netamente preferible por sobre los demás.

A continuación describiremos el job shop scheduling y los elementos para su tratamiento posterior. Luego presentaremos múltiples opciones y variantes de scheduling.

2.2 Descripción del problema

El problema del job shop scheduling se describe como sigue. Disponemos de un conjunto de *máquinas (machines)* que realizan diversas tareas y queremos elaborar cierto producto a partir de un insumo. Para esto, seguimos una serie de pasos, donde cada paso consiste en aplicar una máquina determinada durante un período de tiempo. Llamaremos *operación (operation)* a cada uno de esos pasos y *trabajo (job)* a esa secuencia de operaciones necesarias para terminar el producto. Si queremos obtener varios productos distintos, tendremos asociados un trabajo con sus correspondientes operaciones para cada uno de esos productos.

Dados un conjunto de máquinas y un conjunto de trabajos, un *programa (schedule)* es una asignación que fija a cada operación un intervalo de tiempo para ser efectuada. El problema consiste en encontrar un tal programa que minimice el tiempo necesario para completar todas las operaciones denominado *makespan*.

EJEMPLO: Supongamos que tenemos tres máquinas con números 1, 2 y 3 y que elaborar un producto consiste en aplicar las tres máquinas en el siguiente orden: primero la número 1, luego la 2, luego la 3 y por último la 1. Entonces tenemos 4 pasos que cumplir, es decir, 4 operaciones. La primera operación consiste el aplicar la máquina 1, la segunda en aplicar la máquina 2, la tercera en aplicar la máquina 3 y la cuarta en aplicar nuevamente la máquina número 1. Estas operaciones tomadas juntas conforman un trabajo, esto es, las tareas necesarias para lograr el producto.

Supongamos ahora que deseamos elaborar un segundo producto, lo cual se lleva cabo aplicando las máquinas 3, 2 y 1 en ese orden. Entonces tenemos un segundo trabajo determinado por tres operaciones, continuando la numeración anterior, la quinta, la sexta y la séptima, las cuales resultan de aplicar las máquinas 3, 2 y 1, respectivamente.

Continuando, cada operación demora un tiempo fijo en llevarse a cabo. Supongamos que todas las operaciones correspondientes al trabajo 1 tardan 10 min y las del trabajo 2 demoran 20 min. Entonces nos queda conformada una tabla como la siguiente.

operación	1	2	3	4	5	6	7
trabajo	1	1	1	1	2	2	2
máquina	1	2	3	1	3	2	1
tiempo	10’	10’	10’	10’	20’	20’	20’

TABLA: Esquema de datos de un problema de job shop scheduling.

Ya hemos esbozado el problema de job shop y nos hemos introducido en su nomenclatura, ahora vamos a dar una definición formal del problema.

Definición: (Job shop scheduling problem) Sean dados M un conjunto de máquinas, J un conjunto de trabajos y O un conjunto de operaciones. Para cada operación $i \in O$ tenemos ligadas un trabajo $j_i \in O$ al que pertenece y una máquina $m_i \in M$ en la que debe realizarse consumiendo un tiempo $p_i \in \mathbb{R}$ ininterrumpido y positivo. Además, es dada una relación de precedencia binaria $\prec$ que descompone O en cadenas, una para cada trabajo. Encontrar un tiempo de comienzo s_i para cada operación $i \in O$ tal que se minimiza el makespan, definido como $\max_{i \in O} (s_i + p_i)$, sujeto a las siguientes restricciones

i)	$s_i \geq 0$	para todo $i \in O$
ii)	$s_j \geq s_i + p_i$	para todo $i,j \in O$ con $i \prec j$
iii)	$s_j \geq s_i + p_i$ o $s_i \geq s_j + p_j$	para todo $i,j \in O$ con $m_i = m_j$

Hagamos una serie de observaciones. La restricción i) implica que ninguna máquina está disponible antes del instante 0.

La restricción ii) da cuenta de la relación de precedencia $\prec$. Esta relación refleja el hecho de que un trabajo se compone de operaciones que tienen que efectuarse en un orden preciso y no en otro. Esto resulta claro si imaginamos un trabajo que consista en forjar, horadar y empaquetar una pieza. Vale decir, entonces, que $\prec$ es una relación de precedencia fijada por el problema, entre operaciones que pertenezcan al mismo trabajo, tal que, si $i \prec j$ entonces j no puede comenzar antes que i finalice.

Finalmente la restricción iii) implica que ninguna máquina puede procesar dos operaciones al mismo tiempo. Ahora, como suponemos que una máquina no puede realizar más de una operación a la vez, es preciso fijar una secuencia para procesar las operaciones correspondientes a una misma máquina. Es decir, el programa debe imponer un orden $\prec_\pi$ entre las operaciones que se efectúen en la misma máquina, quedando determinadas una secuencia de operaciones en cada una de dichas máquinas.

Subrayamos a continuación los elementos esenciales para comprender el problema de job shop.

- *máquina (machine)*. Es cada uno de los entes que realizan alguna tarea. Disponemos de un grupo de ellas. Suponemos que una máquina cualquiera no puede realizar más de una única operación a la vez.

- *operación (operation)*. Es cada uno de los pasos necesarios para lograr un producto terminado. Una operación consiste en aplicar una máquina determinada durante un período determinado de tiempo ininterrumpido.

- *trabajo (job)*. Establece el objetivo de lograr un producto en particular. Y por comprensión, indica el conjunto de operaciones para lograr dicha meta.

- *programa (schedule)*. Es una asignación que fija a cada operación un intervalo de tiempo para ser efectuada.

- *intervalo operativo (makespan)*. Dado un programa, es el período o lapso que abarca la totalidad de la producción, es decir, la realización de todas las operaciones.

- *relación de precedencia* $\prec$. Indica el orden en que deben ser efectuadas las operaciones correspondientes a un mismo trabajo. Este orden queda determinado por el problema.

- *relación de precedencia* $\prec_\pi$. Indica el orden en que deben ser efectuadas las operaciones correspondientes a una misma máquina. Este orden puede ser elegido entre algunos admisibles.

EJEMPLO: Cabe preguntarse ¿está bien definido el problema?, ¿si todas las operaciones tienen que pasar obligatoriamente por todas las máquinas, no implicaría esto un tiempo fijo e independiente de todo programa? La respuesta es que el programa determina que algunas operaciones (que corresponden a distintas máquinas) se efectúen simultáneamente y por lo tanto aprovecha mejor el tiempo disponible y permite entonces reducir el tiempo necesario para completar la producción. De aquí, deducimos que en la medida en que mejor se explote este recurso tanto mejor será el programa, en particular, el programa óptimo tiende a efectuar la mayor cantidad posible de operaciones en simultáneo, minimizando así la demora. Veamos un ejemplo.

Imaginemos un trabajo que efectúa una única operación en la máquina 1 y un segundo trabajo que efectúa una operación en la máquina 1 y luego otra en la 2; supongamos que las tres operaciones insumen el mismo tiempo equivalente a 10 min.

Si procesamos primero la operación correspondiente al primer trabajo, el trabajo 2 permanece a la espera de que se libere la máquina 1 para comenzar, por lo tanto esta política produce un consumo de 30 min. Si, en cambio, comenzamos con el segundo trabajo podemos luego realizar la segunda operación de este en simultáneo con la que constituye el primer trabajo. Vamos a representar este fenómeno en un *diagrama de Gantt (Gantt chart)*.

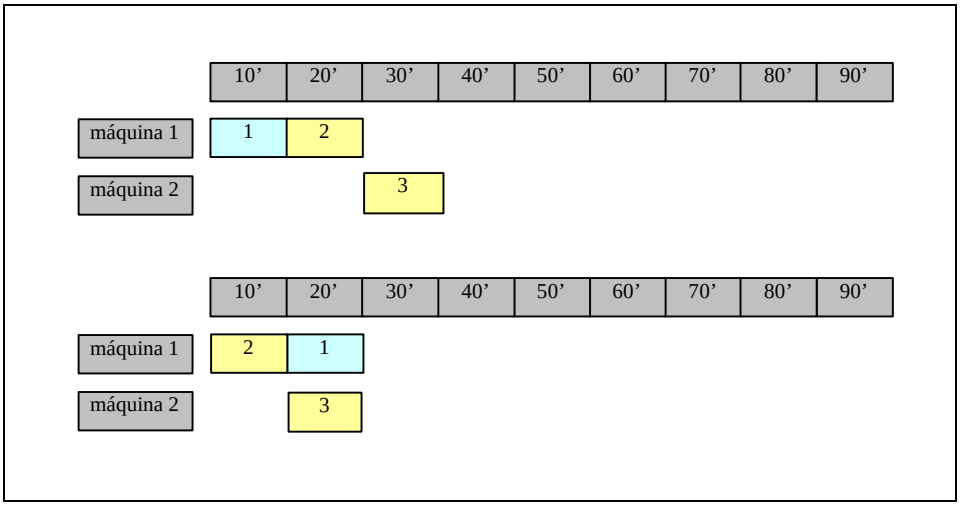


FIGURA: Se representa en un diagrama de Gantt la variación del makespan en función del orden en que se procesan las operaciones.

EJEMPLO: Vamos a esbozar cómo crece el número de instancias o soluciones factibles para comprender el porqué de la intratabilidad del job shop. Para esto, describiremos un ejemplo de flow shop, el cual es una categoría de problemas mucho más restrictivos que el job shop y por lo tanto con menor cantidad de soluciones factibles para revisar.

Supongamos que tenemos un sistema de n máquinas y n trabajos, cada trabajo consiste en aplicar todas las máquinas 1, 2, …, n en ese orden. Cada máquina entonces realiza n operaciones y definiendo el orden en que se efectuarán esas n operaciones se determina una solución factible. En efecto, es fácil ver que no puede inducirse un grafo acíclico y por lo tanto la solución obtenida resulta factible.

Finalmente, en cada máquina tenemos n! posibles secuencias y en las n máquinas tenemos $(n!)^n$ posibilidades. Esto indica que debemos buscar una solución óptima entre $(n!)^n$ soluciones factibles y eso sólo para el benévolo caso de flow shop. Para tomar conciencia del tiempo que esto implica, supongamos que pudiésemos calcular el costo de una solución arbitraria en un “único clock” de una PC de 500MHz (suposición harto generosa); revisar todo el conjunto de soluciones de un problema de 5 máquinas y 5 trabajos demoraría 50 segundos mientras que uno de 10 máquinas y 10 trabajos implicaría $2{,}54\ 10^{43}$ millones de años.

2.3 Análisis y representación

Vamos a analizar el problema utilizando un recurso muy difundido en la órbita de la optimización combinatoria, el grafo. Recordemos que un grafo es un par de la forma G=(N,R) donde N es el conjunto de *nodos* o *vértices* del grafo y $R \subseteq N \times N$ el conjunto de *arcos* o *ramas*. Además, cada rama tiene asociado un peso o costo y una dirección, por lo que el grafo se dirá *dirigido*.

En este contexto, los nodos del grafo representarán operaciones y las ramas, precedencias temporales entre operaciones, de manera que, si una operación i precede a j de acuerdo con un programa determinado, entonces el arco (i,j) debe incluirse en el conjunto de ramas. Procedamos a continuación a ampliar esta idea.

En primer lugar, introducimos dos nuevas operaciones s y t. Estas son respectivamente, una operación ficticia que se realiza antes que todas las demás y otra que se realiza luego de todas las demás. Entonces, la producción comienza con la operación s y finaliza con la operación t. Consideraremos que no pertenecen a ninguna máquina o trabajo y que tienen un tiempo de ejecución asociado nulo. De ahora en más, el conjunto O contendrá estas dos operaciones y coincidirá con el conjunto de vértices del grafo.

$$O = \{i : i \text{ operación}\} \cup \{\text{operación s}\} \cup \{\text{operación t}\}$$

En segundo lugar, advertimos un orden entre elementos de O, el orden $\prec$ prescrito por cada trabajo para procesar las operaciones que lo conforman. Entonces incorporamos una rama (i,j) al conjunto A por cada par de operaciones de un trabajo en las que i preceda a j de acuerdo al orden dado por ese trabajo. Además, como s es anterior a toda otra operación, incorporamos a A los arcos (s,i) para toda operación i. Procedemos análogamente con el nodo t. Entonces A queda determinado y permanecerá fijo de ahora en más como

$$A = \{(i,j) : i,j \text{ operaciones distintas, pertenecientes al mismo trabajo, i precede a j de acuerdo al orden dado por ese trabajo}\} \cup \{(s,i), (i,t) : i \text{ operación distinta de s y t}\}$$

y de acuerdo a la notación anterior

$$A = \{(u,v) : u,v \in O,\ u \neq v,\ j_u = j_v,\ u \prec v\} \cup \{(s,v), (v,t) : v \neq s,t\}$$

Por último, en cada máquina existe un orden para efectuar las operaciones que allí se realizan, en efecto, esto es debido a la imposibilidad de una máquina para ejecutar operaciones simultáneamente. Es decir existe un orden $\prec_\pi$ entre operaciones que pertenecen a la misma máquina. Podemos pensar en representar este orden mediante secuencias o permutaciones de operaciones. Si llamamos Π_i al conjunto de las permutaciones de las operaciones correspondientes a la máquina i-ésima, diremos que π_i es una *secuencia en la máquina i-ésima* si $\pi_i \in \Pi_i$ y diremos que π es simplemente una *secuencia* si $\pi \in \Pi_1 \times \ldots \times \Pi_m$ donde m es el número de máquinas del problema. Esta secuencia es equivalente a dar un orden como el anterior.

Naturalmente, tenemos una representación de esas restricciones en términos del grafo. Esto es, dada una secuencia π, la representamos en el grafo mediante el conjunto de arcos $S(\pi)$ denominado *selección*, donde

$$S(\pi) = \{(i,j) : i, j \text{ operaciones distintas, correspondientes a la misma máquina, i se efectúa antes que j según el orden prescrito por } \pi\}$$

y notado en consecuencia

$$S(\pi) = \{(i,j) : i,j \in O,\ i \neq j,\ m_i = m_j,\ i \prec_\pi j\}$$

Entonces, dada una secuencia π podemos representar las precedencias temporales del problema mediante un grafo que contenga todas las operaciones y todos los arcos mencionados anteriormente. Resulta de esta manera

$$G(\pi) = (O, A \cup S(\pi))$$

Hagamos algunas observaciones. Primero, la secuencia, como veremos más adelante, determina el programa y viceversa lo cual nos motiva a considerar como variable del problema a la secuencia en vez del programa. Esto tiene la ventaja de explotar el vínculo directo entre la secuencia y el grafo.

Segundo, advertimos que una secuencia puede ser elegida entre muchas posibilidades, sin embargo, no todas representan un esquema factible. Si la secuencia π induce un ciclo en $G(\pi)$ entonces los arcos no reflejan una relación de precedencia temporal. Por lo tanto, decimos que π es una solución factible del problema si $G(\pi)$ es acíclico.

Dado un programa o su solución factible π asociada, existe un lapso mínimo de tiempo en el cual se puede completar toda la producción, el cual habíamos llamado *makespan*. Consideremos el grafo $G(\pi)$ definido como antes, si tomamos como peso de cada rama (i,j) el valor p_i entonces el makespan es igual a

la longitud del camino más largo entre s y t, denominado *camino crítico (critical path)*. Hemos de subrayar este hecho, ya que, como existen numerosos algoritmos que en tiempo polinomial encuentran el camino crítico en un grafo, disponemos de un procedimiento para encontrar el makespan asociado a una secuencia factible arbitraria.

Finalmente, el problema queda reducido a encontrar una solución factible π que minimice el makespan.

2.3.1 Secuencia, orden y programa

Proposición: Una secuencia π y un orden $\prec_\pi$ son equivalentes.

Demostración: Si $(a_1,\ldots,a_k)$ es una permutación de las operaciones de la máquina i-ésima vale $a_1 \prec_\pi \ldots \prec_\pi a_k$ sii $\pi_i=(a_1, \ldots, a_k)$. Como $\pi=\pi_1\times\ldots\times\pi_m$ resulta la demostración.

2.3.2 Cálculo del makespan

En este apartado intentaremos ofrecer un método expedito para calcular el makespan asociado a una secuencia π. Para esto, demostraremos la equivalencia entre dicho valor y el camino más largo entre los vértices s y t que demarcan los límites de la producción.

Definición: Dado un grafo G cualquiera con vértices s y t. Denominamos camino entre s y t al conjunto ordenado de vértices tales que $s\to u_1\to\ldots\to u_k\to t$. Además llamamos costo o peso de ese camino al valor igual a la suma de los pesos de cada rama que lo conforma, es decir, $c_{su_1}+\ldots+c_{u_kt}$.

Teorema: Sea dado un problema de job shop scheduling, una secuencia factible π y un grafo $G = (O, A\cup S(\pi))$ asociado. Suponemos que el peso de una rama (i,j) es p_i o el tiempo necesario para completar la operación i. Entonces, el makespan es igual a la longitud del camino más largo entre s y t.

Demostración: Sean $\{y_i\}$ un conjunto de valores tales que y_i es igual al costo de un camino óptimo entre s y el vértice i y sea $\{s_i\}$ un programa que respete la secuencia π. Observamos que el costo del camino más largo entre s y t es igual a $y_t-y_s=y_t$ y el makespan es igual a $s_t+p_t=s_t$. Queremos ver que estos dos valores son iguales.

1) Veamos que debe ser $y_t\le s_t$. Sea $s\to u_1\to\ldots\to u_k\to t$ el camino más largo entre s y t, entonces su longitud está dada por $y_t=p_s+p_{u_1}+\ldots+p_{u_k}=p_{u_1}+\ldots+p_{u_k}$.

Como $s_{u_1}\ge 0$ entonces $y_t\le s_{u_1}+p_{u_1}+\ldots+p_{u_k}$. Como existe una rama entre u_1 y u_2 debe ser o bien $u_1\prec u_2$ o bien $u_1\prec_\pi u_2$ pero en ambos casos tenemos $s_{u_1}+p_{u_1}\le s_{u_2}$ y por consiguiente $y_t\le s_{u_2}+p_{u_2}+\ldots+p_{u_k}\ldots$

y así siguiendo llegamos a que $y_t\le s_t$.

2) Veamos que $s_i=y_i$ es un programa.

a) Es cierto que $0\le y_i=s_i$ para toda operación i.

b) Sean $i\prec j$ operaciones en un mismo trabajo, entonces y_i+p_i es el costo de un camino desde s hasta j pero como y_j representa el costo óptimo, debe ser $y_i+p_i\le y_j$ y resulta $s_i+p_i\le s_j$.

c) Sean i, j operaciones en una misma máquina, entonces $i\prec_\pi j$ y razonando análogamente que en el punto b) resulta $s_i+p_i\le s_j$ o bien $j\prec_\pi i$ y resulta $s_j+p_j\le s_i$.

3) De los puntos anteriores se deduce que el costo del camino más largo es el makespan asociado a al programa de mínima demora que preserva el orden π.

Ahora, vamos a definir inicialmente el algoritmo de Ford para encontrar el camino más corto entre los vértices s y t de un grafo dirigido G=(V,E).

Algoritmo de Ford
inicializar
$y_s=0$, $q_s=-1$
$y_u=\infty$, $q_u=-1$ para todo vértice $u\in V$
repetir #V−1 veces
repetir para cada rama $uv\in E$
si $y_v>y_u+c_{uv}$ hacer $y_v=y_u+c_{uv}$

Al finalizar lo anterior, los valores y_u contienen el costo de un camino óptimo de s hasta u, si fuese $y_u=\infty$ significaría que no hay un camino entre s y u. Además, q_v indica el vértice anterior a v en un camino óptimo que llegue hasta v, por lo tanto, los valores de q_v indican cómo recorrer el camino óptimo de atrás para adelante.

Teorema: (complejidad del algoritmo de Ford) Se hacen #V−1 revisiones o pasadas cada una de las cuales procesa #E ramas. Por lo tanto la complejidad es O(#V #E).

Teorema: Como resultado de la pasada k-ésima y_u es óptima si el camino óptimo a u no tiene más de k ramas. Como ningún camino simple puede tener más de #V−1 ramas, el algoritmo termina en no más de #V−1 revisiones si no hay ciclos negativos.

Demostración: Procedemos por inducción en k.
Consideremos k=1. Si $s\rightarrow v$ es óptimo, en la primera pasada tendremos $y_v=\infty > y_s+c_{sv}$. Por lo tanto se asigna $y_v=c_{sv}$.
Paso inductivo. Sea v un vértice cuyo camino óptimo tiene k+1 ramas. Sea dicho camino $s\rightarrow u_1\rightarrow \ldots \rightarrow u_k\rightarrow v$. Por la hipótesis inductiva y_{u_k} es óptimo así que el costo mínimo a v es $y_{u_k}+ c_{u_kv}$. Por otra parte, en la pasada k+1 el valor y_{u_k} no se modifica mientras que el valor revisado de y_v satisface $y_v\leq y_{u_k}+ c_{u_kv}$. Pero el segundo miembro es el costo óptimo de un camino a v. Así que $y_v=y_{u_k}+ c_{u_kv}$.

Ahora modificamos el algoritmo de Ford para hallar caminos más largos y así encontrar en particular el camino más largo entre s y t.

Algoritmo de Ford modificado
inicializar
$y_s=0$, $q_s=-1$
$y_u=0$, $q_u=-1$ para todo vértice $u\in V$
repetir #V−1 veces
repetir para cada rama $uv\in E$
si $y_v<y_u+c_{uv}$ hacer $y_v=y_u+c_{uv}$

Análogamente, el valor y_u representa el costo de un camino óptimo desde s hasta u y el valor q_u indica el vértice anterior a u en un camino óptimo.

Puede verse que la complejidad resulta igualmente O(#V #E) y la demostración es completamente análoga a la anterior.

EJEMPLO: Continuamos con el ejemplo anterior. El conjunto de vértices está formado por O = {1, 2, 3, 4, 5, 6, 7, s, t}.

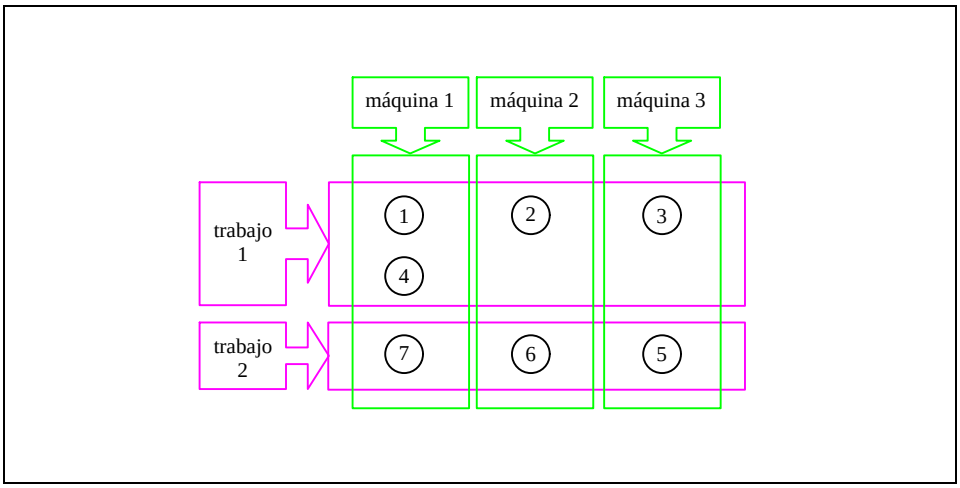


FIGURA: Esquema de representación conjunto de vértices. Se omiten s y t.

Veamos el conjunto A de ramas. El trabajo 1 está constituido por las operaciones 1, 2, 3 y 4 y deben efectuarse en ese orden y no en otro, por lo tanto el conjunto A debe contener los arcos (1,2), (1,3), (1,4), (2,3), (2,4) y (3,4).

En efecto, como la operación 1 se realiza antes que la 2, debe estar presente en A el arco (1,2), como la operación 1 se realiza antes que la 3, el conjunto debe contener el arco (1,3) … y así siguiendo.

Procediendo análogamente, el trabajo 2 aporta los arcos (5,6), (5,7) y (6,7). Si ahora agregamos los arcos de las formas (s,·) y (·,t) queda formado definitivamente el conjunto A como

$$A = \{(1,2), (1,3), (1,4), (2,3), (2,4), (3,4)\} \cup \{(5,6), (5,7), (6,7)\} \cup \{(s,i), (i,t) : \ i=1,\ldots,7\}$$

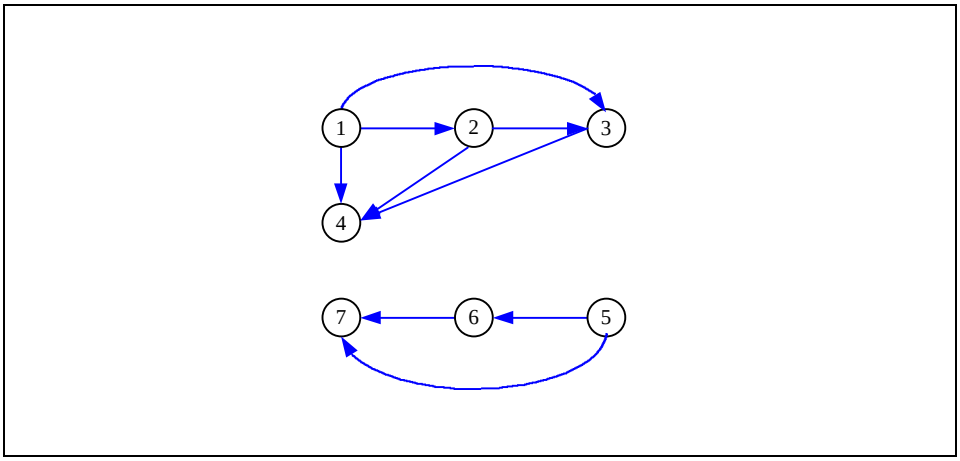


FIGURA: En flechas azules elementos del conjunto A que no involucran los vértices s y t

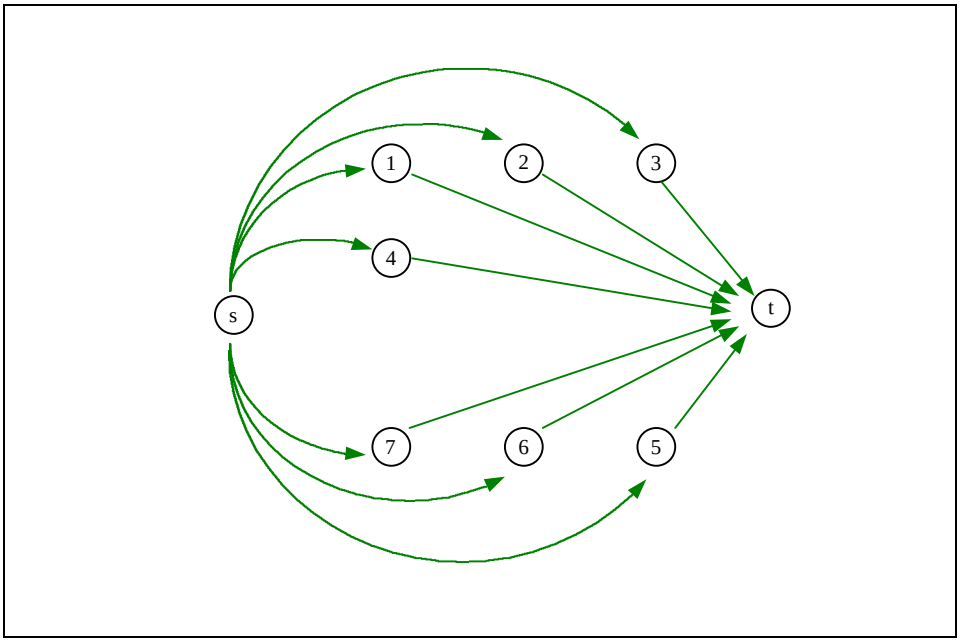


FIGURA: En flechas verdes, elementos del conjunto A que involucran los vértices s y t

Consideremos la máquina 1. Allí se realizan tres operaciones: 1, 4 y 7. Dado que una máquina no puede realizar tareas simultáneas, es preciso secuenciarlas, esto es, decidir el orden en que serán procesadas. Por ejemplo, la secuencia podría ser 1, 4, 7 y entonces el conjunto S_1 de los arcos asociados a la máquina 1 sería {(1,4), (1,7), (4,7)} o bien podría ser 1, 7, 4 y en consecuencia S_1={(1,7), (1,4), (7,4)}.

Ahora, una selección sería por ejemplo S = {(1,4), (1,7), (4,7)} ∪ {(2,6)} ∪ {(3,5)}, donde cada conjunto es una selección sobre las máquinas 1, 2 y 3, respectivamente.

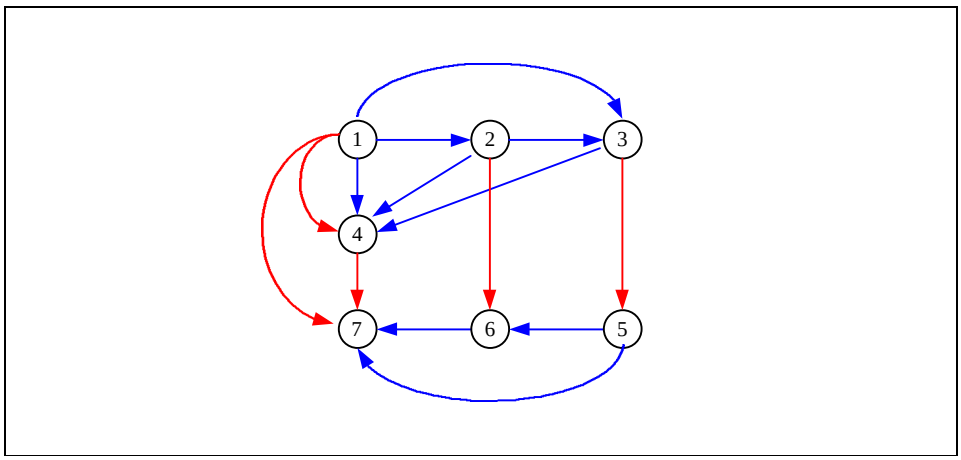


FIGURA: Selección S, en trazo azul ramas de A, en rojo ramas de S. Se omitieron s, t y sus ramas correspondientes.

Notemos que en la elección de S se pide aciclidad del grafo G. Si no lo hiciéramos, podríamos elegir el orden 6, 2 en máquina 2 y el 3, 5 en máquina 3 y habría un ciclo formado por los arcos (6,2), (3,5) de S y (2,3), (5,6) de A. Esto no debemos permitirlo, ya que conduce al absurdo de suponer que la operación 2 se hace antes que sí misma.

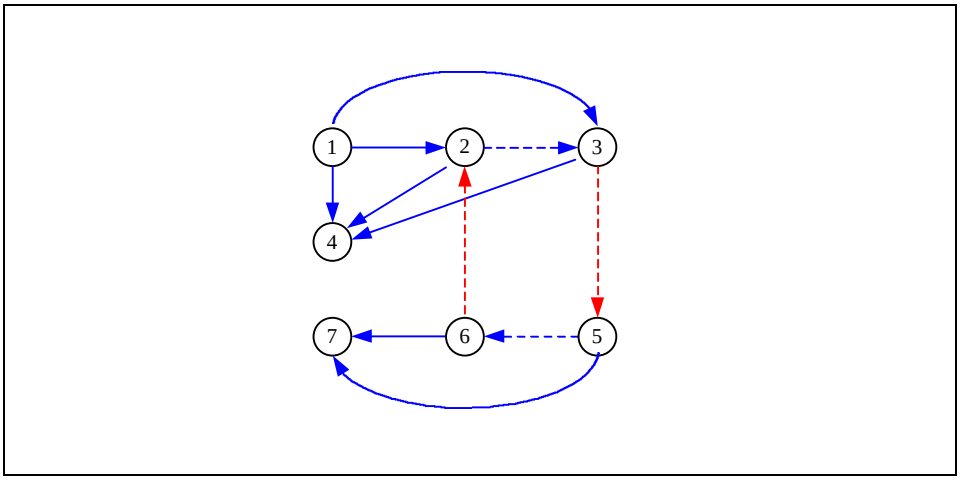


FIGURA: Error en selección, ciclo marcado con trazo punteado.

Ahora, asignando al arco (i,j) un peso igual al tiempo que demora en efectuarse la operación j, buscamos el camino más largo dentro del grafo resultante. En este caso es el camino que recorre los vértices s, 1, 2, 3, 5, 6, 7, t con un peso total de 90 min, lo cual indica que completar la producción con la secuencia dada por la selección S tardará 90 min.

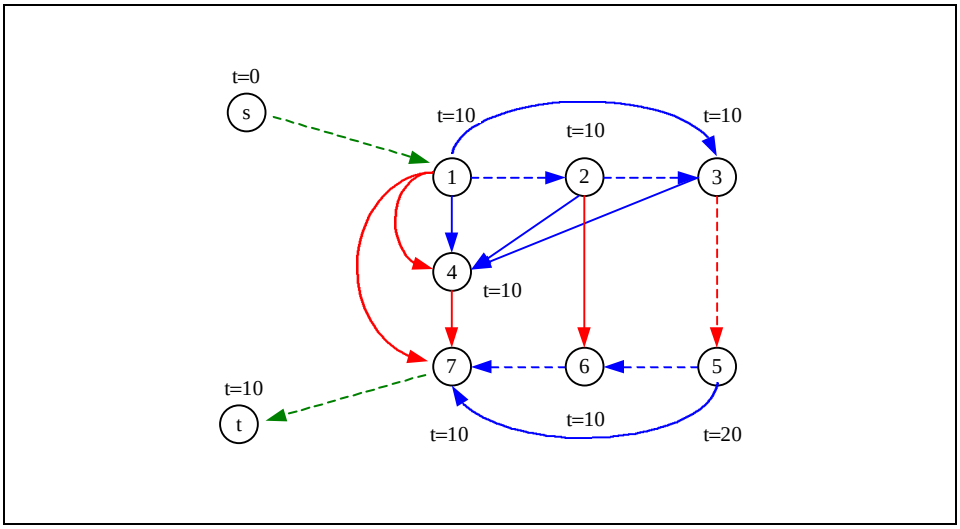


FIGURA: Se marcaron los pesos de las operaciones y se señaló el camino crítico con trazo punteado.

2.4 Elementos de problemas de Scheduling

Cuando observamos más estrechamente la aplicación concreta de técnicas de scheduling, vislumbramos la presencia de elementos particulares que se agregan al problema. Estos elementos reflejan distintas necesidades o circunstancias específicas que detallan precisamente el problema a resolver. Por un lado debemos considerar el modelo de producción utilizado y las condiciones concretas y tangibles que resultan como consecuencias prácticas de su implementación. Y por otro lado, aparecen necesidades peculiares a las que debemos ceñirnos a la hora de precisar el objetivo que queremos alcanzar. Se vislumbra entonces una gran diversidad de problemas los cuales conforman una clase muy heterogénea e imposible de tratar de manera abarcativa.

A continuación describiremos algunos de los elementos que aparecen más frecuentemente.

Los problemas de programación de tareas suelen clasificarse a grandes rasgos bajo la órbita de alguno de los siguientes tres tipos.

- *máquina única (single machine)*. Este es el caso más simple en el cual todos los trabajos requieren el mismo recurso para ser ejecutados. En general, cada trabajo consiste en una sola operación y entonces puede hablarse indistintamente de trabajo y operación.

- *flow shop*. En este caso, el sistema consiste en m máquinas dispuestas en serie y cada trabajo debe ser ejecutado por cada una de las m máquinas sucesivamente, o sea, primero la máquina 1, luego la 2, … y finalmente la m. A menudo se asume que cada máquina tiene un buffer de tipo FIFO. Por lo tanto, el orden en el cual los trabajos visitan cada máquina es siempre el mismo y en otras palabras, los trabajos no pueden pasarse. En este caso suele hablarse de *permutation flow shop*.

- *job shop*. Aunque aquí hay m máquinas, cada trabajo tiene un orden propio con el cual visitarlas. Puede notarse fácilmente que el job shop es un generalización de flow shop, o bien, el flow shop puede verse como un job shop en el cual cada trabajo debe visitar las máquinas en el orden 1, 2, …, n.

La variabilidad de los problemas de programación se manifiesta principalmente en dos niveles, en lo que respecta a las restricciones y en la función cuyo objetivo es optimizar.

En cuanto a las restricciones, en un problema de programación como los anteriores suelen aparecer diversos tipos de o especificaciones suplementarias que contribuyen a definir y complicar el problema. Estas pueden comprender.

- *fecha de disparo (release date)* r_j . Indica el instante de tiempo (a partir de un instante inicial 0) antes del cual no es posible iniciar la ejecución del trabajo j. Por ejemplo, si la materia prima necesaria para efectuar el trabajo j arriba dentro de tres días, el trabajo j no puede comenzarse hasta entonces.

- *fecha consignada (due date)* d_j. Indica el instante de tiempo (a partir de un instante inicial 0) antes del cual la ejecución del trabajo j debería estar terminada. In general, la violación de un tiempo de consignación conlleva algún costo (punitorios , pérdida de confianza por parte del cliente, etc).

- *peso (weight)* w_j. Representa la importancia relativa del trabajo j respecto de los otros.

- *tiempo de configuración (set-up)* s_{ij}. Esta característica está presente sobre todo en problemas de una única máquina e indica que se quiere seguir al trabajo i el trabajo j, entonces, entre estos dos trabajos, es necesario reconfigurar la máquina, lo cual requiere un tiempo s_{ij}.

- *preferencia (preemption)*. En ciertos casos, se consiente en interrumpir un trabajo para permitir la ejecución de uno más urgente. El problema en esos casos se llama *preemptive*.

- *vínculos de precedencia entre trabajos*. Pueden existir relaciones de precedencia entre trabajos. Por ejemplo, un vínculo de este tipo entre los trabajos i y j marca que el trabajo i no puede comenzar hasta haber finalizado el trabajo j.

- *bloqueo (blocking)*. En un problema de flow shop, los trabajos en espera para ser efectuados en una máquina, son hospedados en un buffer. Si en un instante dado el buffer de la máquina i está lleno, un trabajo terminado en la máquina i−1 no puede encontrar un puesto en el buffer de la máquina i y está por lo tanto condenada a permanecer bloqueando la máquina i−1, la cual no podrá iniciar una nueva tarea hasta tanto no se desembarace del trabajo. Este fenómeno se llama *blocking*.

- *opción sin espera (no-wait)*. Si un hubiese un buffer a la entrada de la máquina i, la máquina i−1 estaría ocupada hasta que se liberase la máquina i. Aún más restrictiva es la situación *no-wait* en la cual a los jobs no se les permite siquiera esperar dentro de una máquina y en cambio, debe

garantirse que al instante de completar una operación en una máquina, maquina sucesiva está ya disponible para continuar el trabajo.

En cuanto a objetivos, los problemas de programación pueden ser muy distintos. Para definirlos, introducimos de forma preliminar algunos conceptos.

- *tiempo de completamiento* C_j. Es el instante en el cual el último trabajo j (y por lo tanto el trabajo entero) termina. Si no admitimos interrupciones, C_j está dado por la suma del instante de inicio de la última operación del trabajo j y del tiempo de procesamiento de dicha operación.

- *tiempo de llegada (lateness o arrival)* L_j. Es la diferencia entre el tiempo de completamiento y la fecha consignada para la entrega del trabajo j. Notemos que si es positiva indica un retraso y si es negativa un adelanto. Esto es, $L_j=C_j-d_j$.

- *tiempo de demora (tardiness)* T_j. Coincide con el tiempo de llegada cuando éste es positivo, de otra manera vale 0. O sea, $T_j=\max\{L_j, 0\}$.

Ahora bien, el objetivo es por lo general, optimizar alguno de los siguientes valores.

- *máximo tiempo de completamiento (makespan)* C_{max}. Dado un programa S se define como $\max\{C_1,\ldots,C_n\}$ y representa la medida (respecto del instante 0) del tiempo necesario para concluir toda la actividad.

- *máximo tiempo de llegada* L_{max}. Dado un programa S se define como $\max\{L_1,\ldots,L_n\}$, o sea el retardo del trabajo que termina con mayor retardo respecto a su propia fecha consignada. (Notemos que puede ser negativo y en tal caso representa el anticipo del trabajo que termina con menor anticipo).

- *máximo tiempo de demora* T_{max}. Se define como $\max\{L_{max}, 0\}$.

- *suma ponderada de los tiempos de completamiento* T_{max}. Dado un programa S se define como

$$\sum_{j=1}^{n} w_j C_j$$

En el caso en el cual los pesos son todos iguales esta cantidad aparece como una medida del volumen total de tiempo que el sistema asigna a cada trabajo, indicativo del servicio que puede ofrecer ese sistema. Además, los pesos introducen la noción de prioridad.

Finalmente, puede darse el caso de función multi-objetivo, en los cuales hay presentes simultáneamente dos objetivos distintos. Incluso, pueden coexistir varios objetivos entre los cuales se debe lograr el mejor compromiso. Usualmente debemos distinguir dos enfoques diferentes para abordar estos problemas. A saber, si f_1 y f_2 son dos funciones objetivo.

- Se encausa a un problema de un único objetivo, optimizando respecto a f_1 e imponiendo como restricción una cota al valor de f_2 (o viceversa).

- Se determinan todas las soluciones *no dominadas (soluzione nodominata)*. Una solución S es no dominada si es posible encontrar una solución mejor para la primera función objetivo sólo a condición de empeorar respecto de la segunda (y viceversa).

Capítulo III
Taboo Search Algorithm

3.1 Generales

El algoritmo de Taboo Search es un procedimiento heurístico para hallar soluciones cuasi óptimas del problema de Job Shop Scheduling. Primeramente introducido por Glover (1985) presentamos la versión de Nowicki y Smutnicki (1996).

Inicialmente definimos una noción fundamental llamada *move (movimiento)*. Un move es una perturbación particular que transforma una solución en otra. Llamamos entorno de una solución al subconjunto de las soluciones que se obtienen de la primera por medio de un move.

Partiendo de una solución o secuencia inicial, el algoritmo a cada paso, busca en el entorno de la secuencia dada, una nueva secuencia para la próxima iteración. Esto resulta en un camino o recorrido por el conjunto de secuencias. Para evitar que el camino se cierre sobre sí mismo y origine un ciclo en el algoritmo, guardamos las últimas k secuencias generadas en lo que denominamos una lista taboo.

Una vez concluido un trayecto, ponemos en práctica una estrategia de diversificación, que consiste en retroceder un trecho y emprender la marcha por un nuevo camino. Esto ocasiona que los caminos se abran en racimo, irradiando el conjunto de secuencias.

Finalmente, la secuencia buscada es la mejor secuencia entre todas las examinadas anteriormente.

3.2 Inicio

Partimos de un problema de job shop scheduling con dos condiciones fundamentales:

- Las operaciones demoran tiempos positivos, esto es, estrictamente mayores que 0.

- No hay dos operaciones consecutivas de un trabajo que se efectúen en la misma máquina.

Estas condiciones, pese a su carácter imperativo, imponen leves restricciones al problema concreto, ya que podemos satisfacerlas fácilmente modificando ligeramente el problema. En efecto, si fuese necesario, en el primer punto suponemos que una operación de tiempo nulo demora un tiempo $\varepsilon>0$ despreciable y en el segundo punto intercalamos una operación ficticia que se realice en otra máquina y que tarde igualmente $\varepsilon>0$.

El problema de job shop, toma un aspecto conocido cuando lo formulamos en términos de un grafo, siguiendo el procedimiento descrito en el capítulo precedente obtenemos el conjunto de operaciones O y el conjunto de arcos A. Sólo resta encontrar una solución factible π para definir el grafo $G(\pi)=(O,A\cup S(\pi))$ conteniendo todos los elementos necesarios para comenzar nuestra tarea.

Esta solución factible inicial es proporcionada por algún algoritmo secundario como SPT, LPT, MWKR, LWKR, MOPNR, RATIO o INSA. Generalmente, un solución obtenida de este modo resulta de buena calidad, con lo cual pretendemos que se ubique, en un sentido intuitivo, más cerca de la solución óptima y por lo tanto incida favorablemente en el desempeño del Taboo Search en cuanto a i) la calidad de la solución final que encuentre y ii) el tiempo que demore en encontrar esa solución final.

Además, mencionamos un beneficio adicional suministrado por del uso de dichos algoritmos: nos aportan una idea aproximada de la dimensión y complejidad del problema antes de tratarlos con el Taboo Search. Esto permite ligeros ajustes sobre el algoritmo con el fin de adaptarlo al problema y mejorar así su performance. Más adelante, describiremos uno de esos métodos citados, conocido como INSA.

3.2.1 Obtener una solución inicial, el INSA

Describiremos un método de entre los citados conocido como INSA o *algoritmo de inserción (insertion algorithm)* destinado a encontrar una solución factible de buena calidad, el cual fue primeramente concebido para problemas de flow shop (Nawaz y alumnos, 1983) y luego mejorado y extendido a problemas de job shop (Werner y Winkler, 1992). Podemos agregar que el INSA tiene una complejidad $O(n^3m^2)$.

La idea general es comenzar con un orden conteniendo todas las operaciones del trabajo cuyo tiempo de proceso individual es mayor, luego, insertar las operaciones restantes en el orden en que no se incrementen los tiempos de procesamiento.

Notamos $\sigma=(\sigma_1,\ldots, \sigma_m)$ la secuencia parcial y σ_k la secuencia parcial en la máquina k-ésima, es decir, si m_k es la cantidad de operaciones en la máquina k-ésima, $\sigma_k=(\sigma_1(1),\ldots, \sigma_k(m_k'))$ una permutación del conjunto $\{1,\ldots, m_k'\}$con $m_k'\leq m_k$. Para cada σ definimos el grafo $G'(\sigma)=(O, A\cup S'(\sigma))$ donde

$$S'(\sigma)=\{(\sigma_k(i), \sigma_k(i+1)) : 1\leq k\leq m, 1\leq i\leq m_k-1\}$$

Supongamos que queremos insertar una operación x en σ. Sea $d(x,\sigma)$ la longitud del camino más largo en $S'(\sigma)$ que pase por la operación x, sea μ la máquina donde debe efectuarse x y sea Π_μ' el conjunto de $m_\mu'+1$ permutaciones obtenidas de σ_μ por la inserción de x en distintas posiciones, esto es

$$\Pi_\mu'=\{(\sigma_\mu(1), \ldots, \sigma_\mu(j-1), x, \sigma_\mu(j), \ldots, \sigma_\mu(m_\mu')) : 1\leq j\leq m_\mu'+1\}$$

El conjunto Π_μ' genera nuevos órdenes parciales manteniendo fijos los de las máquinas distintas a μ. pero hemos de observar que no todos esos órdenes son factibles, por lo tanto, sea Π' un subconjunto que contiene todos los órdenes factibles. Veremos luego que ese subconjunto no es vacío. Para cada orden de proceso $\sigma'\in\Pi'$ se calcula el costo $d(x,\sigma)$ y se elige el x que minimice dicha cantidad.

En vista de las consideraciones anteriores, formulamos el INSA. Suponemos por simplicidad que cada trabajo tiene exactamente una operación en cada una de las máquinas.

INSA o algoritmo de inserción

encontrar el trabajo más largo, esto es, encontrar $k\in J$ tal que maximice la suma de los p_i sobre el conjunto $\{i : i\in O, j_i=j_k\}$
hacer $m_i'=1$ para toda máquina $i\in M$ y $\sigma_{m_i}(1)=i$ para todo $i\in j_k$
encontrar una permutación ω del conjunto $\Omega=\{i : i\in O, j_i\neq j_k\}$ tal que resulte un orden no decreciente de los tiempos de procesamiento
repetir para i desde 1 hasta $\#\Omega$
    para el orden parcial σ, sea la operación $x=\omega(i)$ y la máquina $\mu=m_x$
    encontrar el conjunto Π_μ' y luego encontrar el conjunto Π'
    encontrar $\sigma'\in\Pi'$ tal que $d(x,\sigma)=\min\{d(x,\alpha) : \alpha\in\Pi'\}$
    hacer $\sigma=\sigma'$ y $m_\mu'=m_\mu'+1$

Quedó planteada con anterioridad una propiedad que pasaremos a demostrar.

Proposición: Sean un orden parcial σ, una operación x, μ la máquina donde debe efectuarse x y Π_μ' el conjunto de las permutaciones obtenidas de σ_μ mediante la inserción de x en distintas posiciones. Entonces, el conjunto Π' de las soluciones fatibles de Π_μ' es no vacío.

Demostración: Consideremos el grafo $G(\sigma)$, es decir, el grafo previo a la inserción de x. Sean $a_1,\ldots,a_n$ las operaciones ya ordenadas en la máquina μ (es decir, $a_i=\sigma_\mu(i)$ para $1\leq i\leq m_\mu'$ y $n=m_\mu'$). Consideremos los conjuntos

$V=\{i : 1\leq i\leq n$, existe un camino dirigido en $G(\sigma)$ desde a_i hasta $x\}$
$W=\{i : 1\leq i\leq n$, existe un camino dirigido en $G(\sigma)$ desde x hasta $a_i\}$

Procedemos ahora por casos.

1) Si $V=\emptyset$ entonces $(x, a_1, \ldots, a_n)\in\Pi'$. En efecto, agregando al grafo $G(\sigma)$ las ramas $x\rightarrow a_i$ $1\leq i\leq n$ resulta el grafo $G(\sigma')$. Si hubiese un ciclo en este último debería pasar por x, pero no existe ningún camino desde a_i hasta x.
2) Si $W=\emptyset$ entonces $(a_1, \ldots, a_n, x)\in\Pi'$. En efecto, agregando al grafo $G(\sigma)$ las ramas $a_i\rightarrow x$ $1\leq i\leq n$ resulta el grafo $G(\sigma')$. Si hubiese un ciclo en este último debería pasar por x, pero no existe ningún camino desde x hasta a_i.
3) Si V,W no vacíos sean entonces k_1 el máximo de V y k_2 el mínimo de W. Claramente no puede ser $k_1=k_2$ de lo contrario habría un ciclo en $G(\sigma)$. Proponemos insertar x entre a_k y a_{k+1} donde k es tal que $k_1\leq k\leq k_2$, es decir $(a_1, \ldots, a_k, x, a_{k+1}, \ldots, a_n)$. Para todo $1\leq i\leq k$, como no existe camino desde x hasta a_i, los arcos $a_i\rightarrow x$ no crean ciclo y para todo $k+1\leq i\leq n$, como no existe camino desde a_i hasta x, los arcos $x\rightarrow a_i$ no crean ciclo.

3.3 Neighborhood

Definimos un move $v=(x,y)$ para la secuencia π, como un intercambio en el orden de dos operaciones x e y correspondientes a una misma máquina, dando lugar a una nueva secuencia π'.

Para hallar moves convenientes, partimos el camino crítico de $G(\pi)$, el grafo asociado a la secuencia π, en bloques de operaciones que sean consecutivas en el camino crítico y que correspondan a la misma máquina. Las dos primeras y las dos últimas operaciones de cada bloque, serán nuestros moves. En el primer bloque sólo consideraremos las dos últimas operaciones mientras que en el último bloque sólo consideraremos las dos primeras.

Notaremos $V(\pi)$ al conjunto de moves selectos para realizar sobre la secuencia π y definimos el entorno de π como las secuencias π' que se obtienen de π haciendo un move de $V(\pi)$.

Definición: Sea $u_1, \ldots, u_n$ el camino crítico en $G(\pi)$ el cual partimos en r bloques. Un bloque B_j se define como

i) $B_j=(u_{a_j}, u_{a_j+1}, \ldots, u_{b_j})$ tal que $a_j\leq b_j$ para todo $1\leq j\leq r$. Además $a_{j+1}=b_j+1$ para todo $1\leq j\leq r-1$ y $a_1=1$, $b_r=n$.
ii) Todas las operaciones de B_j corresponden a la misma máquina.
iii) Dos bloques consecutivos tienen operaciones que se efectúan en máquinas diferentes.

El conjunto de moves se define como $V(\pi)=V_1(\pi)\cup\ldots\cup V_r(\pi)$ donde

i) $V_1(\pi)$ es igual a $\{(u_{b_1-1}, u_{b_1})\}$ si $a_1<b_1$ y $1<r$, caso contrario es vacío.
ii) $V_r(\pi)$ es igual a $\{(u_{a_r}, u_{a_r+1})\}$ si $a_r<b_r$ y $1<r$, caso contrario es vacío.
iii) Para todo $1<j<r$ vale que $V_j(\pi)$ es igual a $\{(u_{a_j}, u_{a_j+1}), (u_{b_1-1}, u_{b_1})\}$ si $a_j<b_j$, caso contrario es vacío.

EJEMPLO: Supongamos que el camino crítico está dado por las operaciones 1,…,10 en ese orden. Las tres primeras corresponden a la máquina 1, las siguientes a la máquina 2 y las tres últimas nuevamente a la máquina 1. Tenemos tres bloques: B_1={1, 2, 3}, B_2={4, 5, 6, 7} y B_3={8, 9, 10} y los moves: V={(2,3), (4,5), (6,7), (8,9)}.

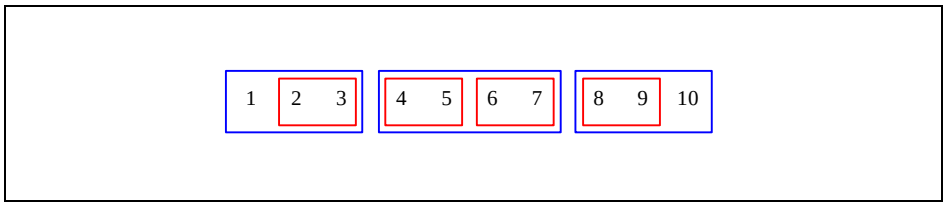


FIGURA: El camino crítico. En azul los bloques, en rojo los moves.

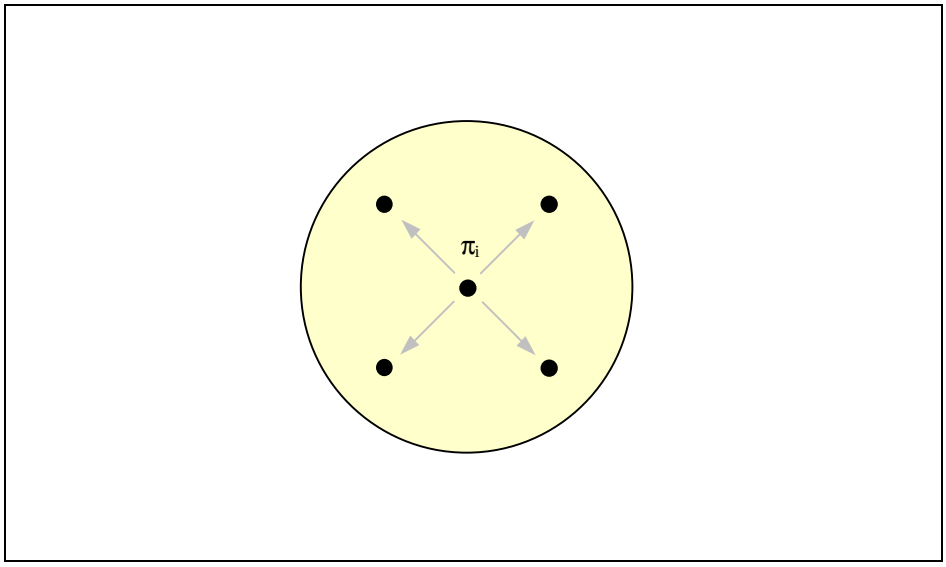


FIGURA: Una secuencia π y su entorno, el cual queda definido por el conjunto V(π).

3.3.1 Aspectos teóricos del entorno

En vista del hecho de que no todas las secuencias son factibles es lícito preguntarse si un move preserva esa propiedad, esto es, partiendo una secuencia factible ¿un move nos conduce a otra secuencia también factible? La respuesta afirmativa se plasma en el siguiente par de proposiciones.

Proposición: Si $(x,y) \in V(\pi)$ entonces las operaciones x e y se procesan seguidamente en la misma máquina, esto es, son consecutivas en π y por lo tanto efectuar el move (x,y) equivale a revertir el arco $x \rightarrow y$.

Demostración: La haremos en dos tiempos.

1) Por definición, x e y pertenecen a la misma máquina, digamos i. La operación x precede a y puesto que el arco $x \rightarrow y$ pertenece al camino crítico. Veamos que esa precedencia es inmediata. Si hubiese una operación z entre x e y en la máquina i, habría un camino entre x e y de peso p_x+p_z pero el camino más largo entre esos dos vértices es $x \rightarrow y$ de peso p_x que figura en el camino crítico. Notemos que hemos usado fuertemente el hecho de considerar exclusivamente operaciones con tiempos no nulos.

2) Como vimos recién, x e y son consecutivos en la máquina i-ésima y por lo tanto π_i es de la forma AxyB, donde A y B son dos subsecuencias de π_i. Si efectuamos el move (x,y) resulta π_i' que es de la forma AyxB, luego, la única precedencia que cambia transcurre entre x e y.

Proposición: Sea π una secuencia factible. Entonces, efectuando un move $(x,y) \in V(\pi)$ resulta una nueva secuencia factible π'.

Demostración: Partimos de un grafo $G(\pi)$ que no tiene ciclos. Como vimos, un move (x,y) equivale a revertir un único arco $x \rightarrow y$ de $G(\pi)$. Por lo tanto, un ciclo en $G(\pi')$ debe pasar por el arco $y \rightarrow x$. Pero esto implica la presencia de un camino C que va desde x hasta y, el cual sólo puede provenir heredado de $G(\pi)$.

Veamos que un tal camino no existe en $G(\pi)$. En primer lugar, C no puede tener vértices intermedios porque de lo contrario tendría un peso mayor que el camino xy el cual figura en el camino crítico. Entonces C se reduce a un arco entre x e y que pertenece a A y x e y son operaciones consecutivas de un trabajo que se realizan en una misma máquina. Pero recordemos que desde el principio habíamos vedado esa posibilidad.

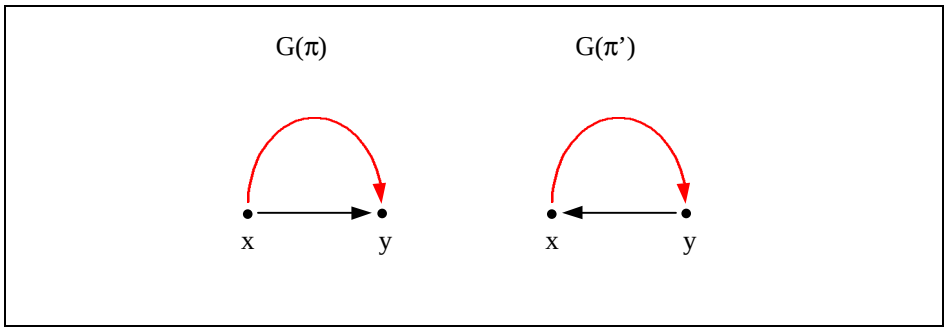


FIGURA: Una configuración como en $G(\pi)$ produce un ciclo en $G(\pi')$.

A continuación, mostraremos una propiedad que nos asegura que durante el curso de la ejecución del Taboo Search siempre tendremos a disposición algún move para elegir y proseguir con nuestro camino y si así no fuere, podemos estar persuadidos de que hemos alcanzado el óptimo.

Proposición: Si $V(\pi)=\varnothing$ entonces π es la solución óptima.

Demostración: Sea $u_1 \ldots u_n$ el camino crítico en $G(\pi)$. Por definición $V(\pi)$ puede ser vacío sólo en dos posibles casos.

1) Caso $a_j=b_j$ para todo $1\leq j\leq r$. Esto significa que todos los bloques tienen un único elemento, luego u_i y u_{i+1} pertenecen a distintas máquinas para todo $1\leq i\leq n-1$, luego $u_i\rightarrow u_{i+1}\in A$ para todo $1\leq i\leq n-1$, luego todas las operaciones del camino crítico pertenecen a un mismo trabajo k. Aún más, el camino crítico está formado por todas las operaciones de ese trabajo, ya que de lo contrario podrían insertarse las operaciones faltantes y conseguir un camino más largo.

Pero para toda secuencia π' el grafo $G(\pi')$ contiene el camino $u_1\ldots u_n$ ya que es un trabajo y por lo tanto el makespan $C(\pi')\geq p_{u_1}+\ldots+p_{u_n}=C(\pi)$.

2) Caso r=1. Aquí todas las operaciones del camino crítico pertenecen a la misma máquina. Razonando de igual manera sabemos que todas las operaciones de esa máquina están en ese camino crítico. Pero para toda secuencia π' la máquina en cuestión debe procesar sucesivamente las operaciones $u_1\ldots u_n$ en algún orden y por lo tanto el makespan $C(\pi')\geq p_{u_1}+\ldots+p_{u_n}=C(\pi)$.

La idea central de cualquier método iterativo como el taboo search es recorrer soluciones hasta encontrar una que sea óptima. Pero ahora, nos preguntamos, ¿es posible, partiendo de una solución inicial cualquiera llegar a una solución óptima realizando únicamente moves? Van Laarhoven, Aarts y Lenstra (1992) aseguran este resultado mediante una proposición.

Proposición: (propiedad de conexidad) Dado una secuencia $\pi^0\in\Pi$ arbitraria , existe una sucesión finita de secuencias $\pi^0,\ldots,\pi^n$ tal que π^{i+1} pertenece al entorno de π^i para todo $0\leq i\leq n-1$ y π^n es una solución óptima.

Demostración: No se encuentra.

3.3.2 Evolución del move

¿Porqué los moves se eligen así? Contestar esta pregunta es el objetivo central de esta sección y no puede realizarse con claridad sin contemplar la evolución que sufrieron los moves y que le otorgaron la complejidad que hoy conocemos. Este desarrollo es el resultado de mucho tiempo de investigación y experimentación tendiente hacia una misma aspiración, la de obtener entornos eficientes, lograda a través de la política de desechar aquellos intercambios que se saben a priori no provechosos. Sin embargo, intentaremos esbozar los principios sobre los que se basa esa manera tan peculiar de intercambio.

Nuestra meta última consiste en disminuir el makespan y el método básico que adoptamos para alcanzar dicho fin es intercambiar el orden de un par de operaciones consecutivas. Comenzamos entonces considerando el conjunto de moves

$V''(\pi)=\{(u,v)$: u y v son operaciones sucesivas en la misma máquina $\}$

Ahora observemos que, nuestra búsqueda es eficiente en la medida en que nuestro conjunto de intercambios deseche los movimientos que no nos conduzcan a una mejor solución. Detengámonos en la siguiente proposición

Proposición: Sea $G(\pi)$ un grafo y sea $(u,v)\in V''(\pi)$ un par de operaciones que se realizan consecutivamente en la misma máquina. Si la rama $u\rightarrow v$ no pertenece al camino crítico de $G(\pi)$ entonces el intercambio (u,v) no reduce el makespan.

Demostración: Como vimos anteriormente, un intercambio en el orden de dos operaciones consecutivas u y v es equivalente a la reversión del arco $u\rightarrow v$. Pero si esta rama no está presente en C, el camino crítico en $G(\pi)$, entonces el camino C está presente, luego del intercambio, en $G(\pi')$. Por lo tanto

$C(\pi')$ = costo del camino más largo en $G(\pi')\geq$ longitud de C =

= costo del camino más largo en $G(\pi)=C(\pi)$

Entonces, eliminemos los intercambios que se sabe no van a mejorar el makespan y entonces nos queda el conjunto (Van Laarhoven, 1992)

$V'(\pi)=\{(u,v)$: u y v son operaciones sucesivas en la misma máquina, u y v son operaciones sucesivas en el camino crítico de $G(\pi)\}$

Se verifica que V'(π) es más eficiente que V"(π) en el sentido en que nos ahorra revisar intercambios que la proposición anterior nos indica que no serán convenientes. Además, si recordamos que un bloque era un conjunto de operaciones de la misma máquina adyacentes en el camino crítico, es fácil ver que podemos reformular V'(π) en términos de bloques

V'(π)={(u,v) : u y v sucesivas en el mismo bloque}

Ahora, queremos continuar achicando el entorno y consideramos el conjunto definido anteriormente

V(π)={(u,v) : u y v son las dos primeras (si no es el bloque final) o dos últimas (si no es el bloque inicial) operaciones de un bloque}

Sorprendentemente, si nos quedamos únicamente con las operaciones en el extremo de los bloques, se conserva la performance de las soluciones del entorno. Como $V(\pi)\subseteq V'(\pi)$ esperaríamos encontrar moves más provechosos en V'(π) que en V(π), sin embargo, la siguiente proposición nos muestra que los moves que no están en V(π) no son promisorios.

Proposición: (Matsuo, Suh y Sullivan, 1988) Si α es una secuencia que se obtiene de π haciendo un intercambio $(u,v)\in V'(\pi)\setminus V(\pi)$
entonces el makespan de G(α) no es menor que el de G(π), es decir, $C(\alpha)\geq C(\pi)$.

Demostración: Un plomo, ver en trabajo de Nowicki-Smutnicki.

Finalmente, hemos arribado en el conjunto V(π) que mantiene las mejores soluciones y una cantidad de elementos sensiblemente menor que el conjunto de moves inicial. En efecto,

$$\#V''(\pi)=\sum_{i=1}^{m}(\#M_i-1)$$

$$\#V'(\pi)=\sum_{i=1}^{r}(\#B_i-1)$$

$$\#V(\pi)=\sum_{i=2}^{r-1}\min\{2,\#B_i-1\}+\min\{1,\#B_i-1\}+\min\{1,\#B_r-1\}$$

3.4 Taboo List

Con el objetivo de mejorar la eficiencia del proceso de búsqueda el Taboo Search utiliza además de la información local proporcionada por el entorno, información obtenida anteriormente por el mismo proceso de búsqueda. Esto obedece a la intención de obtener una idea global del conjunto de soluciones, una visión panorámica del relieve en contraste a la visión precisa pero estrecha que nos brinda el entorno. Este uso sistemático de información almacenada es una característica esencial del Taboo Search.

El algoritmo se vale de esa percepción global para evitar caer en un ciclo y para orientar la búsqueda en nuevas direcciones y lo hace reduciendo el entorno de una solución mediante la remoción de elementos considerados indeseables o tabú. Esta soluciones tabú son almacenadas en una estructura de datos llamada *lista tabú (taboo list).*

La lista tabú se basa en los últimos eventos transcurridos y el alcance de esta retrospectiva es fijado previamente. Aquellas soluciones que deben ser evitadas son representadas en la lista en términos de moves en concordancia con el proceso de búsqueda que manifiesta preferencia por dicho enfoque.

La lista taboo T es una lista ordenada de un largo prefijado de moves prohibidos. Comenzando con una lista vacía, incorporamos nuevos elementos a T eliminando el elemento más antiguo de la lista toda vez que se supere el largo prescrito. Podemos pensar que el ingreso de un move consiste en un shift a la izquierda sobre T seguido de la incorporación por derecha del nuevo move. Cada vez que se efectúa un move (x,y) se incorpora su inverso (y,x) a la lista taboo.

incorporar	lista T
u_1	$\{u_1\}$
u_2	$\{u_1, u_2\}$
u_3	$\{u_1, u_2, u_3\}$
u_4	$\{u_2, u_3, u_4\}$

TABLA: Supongamos que vamos a incorporar sucesivamente los moves u_1, u_2, u_3, u_4 a una lista T inicialmente vacía y con una longitud máxima fijada de tres elementos.

3.5 Neighborhood Searching Procedure

Una vez delineados los paradigmas de entorno y lista tabú, nos adentramos en un método denominado *NSP (neighborhood searching procedure)* diseñado para elegir una solución (o su move equivalente) de entre las que nos facilita un entorno dado.

Esta elección comporta la consideración de elementos tales como el beneficio que reporta cada solución y la prohibición impuesta por la lista tabú. Evidentemente, tenemos predilección por las soluciones que se descubren beneficiosas y ausentes de la lista tabú, pero estas elevadas aspiraciones no siempre son colmadas y gran parte de la eficacia de nuestro algoritmo descansa sobre la política que adoptemos al contemplar este caso.

Aquí advertimos in situ ese juego entre localidad y globalidad que habíamos mencionado al tratar el tema de la lista tabú, que se materializa en el NSP.

La tarea aquí, consiste en revisar el entorno de una secuencia dada π en busca de una secuencia π' que conduzca a una mejora, o lo que es equivalente, encontrar un move apropiado para efectuar sobre la secuencia π. Sean dadas la secuencia π, el conjunto de moves no vacío $V(\pi)$, la lista taboo T y el valor del mejor makespan conocido C^*.

Vamos a distinguir entre tres clases de moves: los no prohibidos, los prohibidos que mejoran C^* y los prohibidos que no mejoran C^*. Consideremos el conjunto H que reúne los moves de las dos primeras categorías. Si H no es vacío, elegimos el mejor move de ese conjunto. Si es vacío y $V(\pi)$ tiene un solo elemento, elegimos ese move. Si no vale ninguno de los anteriores, tomamos el último elemento que ingresó a T y lo añadimos a T reiteradamente hasta que aparezca un move no prohibido y elegirlo.

Una vez obtenido, efectuamos el move seleccionado (x,y) sobre la secuencia π resultando una nueva secuencia π' e introducimos (y,x) en T.

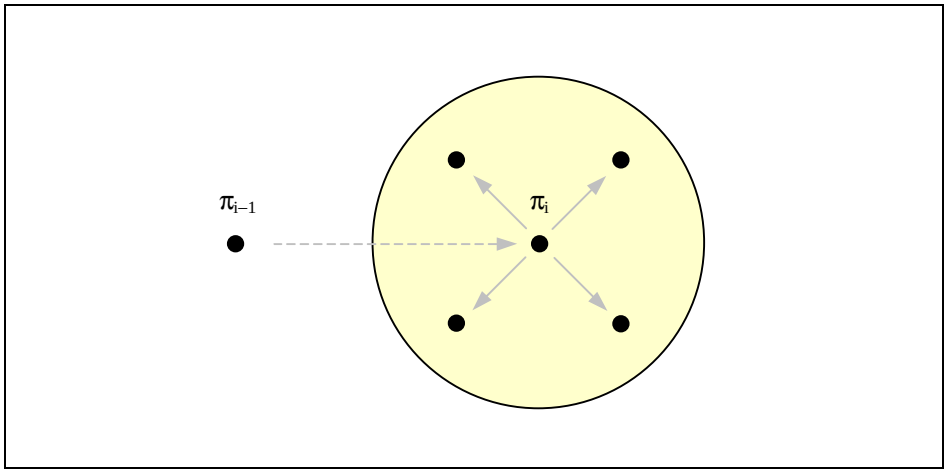


FIGURA: Dada una secuencia π_i , NSP elige un move.

EJEMPLO: Sean una secuencia π, unos moves u_1, u_2, u_3, u_4, u_5, una lista taboo $T=\{u_4, u_3, u_5\}$ con una longitud máxima de tres elementos y un makespan C^*. Analicemos tres casos.

Caso 1) Si nuestro conjunto de moves fuera $V(\pi)=\{u_1, u_2, u_3\}$ con u_1, u_2, u_3 como en la tabla, entonces $H=\{u_1, u_2\}$ y elegimos como v al move más beneficioso de H.

Caso 2) Si fuera $V(\pi)=\{u_3\}$ con u_3 como en la tabla, entonces $H=\varnothing$ y como V tiene un solo elemento, elegimos $v=u_4$.

Caso 3) Y si fuera $V(\pi)=\{u_3, u_4\}$ con u_3, u_4 como en la tabla, entonces $H=\varnothing$. Reingresamos a T su elemento más actual u_5, resultando $T=\{u_3, u_5, u_5\}$ y $H=\{u_4\}$. Elegimos $v= u_4$.

Finalmente y en cualquiera de los tres casos, incluimos en T el par (y,x) donde v= (x,y).

Caso	moves			T	H	v
	no prohibidos	prohibidos y provechosos	prohibidos y no provechosos			
1	u_1	u_2	u_3	$\{u_4, u_3, u_5\}$	$\{u_1, u_2\}$	el mejor en H
2			u_3	$\{u_4, u_3, u_5\}$	$\varnothing$	u_3
3			u_3, u_4	$\{u_4, u_3, u_5\}$	$\varnothing$	
		u_4	u_3	$\{u_3, u_5, u_5\}$	$\{u_4\}$	u_4

TABLA: Se muestra un esquema de elecciones de T, H y v según distintas alternativas.

3.6 Taboo Search Algorithm

El procedimiento consiste en recorrer, guiado u orientado por el NSP, un camino en el conjunto de secuencias. Cada paso introduce un avance a una nueva secuencia.

Comenzamos con una secuencia inicial π y una lista taboo T vacía. En cada iteración, recogemos una colección de moves $V(\pi)$ y luego aplicamos NSP para obtener una nueva secuencia π' y una nueva lista taboo T'. Actualizamos π y T y si mejoramos C^*, actualizamos C^* y su correspondiente secuencia π^*.

El algoritmo se detiene en dos casos: cuando $V(\pi)$ es vacío y entonces hemos encontrado un óptimo y cuando el número de iteraciones sin mejorar supera una constante fijada con anterioridad.

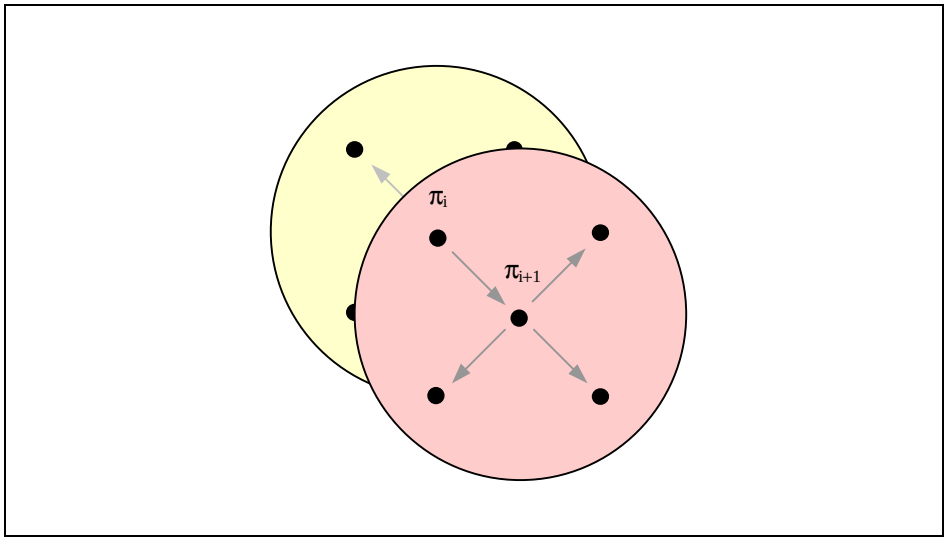


FIGURA: Un paso del NSP. A partir de una solución dada π , se construye el entorno de π y se elige una nueva solución π', se construye el entorno de π'...

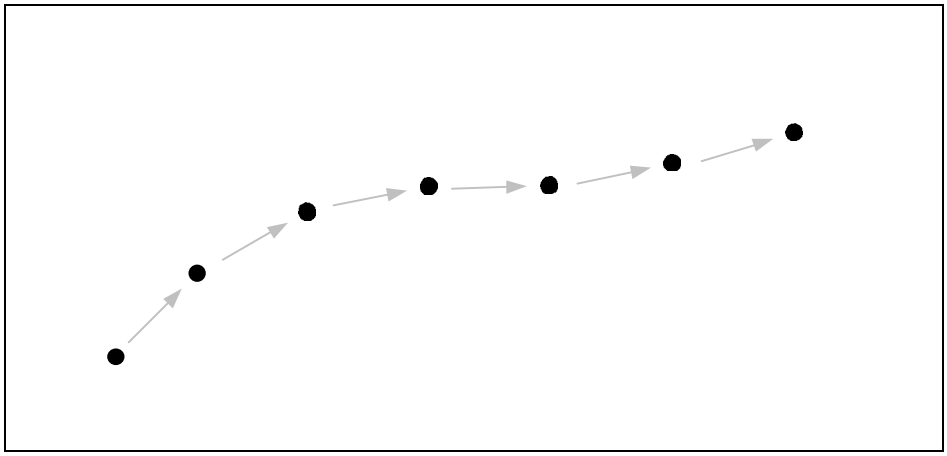


FIGURA: Camino de secuencias creado por TSA.

EJEMPLO: Sea la secuencia 1, 2, 3 en máquina 1 y 4, 5, 6 en la máquina 2. Si T={(2,3), (4,6), (5,6)} y v=(1,2) el move seleccionado, entonces la secuencia queda 2, 1, 3 en máquina 1 y 4, 5, 6 en la máquina 2 y T={(4,6), (5,6), (2,1)}.

3.7 Back Jump Tracking

Proponemos modificar el TSA incorporando una medida de diversificación, que consiste en retroceder un trecho una vez que se ha encontrado el fin de un camino y desviando el rumbo, emprender otro recorrido desde allí.

Contamos con una lista L de largo prefijado para guardar los puntos donde se ramifica el camino. Comenzamos con una lista L vacía y eliminamos de L el elemento más antiguo toda vez que al introducir un nuevo elemento se supere la longitud estipulada.

Durante la corrida de TSA, si una secuencia π mejora π^*, guardamos en L la terna formada por π, la lista taboo y el conjunto $V(\pi)$ excluyendo el move seleccionado por NSP para efectuarse sobre π.

Una vez concluído TSA, volvemos a aplicarlo sobre una terna que extraemos de L y puesto que hemos eliminado el move que naturalmente se elegiría, obligamos al algoritmo a optar por un nuevo camino, generando una desviación o ramificación del camino original. Y así continuamos hasta que la lista L quede vacía.

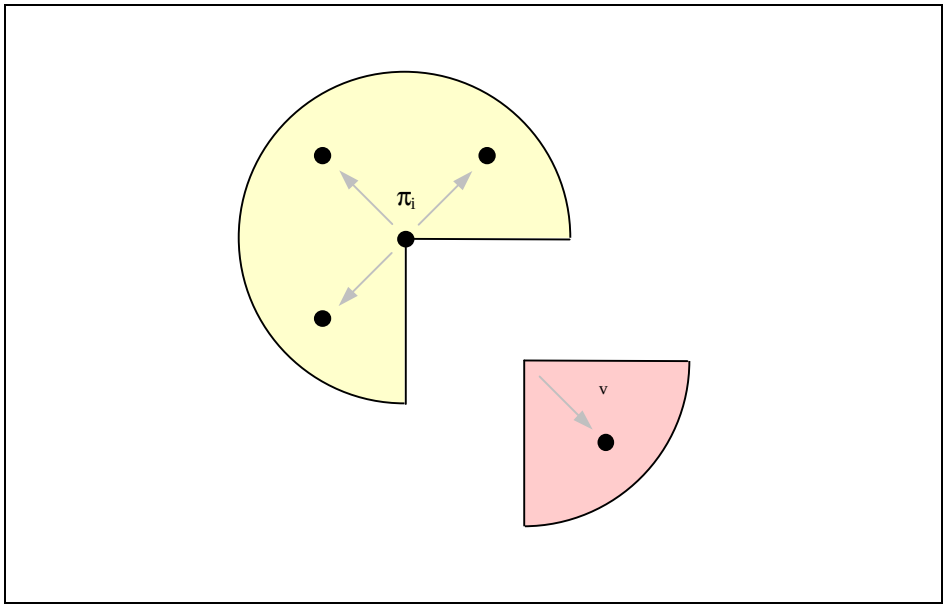


FIGURA: Las secuencias que marcan una mejoría, como π_i, son consideradas para realizar back jump. Si v es el move que se elige para pasar a la siguiente secuencia, se guarda en la lista L la terna $(\pi_i, V(\pi_i)-\{v\}, T)$.

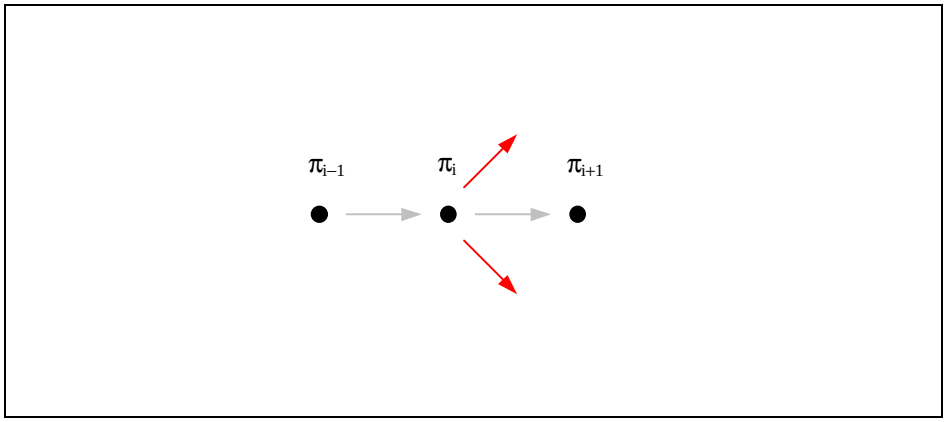


FIGURA: Cuando realizamos un back jump, regresamos a π_i y como habíamos excluido el move que conduce a π_{i+1} tomamos forzosamente un nuevo camino.

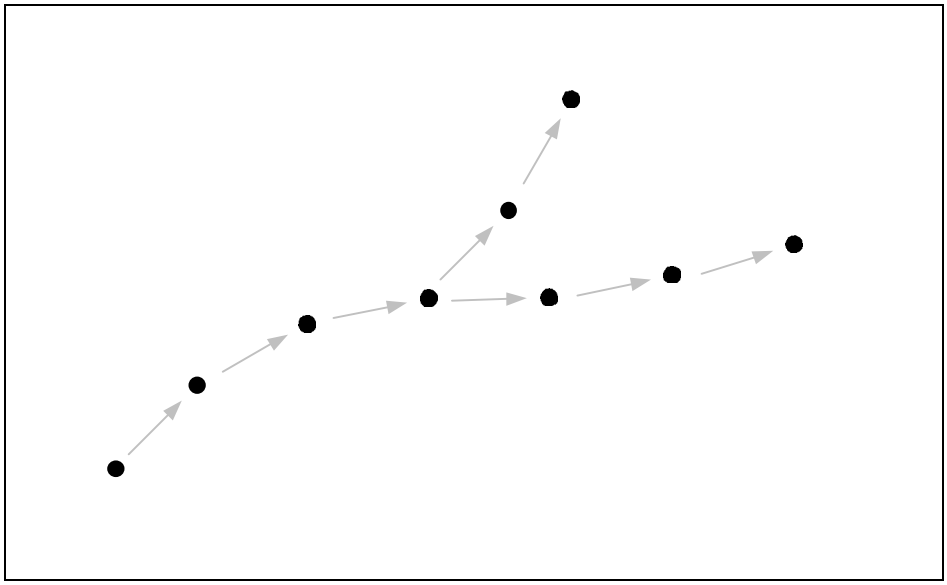


FIGURA: Continuando con esa impronta, resulta un camino ramificado.

3.8 Pseudocódigo de Taboo Search Algorithm

Algoritmo de Taboo search con back jump tracking.

TSAB
comienza con un grafo G=(O,A) y valores máximos permitidos de cantidad de iteraciones sin mejorar, largo de lista L y largo de lista T
retorna la mejor secuencia y makespan conocido π^* y C^*

sean L y T listas vacías
crear una secuencia factible π, calcular makespan C y camino crítico
hacer $\pi^*=\pi$, $C^*=C$, i=0
repetir hasta que el número de iteraciones sin mejorar i supere máximo permitido
    si el actual π no ha sido cargado de la lista L
        encontrar moves V=V(π)
        si V es vacío
            entonces π^* es secuencia óptima, retornar secuencia π^* y makespan C^*
    obtener nuevo π , nuevo T y move v aplicando NSP
    si hay activa una orden de guardar
        si V−{v} no vacío
            guardar (π', V−{v}, T') en L, si el largo de L supera el máximo permitido desecho el elemento más antiguo
        desactivar orden de guardar
    calcular makespan C y camino crítico
    si $C<C^*$
        actualizar mejor conocido $\pi^*=\pi$, $C^*=C$,
        fijar valores para guardar $\pi'=\pi$, T'=T y activar orden de guardar
        hacer i=0
    sino
        incrementar i en uno
si L no es vacío
    extraer elemento más reciente de L y obtener secuencia π, conjunto de moves V y lista taboo T
    activar orden de guardar
    volver al bucle anterior con i=0
retornar secuencia π^* y makespan C^*

Dada una secuencia, construye el grafo respectivo y calcula en makespan y camino crítico

calcular makespan y camino crítico
comienza con una secuencia dada π, y un grafo G=(O,A)
retorna makespan y camino crítico

sea $S(\pi)=\{(i,j)$: i,j operaciones distintas correspondientes a la misma máquina, i precede a j según el orden dado por $\pi\}$
hacer $G(\pi)=(O,A\cup S(\pi))$
obtener makespan y camino crítico aplicando LPS
retornar makespan y camino crítico

Neighborhood searching procedure. Elige un move entre varios candidatos para realizar.

NSP
comienza con una secuencia π, un conjunto de moves V, una lista taboo T y un makespan C^*
retorna un move seleccionado v, una nueva secuencia π y una nueva lista taboo T

para todo π' en el entorno de π dado por V
    calcular makespan C(π')
sea H={w : w es un move que no pertenece a T} $\cup$
    {w : w move que pertenece a T, C(π')<C^* donde π' se obtiene de π haciendo w}
si el conjunto V\T$\cup$H es no vacío
    v = mejor move de ese conjunto
sino si V tiene un solo elemento
    v = elemento de V
sino
    sea w = último elemento ingresado a T
    repetir mientras V\T sea vacío
        meter w en T, si T tiene más de maxt elementos desechar el más antiguo
    v = elemento de V\T
hacer move v=(x,y) en π, meter (y,x) en T, si el largo de T supera el máximo permitido desechar
    el elemento más antiguo
retornar v, π, T

Encuentra un conjunto de moves convenientes para realizar.

encontrar moves
comienza con un camino crítico $u_1, \ldots, u_n$
retorna conjunto de moves V

partimos el camino crítico en bloques B tales que
    i) B={$u_p, \ldots, u_q$}
    ii) $u_p, \ldots, u_q$ son consecutivos en el camino crítico y se efectúan en una máquina μ
    iii) la operación anterior a u_p no se efectúa en la máquina μ (p>1)
    iv) la operación posterior a u_q no se efectúa en la máquina μ (q<n)
sea V un conjunto vacío
si existe más de un bloque
    repetir para todo bloque B con por lo menos dos elementos
        sean x, y, w, z la primera, segunda, penúltima y última operaciones del bloque
            B
        si B es el bloque inicial
            meter (w,z) en V
        sino si B es bloque final
            meter (x,y) en V
        sino
            meter (x,y) y (w,z) en V
retornar V

Longest path search. Dado un grafo, encuentra el camino crítico, el makespan, etc. Es el algoritmo de Ford modificado según vimos en 2.3.2.

LPS

comienza con un grafo G = (O,R) tal que toda rama $(u,v) \in R$ tiene un peso asignado p_v
retorna makespan, camino crítico y valores L(s,·), L(·,t)

para cada $u \in O$
    sean L(s,u)=0, L(u,t)=0, $\varepsilon(u)=-1$
repetir #O−1 veces
    repetir para toda rama $(u,v) \in R$
        si $L(s,v) < L(s,u) + p_v$
            $L(s,v) = L(s,u) + p_v$ y $\varepsilon(v)=u$
        si $L(u,t) < L(v,t) + p_v$
            $L(u,t) = L(v,t) + p_v$
el makespan es C=L(s,t) y los valores de $\varepsilon(u)$ indican cómo recorrer el camino óptimo de atrás
    para adelante comenzando por t
retornar C, camino crítico y valores L(s,·), ·,t)

Capítulo IV
Shifting Bottleneck Procedure

4.1 Generales

Presentamos un procedimiento heurístico para hallar soluciones cuasi óptimos del problema de Job Shop Scheduling. El método presentado es de Balas-Vazacopoulos (1998) y es un híbrido de varios métodos de optimización y estructuras de datos. El primero es el GLS, un procedimiento de búsqueda local que utiliza un esquema de intercambios y se estructura por medio de árboles. El segundo es el SBP (Adams, 1988) que secuencia máquinas sucesivamente.

4.2 Preliminares

Antes de comenzar a estudiar el algoritmo en detalle, vamos a definir algunas nociones y esbozar una propiedad que necesitaremos más adelante.

El algoritmo expuesto en el capítulo anterior, el TSA, es de tipo iterativo y como tal opera mediante soluciones factibles las cuales son representadas por una secuencia. El SBP, en cambio, es un algoritmo de tipo constructivo que no maneja soluciones acabadas sino que va confeccionando una solución óptima empleando hasta entonces soluciones parciales. Más precisamente, va recorriendo las máquinas de a una por vez y asigna una secuencia en cada una de las máquinas a medida que las va visitando. Es por esto que necesitamos ahora de una noción que represente una de esas "soluciones parciales".

Entonces recurrimos a la siguiente definición. Decimos que S es una *selección* si resulta de quitar de $S(\pi)$ los arcos correspondientes a algunas máquinas. Y que es *completa* si no se ha eliminado arco alguno, es decir, todas las máquinas están secuenciadas.

Para cada operación u notamos $\alpha(u)$ y $\gamma(u)$ a la operación que precede y a la que sucede a u dentro del trabajo al que pertenece. Observemos que $(\alpha(u),u)$ y $(u,\gamma(u))$ pertenecen a A. Análogamente, notamos $\beta(u)$ y $\delta(u)$ a la operación (si existe) que precede y a la que sucede a u dentro de la máquina a la que pertenece. Además $(\beta(u),u)$ y $(u,\delta(u))$ pertenecen a S.

Definamos L(u,v) al período de tiempo comprendido entre el instante en que comienza la operación u y el instante en que culmina la operación v. Entonces quedan también definidos L(s,u), L(u,t) y L(s,t) como el tiempo desde el comienzo de la producción hasta u, desde u hasta finalizar la producción y el tiempo total que dura la producción, respectivamente.

Para calcular cualquiera de estos valores nos basamos en una proposición ligeramente más general que la utilizada para calcular el makespan.

Proposición: Consideremos el grafo $G=(O,A\cup S)$ definido como antes. Si tomamos como peso de cada rama (i,j) el valor p_i entonces L(u,v) = longitud del camino más largo entre u y v.

Demostración: Análoga a demostrar que L(s,t) es igual al makespan.

operaciones	s	1	2	3	4	5	6	7	t
trabajo		1	1	1	1	2	2	2	
máquina		1	2	3	1	3	2	1	
tiempo		10	10	10	10	20	20	20	
L(s,·)	0	0	10	20	30	30	50	70	90
L(·,t)	90	90	80	70	30	60	40	20	0
α()		s	1	2	3	s	5	6	
γ()		2	3	4	t	6	7	8	
β()					1	3	2	4	
δ()		4	6	5	7				

EJEMPLO: Sea un caso con trabajos, máquinas y tiempos como en la tabla. Si tomamos la secuencia 1, 4, 7 en primera máquina, 2, 6 en segunda y 3, 5 en tercera quedan como sigue.

4.3 El entorno

En esta sección expondremos la definición particular de intercambio y de entorno de una selección según los cánones fijados por el SBP. Luego prescribiremos una serie de condiciones para asegurarnos que un intercambio sobre una selección factible produzca otra selección también factible.

Dada una selección S, definimos un entorno de S como el conjunto de las selecciones que se obtienen de S a través de una perturbación específica o *interchange (intercambio)*. La perturbación que tenemos en mente reside en tomar un par de operaciones u, v afectadas a la misma máquina, u precediendo a v según S y crear otra selección S' moviendo u inmediatamente después de v llamado *forward interchange (intercambio hacia adelante)* o bien, moviendo v inmediatamente antes de u llamado *backward interchange (intercambio hacia atrás)*.

Vamos a pedir que tanto u como v pertenezcan al camino crítico y que todas las operaciones en la porción de camino crítico entre u y v inclusive pertenezcan a la misma máquina. Esto responde al deseo de destruir el camino crítico, el cual es la barrera que nos impide reducir el makespan.

Además, exigimos que se cumplan a modo de condiciones y según sea el tipo de intercambio considerado, alguna de las hipótesis de las siguientes proposiciones para asegurar que la nueva selección S' determine un grafo acíclico.

Proposición: Si el camino crítico contiene a u, v, $\gamma(v)$ y vale $L(v,t) \geq L(\gamma(u),t)$ entonces un forward interchange sobre u y v produce una selección acíclica.

Proposición: Si el camino crítico contiene a u, v, $\alpha(u)$ y vale $L(s,u)+p_u \geq L(s,\alpha(v))+p_{\alpha(v)}$ entonces un backward interchange sobre u y v produce una selección acíclica.

Finalmente, llamaremos *pares críticos* a los pares de operaciones que satisfagan todas las restricciones anteriores. Y entonces, el entorno queda definido como el conjunto de selecciones que se obtienen de una dada a través de un intercambio sobre un par crítico.

EJEMPLO: Supongamos que en una máquina se hacen las operaciones 1, 2, 3, 4, 5, 6 secuenciadas en ese orden. Un forward interchange sobre 2 y 5 se arma extrayendo la operación 2 y colocándola inmediatamente siguiendo la 5, arrojando una secuencia nueva 1, 3, 4, 5, 2, 6. Esto involucra revertir el sentido de los arcos (2,3), (2,4) y (2,5) puesto que el intercambio provocó un cambio en el orden entre 2 y el bloque formado por 3, 4 y 5.

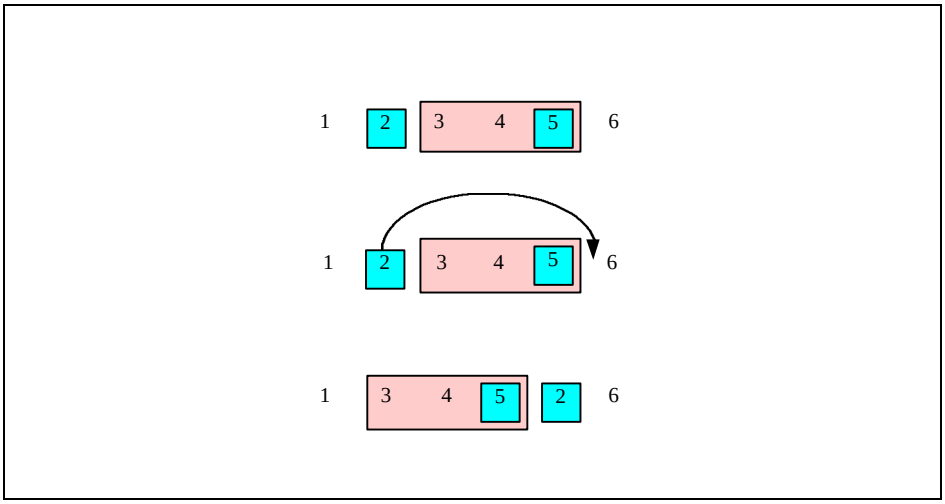


FIGURA: Forward interchange.

EJEMPLO: Análogamente un backward interchange sobre 2 y 5 crea la secuencia 1, 5, 2, 3, 4, 6. Involucra revertir sentido de los arcos (2,5), (3,5) y (4,5), puesto que se invirtió el orden entre 5 y el bloque compuesto por 2, 3 y 4.

Notemos que un intercambio sobre vértices adyacentes es indistinto si es forward o backward.

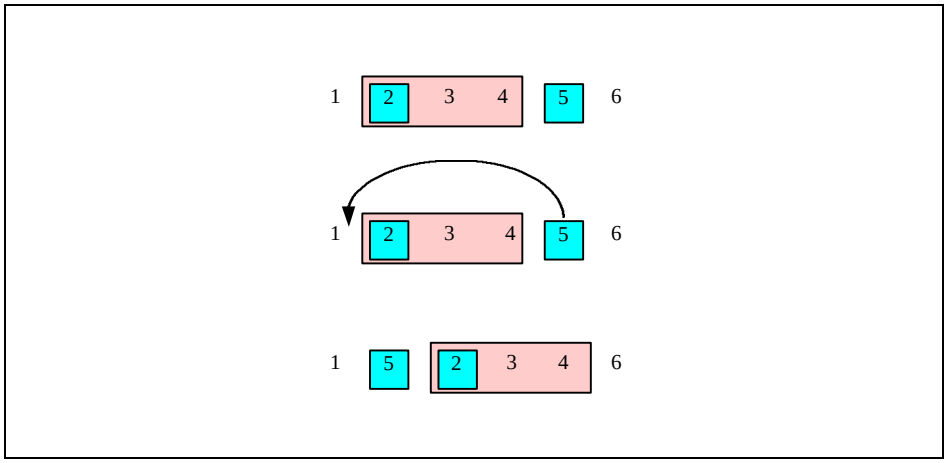


FIGURA: Backward interchange.

4.3.1 Marco teórico del intercambio

A continuación mostramos que pedir que $\alpha(u)$ y $\gamma(v)$ pertenezcan al camino crítico es una condición necesaria para que haciendo un intercambio sobre u y v se reduzca el makespan (Potts, 1980).

Proposición: Si ni $\alpha(u)$ ni $\gamma(v)$ pertenecen al camino crítico entonces un intercambio sobre u y v no reduce el makespan.

Demostración: No la encuentro.

Ahora, demostramos las proposiciones de la sección anterior que aseguran que un intercambio sobre una selección acíclica produce otra selección acíclica.

Proposición: Si el camino crítico contiene a u, v, $\gamma(v)$ y vale $L(v,t) \geq L(\gamma(u),t)$ entonces un forward interchange sobre u y v produce una selección acíclica.

Demostración: Por contradicción. Supongamos que comenzamos con un grafo acíclico G y que trasladando u después de v se crea un ciclo C en G'. Entonces, C contiene el arco $(u,\gamma(u))$ o bien contiene el arco $(u,\delta(v))$. Si $(u,\gamma(u))$ pertenece a C hay un camino en G desde $\gamma(u)$ hasta v y entonces $L(v,t)<L(\gamma(u),t)$ lo cual es contrario a lo asumido. Si $(u,\delta(v))$ pertenece a C hay un camino desde $\delta(v)$ hasta v lo cual es contrario a asumir que G es acíclico.

Proposición: Si el camino crítico contiene a u, v, $\alpha(u)$ y vale $L(s,u) + p_u \geq L(s,\alpha(v)) + p_{\alpha(v)}$ entonces un backward interchange sobre u y v produce una selección acíclica.

Demostración: Análoga a la anterior.

4.4 Guideposts

En esta sección se muestra una estrategia para hacer más eficiente la búsqueda de selecciones óptimas que se basa en restringirse a intercambios que ataquen una porción determinada del camino crítico que se considere particularmente mala. Más precisamente, si luego de efectuar un intercambio empeoró el makespan y se sabe que un tramo del camino crítico no se comportó según lo esperado, ya sea creciendo más o ya sea disminuyendo menos de lo prometido, entonces acusamos una urgencia en remediar este hecho y nos centramos en intercambios sobre ese tramo.

Supongamos hemos efectuado en S un forward interchange sobre (u,v) y que esto ha producido una selección S' con un makespan que lejos de disminuir, se ha incrementado. Esto es,

$L'(s,t) > L(s,t)$

donde L' es análogo a L, evaluado en el nuevo grafo $G = (O, A \cup S')$.

Si además sucede que

i) $L'(s,v) > L(s,v) - p_u$

esto quiere decir que al menos, parte del incremento ocurrió en el segmento del camino crítico entre s y v y es atribuible al hecho que aunque movimos a u después de v, v no sufrió un "shift" hacia la izquierda de un valor p_u , ie: pese a quitar a u de delante de v, v no puede comenzar un tiempo p_u antes.

Por otro lado, si sucede que

ii) $L'(v,t) > L(v,t) + p_u$

significa que al menos, parte del incremento proviene del segmento entre v y t puesto que, insertando a u después de v, se ha incrementado en más de p_u la longitud del camino entre v y t.

Análogamente, si luego de un backward interchange que incrementó el makespan vale que

iii) $L'(s,u) > L(s,u) + p_v$

hay un incremento entre s y u y si vale

iv) $L'(u,t) > L(u,t) - p_v$

lo hay entre u y t.

Entonces, si se cumplen las desigualdades i) o iii) diremos que se ha alcanzado un *left guidepost* y restringiremos la nueva lista de pares críticos, a los pares de operaciones sobre el segmento del camino crítico comprendido entre s y v. En cambio, si se cumplen ii) o iv) diremos que se alcanzó un *right guidepost* y nos restringiremos a las operaciones entre v y t. Si se alcanzan a la vez right y left guidepost no aplicaremos ninguna restricción.

EJEMPLO: Supongamos el problema reflejado en el Gantt Chart de la figura. Las operaciones 1 y 2 pertenecen al mismo trabajo y se realizan en ese orden, las demás operaciones son todas de trabajos distintos.

Si hacemos un forward interchange entre u=3 y v=2 alcanzamos un left guidepost pues al quitar la operación u=3, esperaríamos que la operación v=2 se adelantase, por lo menos, un valor igual a $p_u = 10$ min. Pero como la operación 2 no puede comenzar antes de la finalización de la operación 1, sólo se adelante un valor igual a 5 min.

$L'(s,v) = 15 \quad > \quad L(s,v) - p_u = 20 - 10$

Entonces, restringimos nuestra lista de candidatos a intercambios, a los que se efectúan sobre los vértices 1,2

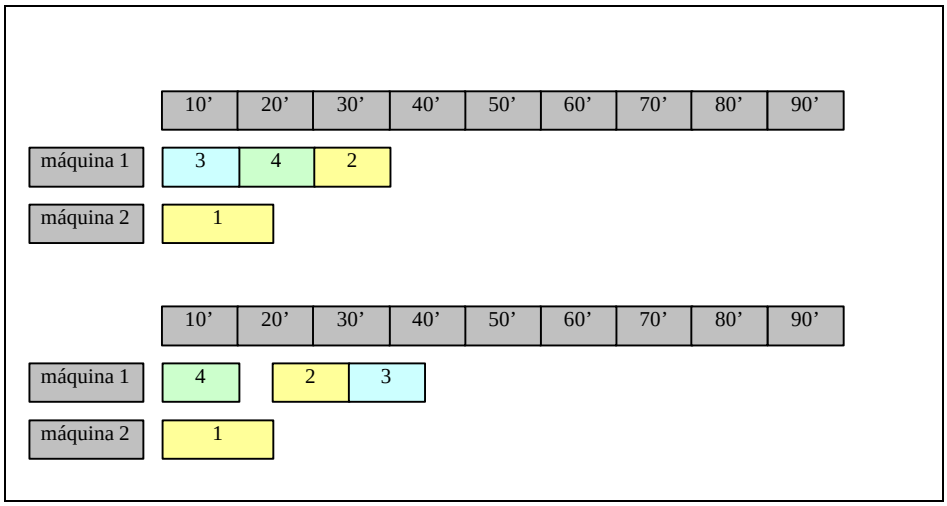


FIGURA: Forward interchange (3,2) que alcanza un left guidepost.

EJEMPLO: Ahora, consideramos un caso análogo al anterior. Si hacemos un forward interchange sobre u=1 y v=4 alcanzamos una right guidepost, puesto que, colocando la operación u=1 a continuación de la v=4, esperaríamos que el tiempo necesario para terminar la producción a partir de la operación v=4 se incrementase como máximo en un valor $p_u = 10$ min. Pero como la operación 2 no puede comenzar antes de la finalización de la operación 1, el corrimiento de la operación 1 "arrastra" la operación 2, alargando ese período.

$L'(v,t) = 30 \quad > \quad L(v,t) + p_u = 10 + 10$

Entonces, restringimos nuestra lista de candidatos a intercambios, a los que se efectúan sobre los vértices 3,4,1,2.

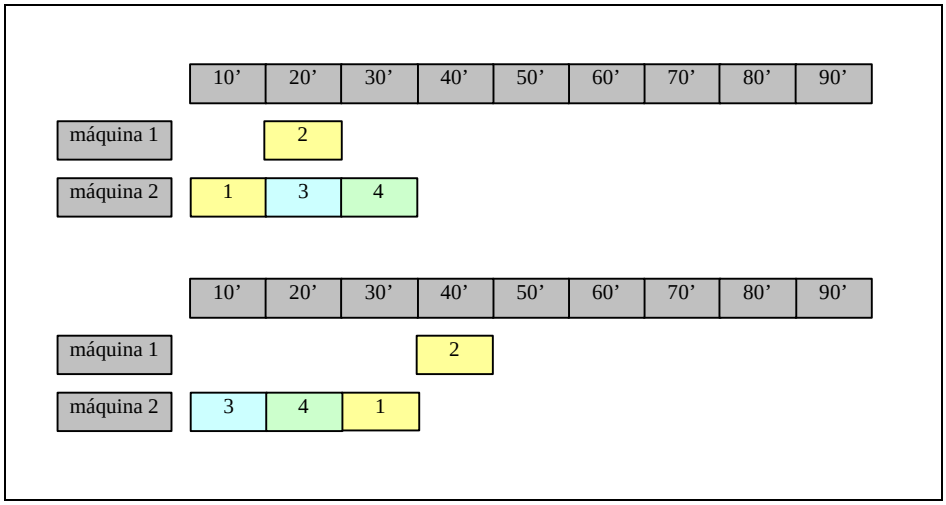


FIGURA: Forward interchange (1,4) que alcanza un right guidepost.

4.5 Neighborhood Trees

Aquí describimos el procedimiento NT el cual optimiza la secuencia en una máquina prefijada, dejando fijas las secuencias en las otras máquinas que estuvieren secuenciadas. Consiste en crear un árbol cuyos nodos corresponden a distintas selecciones, de tal manera que el hijo de un nodo se obtiene realizando un intercambio y por lo tanto pertenece al entorno de su padre.

Formalmente, un nodo viene dado por una selección, un makespan, una lista de arcos fijos y una lista de intercambios. Iniciamos con un árbol Q con un nodo raíz que tiene una selección, un makespan, una lista de arcos fijos vacía y un conjunto de intercambios. El paso general consiste en tomar un nodo no revisado de Q y generar su descendencia.

Tomemos entonces un nodo padre. Primeramente, formulamos una estrategia. Para evitar que nodos distintos produzcan la misma selección, cada vez que creamos un nodo le dejaremos fijos algunos arcos que además se heredarán a sus descendientes. Cuando creemos los hijos de un nodo, rechazaremos aquellos que resulten de intercambios que involucren la inversión de arcos fijos.

Supongamos que la lista de intercambios del padre es $\{(u_1,v_1),\ldots,(u_n,v_n)\}$. Entonces el primer hijo se obtiene heredando arcos fijos, realizando el intercambio (u_1,v_1) y fijando el arco (v_1,u_1) en ese hijo. El segundo hijo, heredando arcos fijos, fijando el arco (u_1,v_1), haciendo el intercambio (u_2,v_2) y fijando el arco (v_2,u_2)…El q-ésimo hijo, heredando arcos fijos, fijando los arcos $(u_1,v_1),\ldots,(u_{q-1},v_{q-1})$, haciendo el intercambio (u_q,v_q) y fijando el arco (v_q,u_q).

Para cada hijo, tenemos ya calculadas la selección y la lista de arcos fijos. Ahora calculamos el makespan, el camino crítico y hallamos la lista de intercambios. Finalmente, si el makespan es peor que el de su padre, estudiamos la existencia de un guidepost activo y si lo hubiere, eliminamos de la lista de intercambios del hijo, los que infrinjan esa restricción.

Y así seguimos hasta que todos los nodos de Q estén revisados.

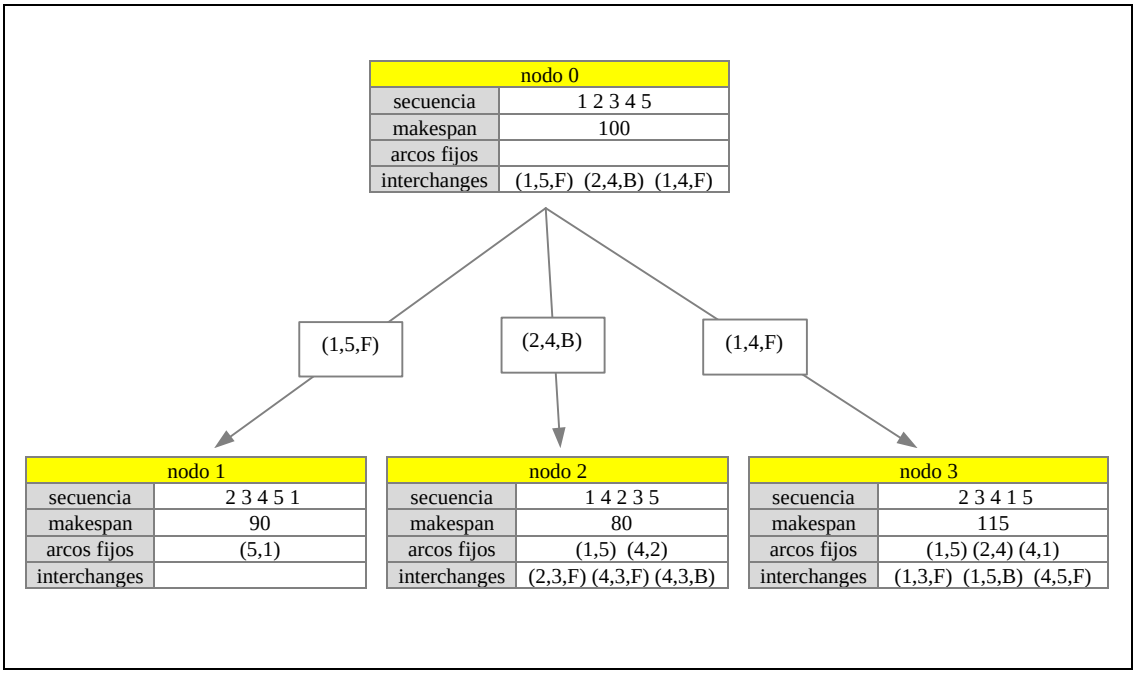


FIGURA: Desarrollamos un árbol a partir de un nodo 0. “F” indica *forward pair* y “B” indica *backward pair*.

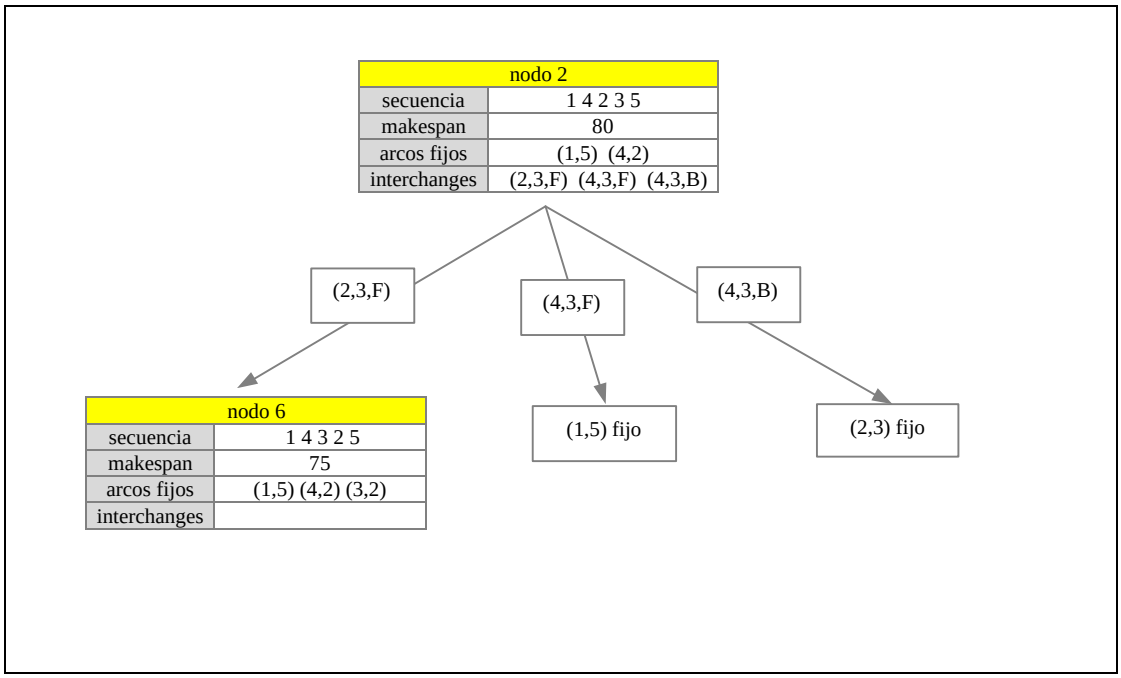


FIGURA: Creación de los hijos del nodo 2. Nótese cómo realizar el intercambio (2,3,F) veda los demás.

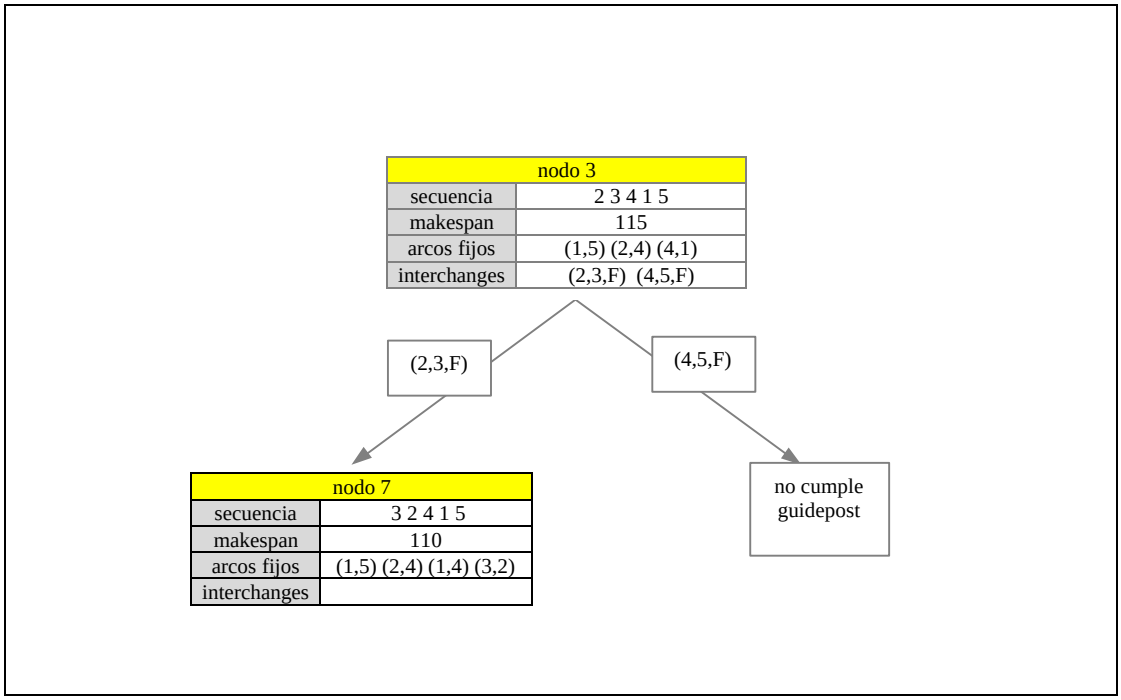


FIGURA: Notemos que cuando generamos el nodo 3 el makespan aumentó. Supongamos que alcanzamos un guidepost y que (4,5,F) no cumple la restricción.

4.6 Guided Local Search

Dada una selección S, una máquina μ y un entero k, la tarea de GLS consiste en aplicar k veces el procedimiento NT sobre la máquina μ. Para esto, le suministra un nodo raíz en cada iteración, inicialmente, crea el nodo raíz, posteriormente, lo elige entre los del árbol Q provisto por NT.

Sean dados una selección S y una máquina μ. Si la máquina μ no ha sido secuenciada, encontramos una secuencia factible e incorporamos los arcos correspondientes a la selección S. El nodo raíz queda formado con la secuencia S, el makespan, una lista de arcos fijos vacíos y la lista de intercambios.

Partiendo del nodo raíz generamos un árbol Q de selecciones aplicando NT sobre la máquina μ. Ahora, elegimos un nodo de Q para ser nodo raíz de la próxima iteración. Si algún nodo mejoró al mejor makespan conocido, elegimos al mejor nodo de Q. Si esto no ocurre, creamos una lista L de los k mejores nodos de Q que hayan superado el nivel $[\log_2 (\text{cantidad de máquinas} * \text{cantidad de trabajos})]$ y elegimos aleatoriamente entre los elementos de L, asignando una probabilidad $K/2^i$ al i-ésimo mejor nodo (i=0,…,k-1). Y si ningún nodo de Q superó al mejor makespan conocido ni alcanzó el nivel requerido simplemente tomamos el mejor nodo de Q.

Una vez designado el nodo, vaciamos la lista de arcos fijos y recalculamos el conjunto de intercambios. Aplicamos NT … y así seguimos k veces.

EJEMPLO: Generamos un árbol Q partiendo de un nodo raíz 0.

1) Primer caso: algunos elementos de Q mejoraron al nodo inicial. Tomamos el mejor nodo, es decir el nodo 4, como nodo raíz de la próxima iteración. Referencia: en la mitad superior del círculo, el número de nodo y en la mitad inferior, el makespan.

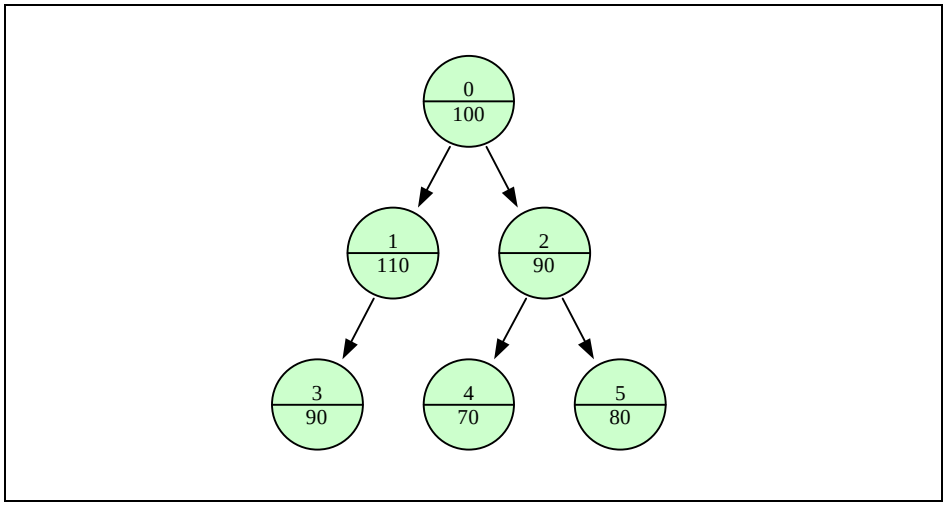


FIGURA: Primer caso.

2) Segundo caso: ningún nodo mejora al inicial. Suponemos que m=#máquinas, n=#trabajos, $\log_2$ (mn)=1 y k=3. En este caso, elegimos al azar entre los k mejores nodos que superen el nivel 1. Estos son los nodos 3, 4 y 5 a los que asignamos probabilidades K, K/2 y K/4 (K=4/7) de ser elegidos respectivamente.

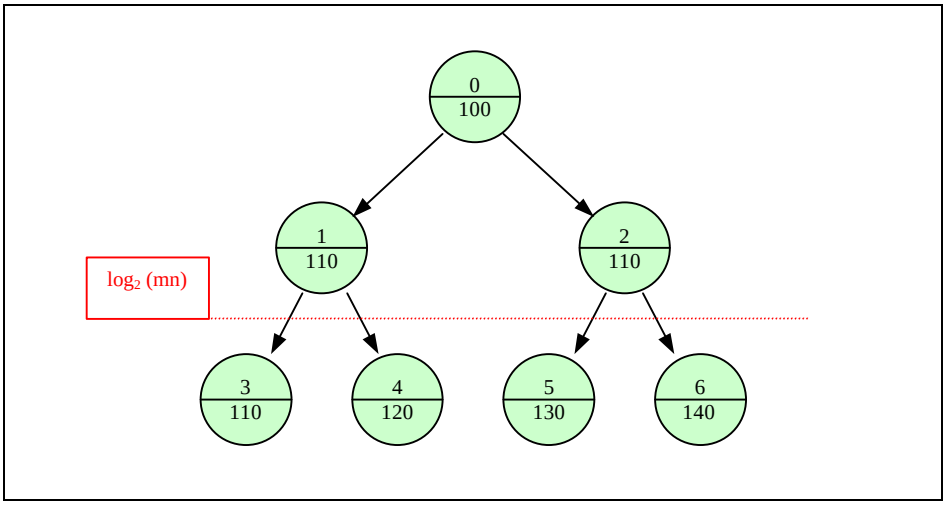


FIGURA: Segundo caso.

3) Ultimo caso: ningún nodo mejora al inicial y no hay nodos que superen el nivel 1. Tomamos el mejor nodo entre los que hubiere en el árbol, esto es el nodo1.

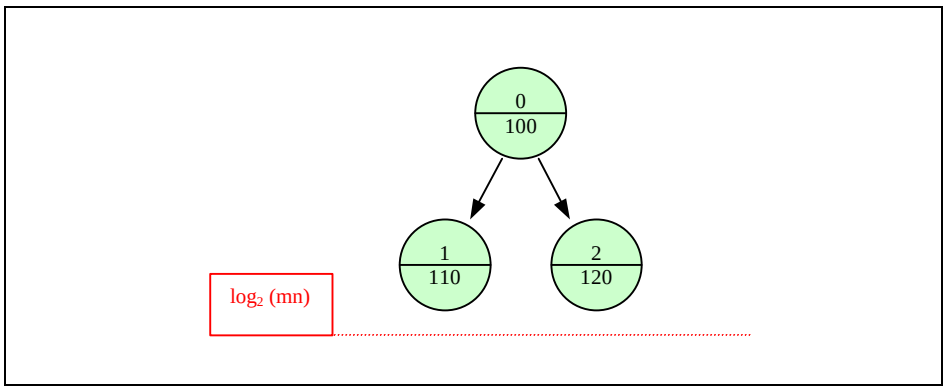


FIGURA: Tercer caso.

4.7 Shifting Bottleneck Procedure

El algoritmo SBP se basa en que entre las máquinas usualmente hay una que es más “crítica” que las otras, esto es, la manera en que se secuencia esta máquina tiene más influencia sobre el makespan que las otras. Diremos que esa máquina es el *bottleneck*.

Comenzando con ninguna máquina secuenciada, cada paso del algoritmo radica en encontrar el bottleneck entre las máquinas todavía no secuenciadas y fijar una secuencia para ese bottleneck.

Sean M el conjunto de máquinas y M_0 el conjunto de máquinas ya secuenciadas. Inicialmente $M_0=\varnothing$.

El paso genérico es como sigue. Para cada máquina $\mu \in M \backslash M_0$ aplicamos GLS (con $k=\log_2(\#J * \#M_0)$) y obtenemos una secuencia en μ optimizada, calculamos el makespan resultante C_μ y eliminamos la secuencia en μ. El bottleneck es entonces, la máquina β correspondiente al mayor C_μ.

Luego, procesamos con el GLS al bottleneck β y obtenemos una secuencia en β optimizada, incorporamos los arcos correspondientes a S, metemos β a M_0 y finalmente reprocesamos todas las máquinas de M_0.

Repitiendo este rutina hasta que $M_0=M$ y llegamos a una selección completa S que es lo mejor que el algoritmo puede ofrecer.

EJEMPLO: El problema consiste en 3 máquinas, 3 trabajos, 9 operaciones. Digamos que la máquina 1 procesa las operaciones 1,2,3 la máquina 2 las 4,5,6 y la máquina 3 las 7,8,9.

Comenzamos con un grafo G=(O,A) y $M_0 = \varnothing$, esto es, sin ningún orden prescrito dentro de cada máquina. En este caso partimos de un makespan de valor 100. Ahora buscamos el bottleneck. Tomamos la máquina 1, creamos un orden factible en esa máquina y aplicamos GLS dando por resultado, digamos, la secuencia 3 1 2 con un makespan igual a 105. No debe asombrarnos que haya crecido el makespan, ya que al secuenciar una máquina estamos introduciendo nuevas restricciones al problema. Finalmente, borramos la secuencia de la máquina 1.

Repetimos el procedimiento en las máquinas 2 y 3.

Una vez encontrado el bottleneck, en este caso la máquina 3, la optimizamos mediante GLS, la incorporamos al conjunto M_0 y nos dedicamos a reprocesar las máquinas que están en M_0 para disminuir aún más el makespan.

Ahora, indentificamos al bottleneck entre las máquinas que pertenecen a $M \setminus M_0 = \{1,2\}$, la metemos en M_0 y reprocesamos las máquinas incluidas en M_0.

Finalmente tomamos la máquina 2, la procesamos, la metemos en M_0 y reprocesamos las máquinas 1, 2, 3 de M_0.

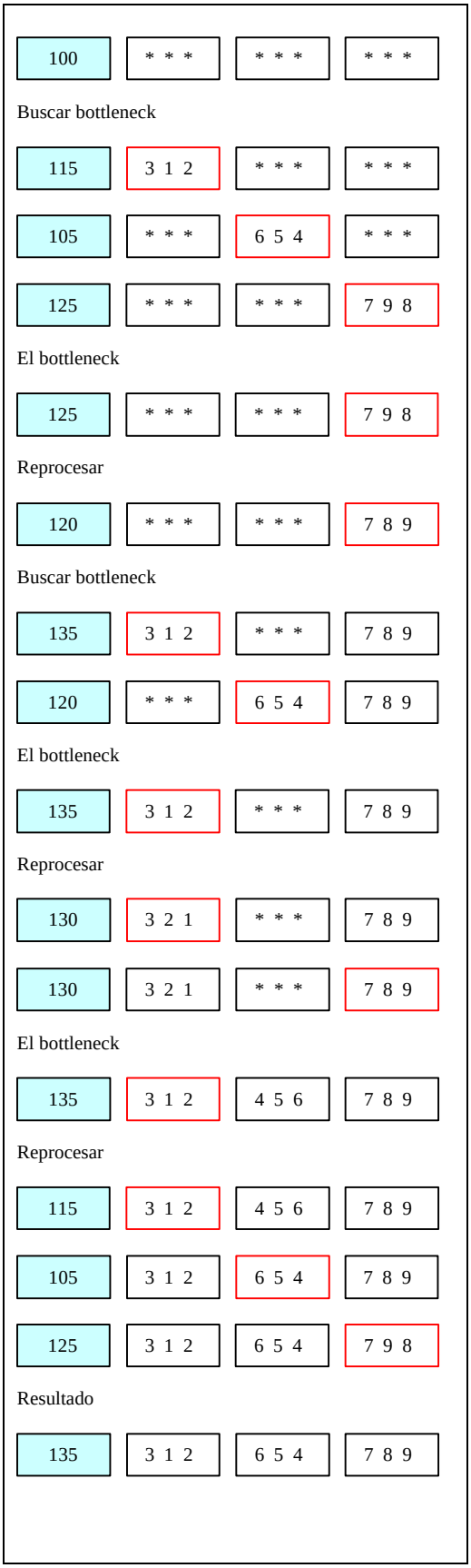


FIGURA: Esquema donde se figura la actuación del algoritmo SBP. En la primera casilla se anota el makespan, en las tres restantes las secuencias en las máquinas 1, 2 y 3 respectivamente.

4.8 Pseudocódigo de Shifting Bottleneck Procedure

Shifting Bottleneck Procedure. Optimiza heurísticamente el problema de job shop scheduling.

SBP
comienza con los conjuntos O de operaciones, J de trabajos, M de máquinas y A de arcos
retorna una selección total S heurísticamente óptima

sean una selección $S=\emptyset$, un grafo $G=(O,A\cup S)$ y un conjunto $M_0=\emptyset$
repetir mientras $M_0 \neq M$
    repetir para todo $\mu \in M\backslash M_0$
        crear secuencia factible en máquina μ aplicando CREAR SECUENCIA
        optimizar secuencia en la máquina μ aplicando GLS
        calcular C_μ el makespan asociado a $G=(O,A\cup S)$ usando LPS
        borrar de S los arcos que corresponden a la máquina μ haciendo $S_\mu=\emptyset$
    encontrar el bottleneck $\beta=\arg\max \{C_\mu : \mu \in M\backslash M_0\}$
    crear secuencia factible en máquina β aplicando CREAR SECUENCIA
    optimizar secuencia en la máquina β aplicando GLS
    meter β en M_0
    repetir para todo $\mu \in M_0$
        optimizar secuencia en la máquina μ aplicando GLS
la secuencia S es lo mejor que se encontró

Guided Local Searching. Optimiza una secuencia en una máquina dada.

GLS
comienza con una secuencia S, un grafo $G = (O,A\cup S)$ y una máquina μ
modifica la secuencia S

sean nodos T = (n,F,D,C,V,W), T' = (n',F',D',C',V',W'), un valor C^*, una secuencia W^*
hacer n=0, F=∅, D=∅, W=S
calcular L(s,·), L(·,t), makespan C y camino crítico en G = (O,A∪W) usando LPS
obtener lista de intercambios V en μ aplicando ENCONTRAR INTERCAMBIOS
    a partir de L(s,·), L(·,t), camino crítico y G = (O,A∪W)
$C^*= C$
$W^*= W$
repetir $k=\log_2 (\#M_0 * \#J)$ veces
    obtener el árbol Q usando NT sobre el nodo T
    si existe un nodo de Q que mejore T
        T'= mejor nodo de Q
    sino, si algún nodo de Q tiene nivel $n > \log_2 (\#M * \#J)$
        tomar los k mejores nodos de Q
        T'= nodo elegido al azar entre esos k
    sino
        tomar T'= mejor nodo de Q
    T=T'
    si el mejor nodo de Q mejora C^*
        C^* = makespan asociado a ese nodo
        W^* = secuencia asociada a ese nodo
$S=W^*$

Neighborhood Trees. Crea un árbol cuyos nodos son secuencias factibles.

NT
comienza con un nodo raíz T = (n,F,D,V,C,W)
retorna un árbol Q

sea el árbol Q={T}, marcar T como no revisado
repetir mientras existan nodos no revisados en Q
    tomar T∈ Q nodo no revisado
    repetir para todo (u,v) ∈ V
        sacar (u,v) de V
        aplicar PROHIBIDO para saber si (u,v) está prohibido por F o por guideposts
        si (u,v) no está prohibido
            n'=n+1
            F'=F
            meter (u,v) en F
            meter (v,u) en F'
            obtener selección W' efectuando el intercambio (u,v) sobre W
                aplicando HACER INTERCAMBIO
            calcular L'(s,·), L'(·,t), makespan C' y camino crítico en
                G' = (O,A∪W') usando LPS,
            obtener lista de intercambios V' en μ aplicando
                ENCONTRAR INTERCAMBIOS a partir de los valores
                L'(s,·), L'(·,t), camino crítico y G = (O,A∪W')
            si C'>C
                hallar D' aplicando ENCONTRAR GUIDEPOST
            meter nodo T'=(n',F',D',V',C',W') en Q y marcarlo como no
                revisado
    marcar T como revisado
retornar árbol Q

Encuentra pares críticos para realizar intercambios.

ENCONTRAR INTERCAMBIOS
comienza con una secuencia W, valores L(s,·), L(·,t) y camino crítico calculados en el grafo
    G = (O,A∪W)
retorna lista de intercambios V

sea un conjunto V=∅
repetir para todo (u,v) par de operaciones
    si se cumplen
        i) u, v son operaciones que corresponden a la máquina μ
        ii) u está antes que v según W
        iii) u, v pertenecen al camino crítico
        iv) si w pertenece al camino crítico y está ubicado entre u y v, entonces w
            corresponde a la máquina μ
    y vale por lo menos una de las siguientes
            v) γ(v) pertenece al camino crítico y L(v,t) ≥ L(γ(u),t)
            vi) α(u) pertenece al camino crítico y $L(s,u) + p_u \geq L(s,\alpha(v)) + p_{\alpha(v)}$
    entonces
        incorporar (u,v) a V
        si vale v)
            marcar (u,v) como forward iinterchange
        si vale vi)
            marcar (u,v) como backward interchange
retornar V

Dada una selección devuelve otra que resulta de realizar un intercambio sobre la primera.

HACER INTERCAMBIO
comienza con una selección W y un par crítico (u,v)
retorna una selección W'

sea un conjunto W'=W
si (u,v) marcado como forward interchange
    cambiar en W' el arco (u,v) por (v,u)
    repetir para todo w operación de μ ,
        si w entre u y v según W
            cambiar en W' el arco (u,w) por (w,u)
si (u,v) marcado como backward interchange
    cambiar en W' el arco (u,v) por (v,u)
    repetir para todo w operación de μ
        si w entre u y v según W
            cambiar en W' el arco (w,v) por (v,w)
retornar W'

Analiza si un intercambio sobre un par crítico dado está prohibido ya sea por arcos considerados fijos o bien por restricciones debidas a un guidepost.

PROHIBIDO
comienza con un par crítico (u,v), una lista de arcos fijos F, un guidepost D y una secuencia W
retorna un mensaje

repetir para todo $(a,b) \in F$
    si a=u y b=v
        retornar "prohibido"
    sino si (u,v) es un forward interchange, a=u y b es posterior a v según W
        retornar "prohibido"
    sino si (a,b) es un backward interchange, b=v y a es anterior a u según W
        retornar "prohibido"
si alcanzamos un left guidepost , esto es $D=\{(s,q)\}\neq\emptyset$, y u después que q o v después que q según W
    retornar "prohibido"
sino, si alcanzamos un right guidepost, esto es $D=\{(q,t)\}\neq\emptyset$, y u antes que q o v antes que q según W
    retornar "prohibido"
sino
    retornar "no prohibido"

Dadas una selección y una máquina, crea una secuencia factible en la máquina indicada y la incorpora a la selección.

CREAR SECUENCIA
comienza con una secuencia W, un grafo $G=(O,A\cup W)$ y una máquina μ
modifica la secuencia W

sea $K=\{u_1,\ldots,u_n\}$ una secuencia cualquiera para las operaciones correspondientes a la máquina μ
calcular $L(s,\cdot)$ en el grafo $G=(O,A\cup W)$ mediante LPS
reordenar K de tal manera que si i antes que j según K entonces $L(s,i) \leq L(s,j)$
incorporar a W los arcos { (i,j) : i, j operaciones distintas que corresponden a μ, i antes que j según la secuencia K }

Analiza la existencia de un guidepost.

ENCONTRAR GUIDEPOST

comienza con un nodo T con valores L(s,·), L(·,t) en el grafo G = (O,A∪W) y un nodo T' con
    valores L'(s,·), L'(·,t) en el grafo G' = (O,A∪W')
retorna el conjunto D

sea D un conjunto
considerar
    i) (u,v) forward interchange y $L(v,t) > L'(v,t) + p_u$
    ii) (u,v) backward interchange y $L(u,t) > L'(u.t) - p_v$
    iii) (u,v) forward interchange y $L(s,v) > L'(s,v) - p_u$
    iv) (u,v) backward interchange y $L(s,u) > L'(s,u) + p_v$
si C'>C y además valen i) o ii) y no valen iii) ni iv)
    alcanzamos un right guidepost, D={(v,t)}
sino, si C'>C y además valen iii) o iv) y no valen i) ni ii)
    alcanzamos un left guidepost, D={(s,v)}
sino
    D=∅
retornar D

Dado un grafo encuentra makespan, camino crítico, caminos de máxima longitud entre el vértice inicial s y un vértice cualquiera y caminos de máxima longitud entre el vértice final t y un vértice cualquiera.

LPS

comenzar con un grafo G = (O,R) tal que toda rama (u,v)∈R tiene un peso asignado p_v

para cada u∈O sea ε(u) un entero
iniciamos para todo u∈O
    L(s,u)=0, L(u,t)=0, ε(u)=−1
repetir #O−1 veces
    repetir para toda rama (u,v)∈R
        si $L(s,v) < L(s,u) + p_v$
            $L(s,v) = L(s,u) + p_v$ y ε(v)=u
        si $L(u,t) < L(v,t) + p_v$
            $L(u,t) = L(v,t) + p_v$
el makespan es C=L(s,t) y los valores de ε(u) indican cómo recorrer el camino óptimo de atrás
    para adelante comenzando por t
retornar C, camino crítico y valores L(s,·), L(·,t)

Capítulo V
Estadísticas

5.1 Introducción

Este capítulo está dedicado a observar el comportamiento de los dos algoritmos anteriormente explicados, el TSAB y el SBP. Estos dos, dependen de algunos parámetros fijados por el usuario para optimizar su desempeño, por lo tanto, exploramos esta relación sucintamente tomando algunos valores con carácter estadístico. Ambos algoritmos se codificaron en el lenguaje C^{++} y se corrieron en una PC pentium

5.2 TSAB

Hemos descrito el algoritmo de TSAB de Nowicki-Smutnicki. Ahora, para no escapar a su destino empírico, veremos cómo se comporta el algoritmo cuando se enfrenta al cómputo de un problema concreto.

Primeramente, debemos mencionar un punto que no ha sido anteriormente tratado: los valores numéricos correspondientes a la longitud de la lista taboo, la longitud de la lista de soluciones guardadas para efectuar back jump tracking y la cantidad máxima permitida de iteraciones sin mejorar (los indicaremos respectivamente *maxt, maxiter*, *maxl*) los cuales son parámetros del algoritmo que no están determinados.

En efecto, no hay disponibles hasta el momento en la bibliografía valores que se puedan decir óptimos, más bien, podemos hablar de valores estadísticamente recomendables para un tamaño específico de problema.

Hemos simulado aleatoriamente un escenario posible y asignado valores arbitrarios para los parámetros libres del TSA. La simulación de un problema es llevada a cabo por un algoritmo a tal efecto que, partiendo de un número de trabajos y máquinas dados por el usuario, fija para cada trabajo una cantidad aleatoria de operaciones y para cada una de estas últimas asigna también aleatoriamente una máquina y un tiempo de demora.

Presentamos a continuación un seguramente esperado perfil de búsqueda del algoritmo. En el primer gráfico se ilustra la evolución del makespan C en el transcurso de las iteraciones, es decir, a lo largo de la sucesión de soluciones que visita el algoritmo. En el segundo gráfico se muestra paralelamente la evolución del mejor makespan conocido C*.

El escenario concreto en que se realizó esta prueba se dispone como sigue:

Tamaño del problema (trabajos × máquinas)	20 x 10
Cantidad de operaciones	173
Longitud lista tabú	10
Longitud lista back jump	5
Máximo iteraciones sin mejora	50

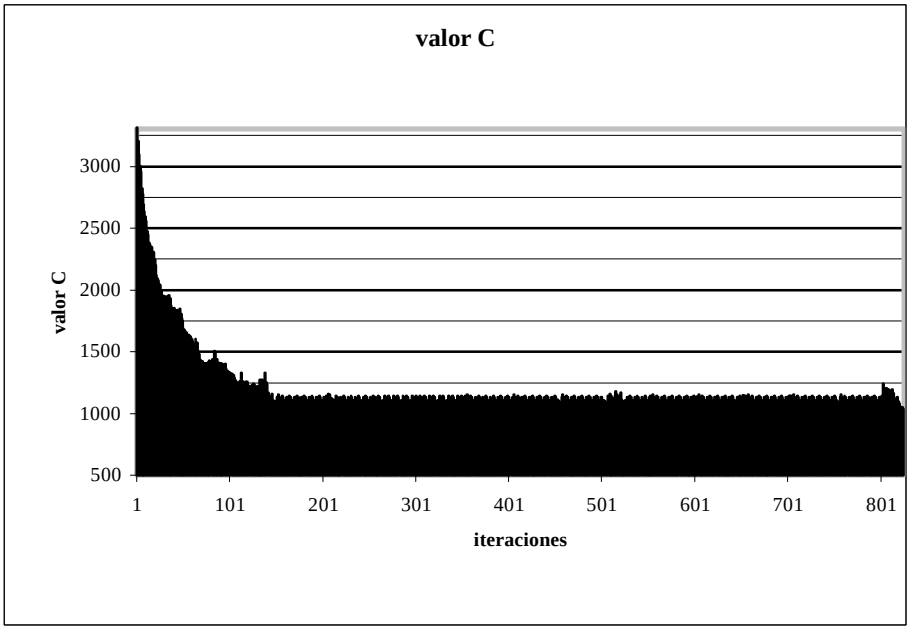


FIGURA: Perfil del recorrido del TSA por el conjunto de soluciones. Se representa el makespan de cada solución en cada iteración.

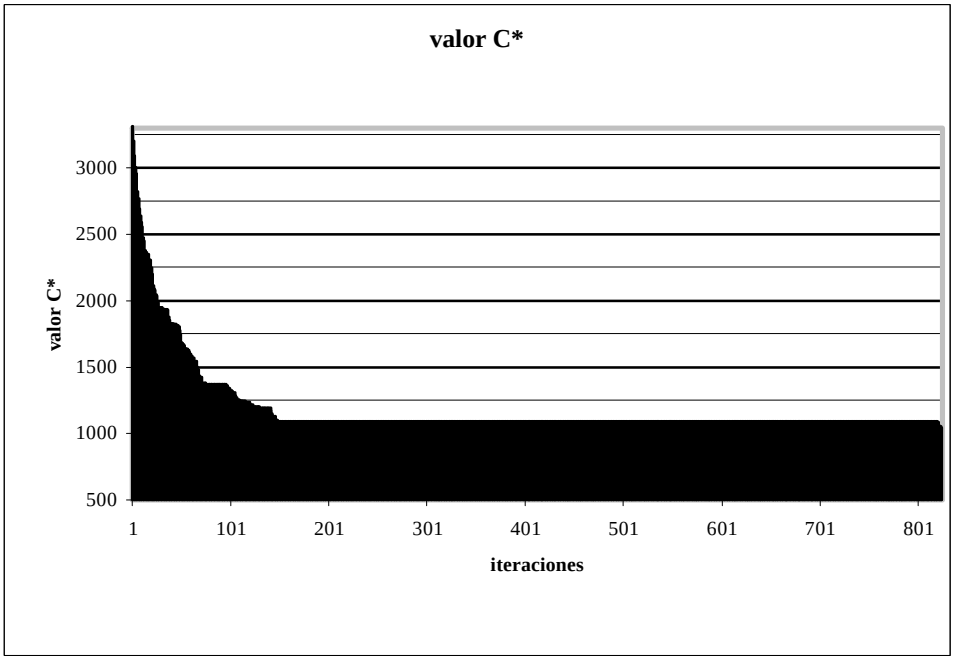


FIGURA: Perfil del recorrido del TSA por el conjunto de soluciones. Se representa el makespan de la mejor solución conocida en cada iteración.

El algoritmo se desempeñó como sigue:

Solución inicial	3315
Mejor solución encontrada	1044 (óptima)
Cantidad de iteraciones	825

Vemos que el TSA alcanzó el valor óptimo de C=1044 cuyo valor y carácter de tal podemos asegurar por coincidir con la duración de uno de los trabajos. En efecto, finalizar todos los trabajos demora ineludiblemente más tiempo que acabar uno solo.

Ahora, aprovechamos para mostrar el modus operandi del TSA más detalladamente y para eso tomamos un pequeño problema de un poco más de una docena de operaciones y aprovechamos un recurso de gran poder que ya hemos usado con anterioridad, el diagrama de Gantt. A través de él veremos cómo actúa el TSA y cómo va modificando las secuencias en cada una de las máquinas para disminuir el makespan. No dejamos constancia de los moves involucrados e invitamos al lector a descubrirlos.

Hemos fijado los siguientes parámetros:

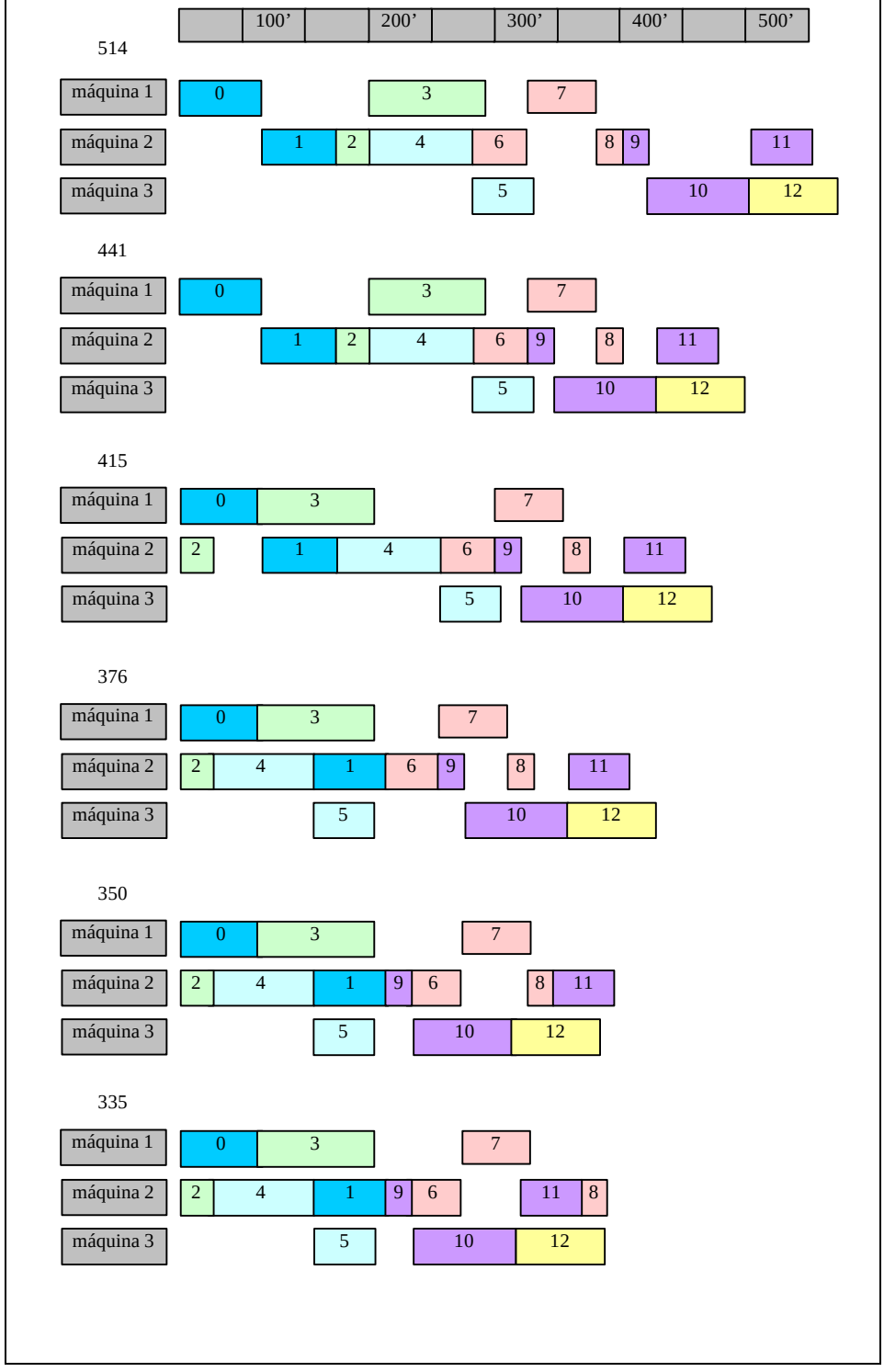
Longitud lista tabú	10
Longitud lista back jump	5
Máximo iteraciones sin mejora	50

y el problema en cuestión es como sigue:

Tamaño del problema (trabajos × máquinas)	6 x 3
Cantidad de operaciones	13

operación	0	1	2	3	4	5	6	7	8	9	10	11	12
trabajo	0	0	1	1	2	2	3	3	3	4	4	4	5
máquina	0	1	1	0	1	2	1	0	1	1	2	1	2
tiempo	65’	58’	26’	91’	86’	45’	41’	55’	18’	17’	78’	49’	70’

TABLA: Descripción de las características del problema seleccionado.


100’
200’
300’
400’
500’
514
máquina 1
máquina 2
máquina 3
441
415
376
350
335

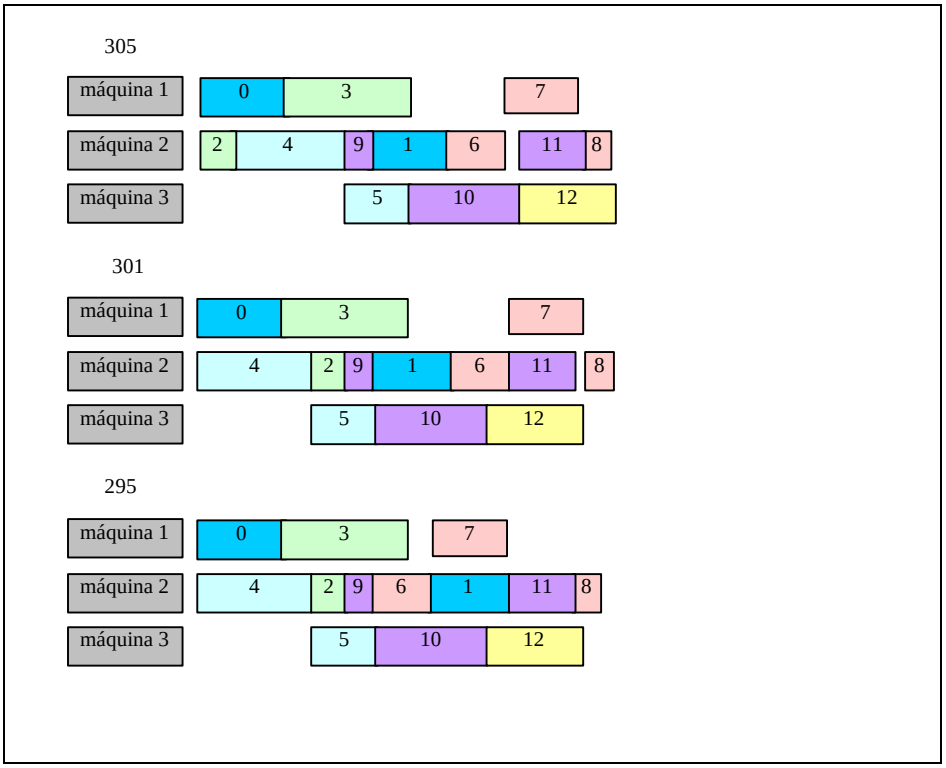


FIGURA: Evolución del makespan al que obliga el TSA, descrito en diagramas de Gantt.

Los resultados fueron satisfactorios. Además, hemos de notar que el algoritmo se detiene sobre un óptimo (una de las máquinas queda saturada).

Solución inicial	514
Mejor solución encontrada	295 (óptima)
Cantidad de iteraciones	8

5.2.1 Los parámetros

Como vimos, el TSAB tiene tres parámetros no determinados que especifica el usuario:

- *máxima longitud de lista tabú (maxt)*

- *máxima longitud de lista de back jump (maxl)*

- *cantidad de iteraciones sin mejorar (maxiter)*

Como era de esperar, la asignación de tales o cuales valores para cada uno de esos parámentros influyen sobre la eficiencia del algoritmo, tanto en lo que respecta a la calidad de la solución final hallada como en el número de iteraciones que demora en concluir.

En nuestra experiencia hemos podido observar algunos hechos que se repiten regularmente los cuales según creemos constituyen tendencias o inclinaciones antes que directrices rigurosas y presentamos a continuación como indicaciones a la hora de elegir una configuración determinada de valores para los parámetros.

- No hay una configuración de los parámetros que sea óptima para cualquier problema cualquiera sea su dimensión.

- Valores pequeños de maxt causan un vagabundeo errático por el conjunto de soluciones, aumento de la dispersión en la calidad de las soluciones proporcionadas, incremento en el número de iteraciones y aumenta las probabilidades de caer y quedar atrapado en un óptimo local.

- Valores grandes de maxt provocan fenómenos inversos a los expuestos en el punto anterior, esto es, un recorrido por el conjunto de soluciones guiado por una política más conservadora,..., etc.

- Incrementando maxt observamos lo siguiente. Inicialmente, el algoritmo queda atrapado en óptimos locales y brinda entonces soluciones de exigua calidad en pocas iteraciones. Luego, las soluciones encontradas son cada vez mejores pero el costo va en incremento. Finalmente, se alcanza una meseta, un estado estacionario con muy poca variación tanto en calidad como en costo.

- El valor de maxiter parece reflejar el compromiso adoptado entre calidad de la solución final y cantidad de iteraciones. Cuanto más grande es maxiter, más demora el algoritmo en hallar una solución pero, por lo general, mejor es ésta.

- Mientras que no encontramos una relación funcional directa entre maxiter y el makespan de la solución hallada, sí puede verse que esta relación existe y es lineal entre este parámetro y el número total de iteraciones a condición de mantener fijo el valor de maxt.

Se han probado diferentes valores de longitud de la lista tabú y cantidad de iteraciones sin mejorar para estudiar el comportamiento del TSAB. Para más detalle, la situación es como sigue:

Tamaño del problema (trabajos × máquinas)	20 x 10
Longitud lista tabú	5, 10, 20, 30, 40, 50
Longitud lista back jump	5
Máximo iteraciones sin mejora	10, 50, 100, 200

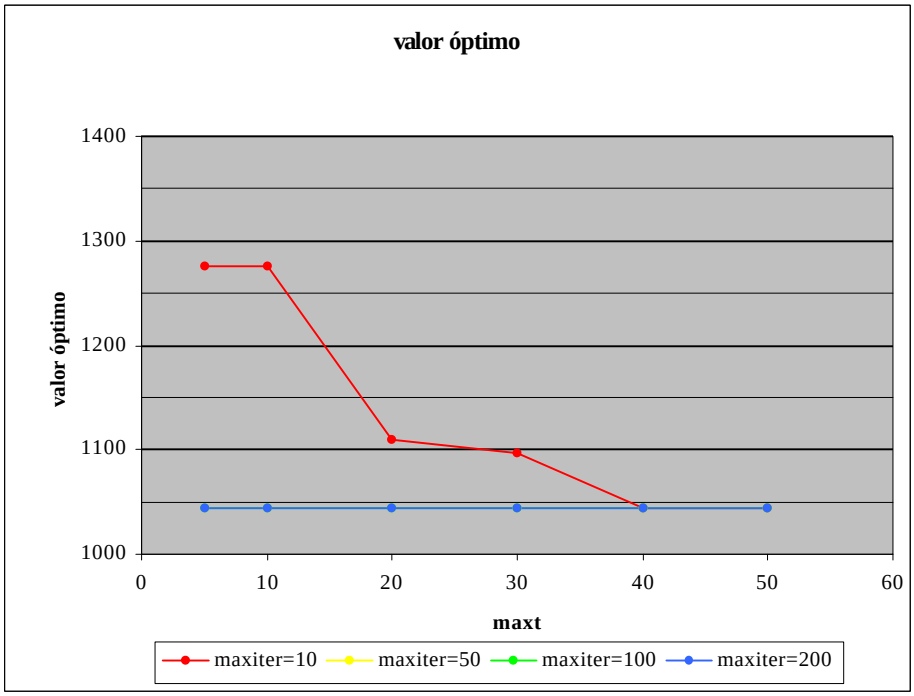


FIGURA: Se grafica para varios valores de maxiter la calidad de la solución óptima hallada versus la longitud lista tabú.

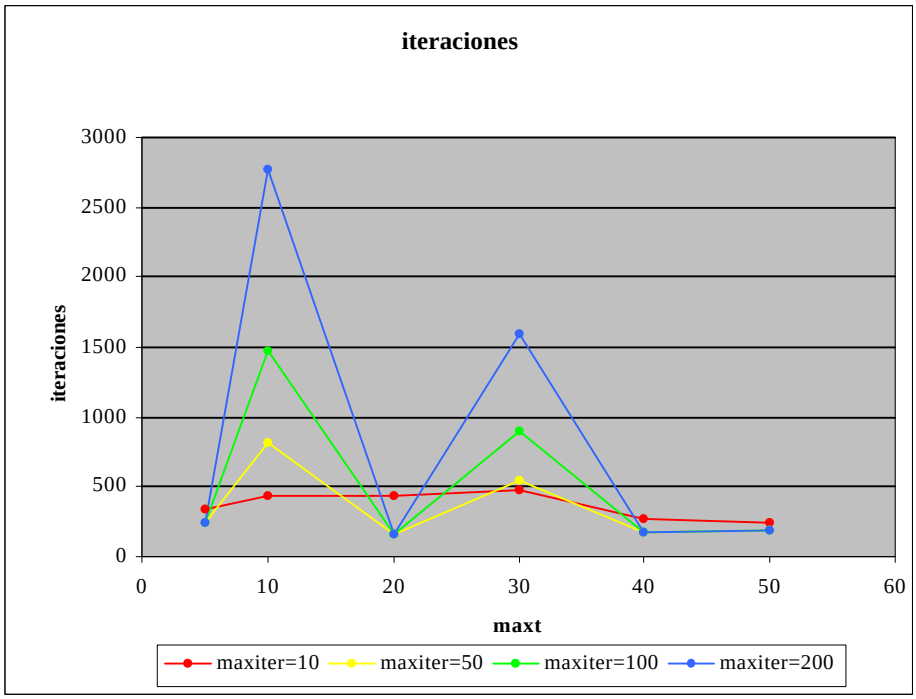


FIGURA: Se grafica para varios valores de maxiter la cantidad de iteraciones que insume el TSA versus la longitud de lista tabú.

5.3 SBP

A continuación mostramos el tratamiento con el SBP del mismo problema anterior. El eje horizontal representa la sucesión de nodos que se van generando mediante el GLS y el eje vertical representa el valor del makespan de la selección asociada a cada nodo. Hemos distinguido en el dibujo dos fases que se van alternando conforme avanza el proceso realizado por el algoritmo, una, se nos ofrece como la búsqueda de la máquina crítica la cual hemos dado en llamar bottleneck, la otra, se limita a tratar esta máquina en particular e insertarla entre otras máquinas ya secuenciadas. En el segundo gráfico, el eje vertical representa el mejor valor actual de entre los nodos de una misma máquina.

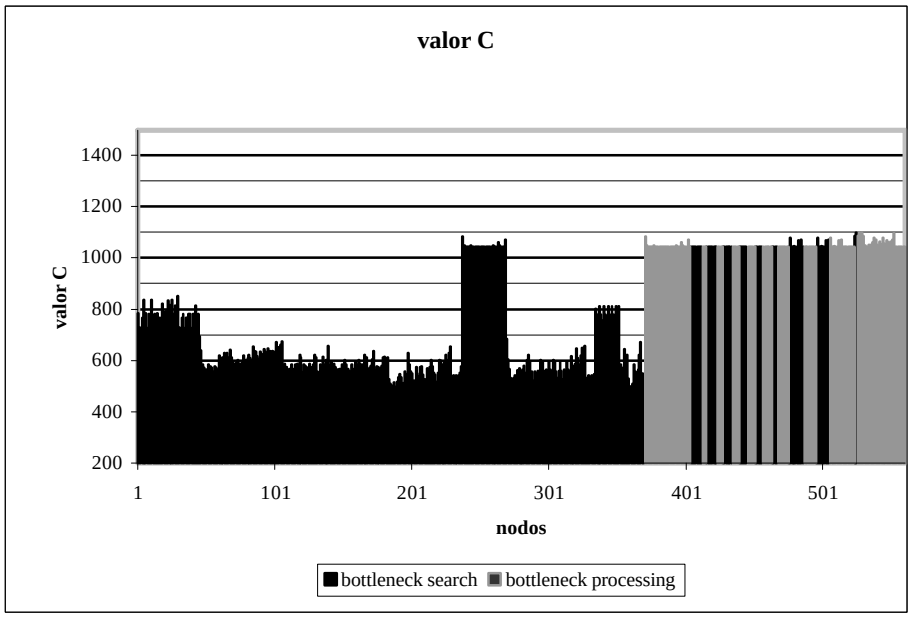


FIGURA: Perfil del recorrido del SBP sobre el conjunto de soluciones. Se grafica el makespan de la solución conforme se van creando nodos. Se distingue entre las tareas de identificación del bottleneck y su procesamiento.

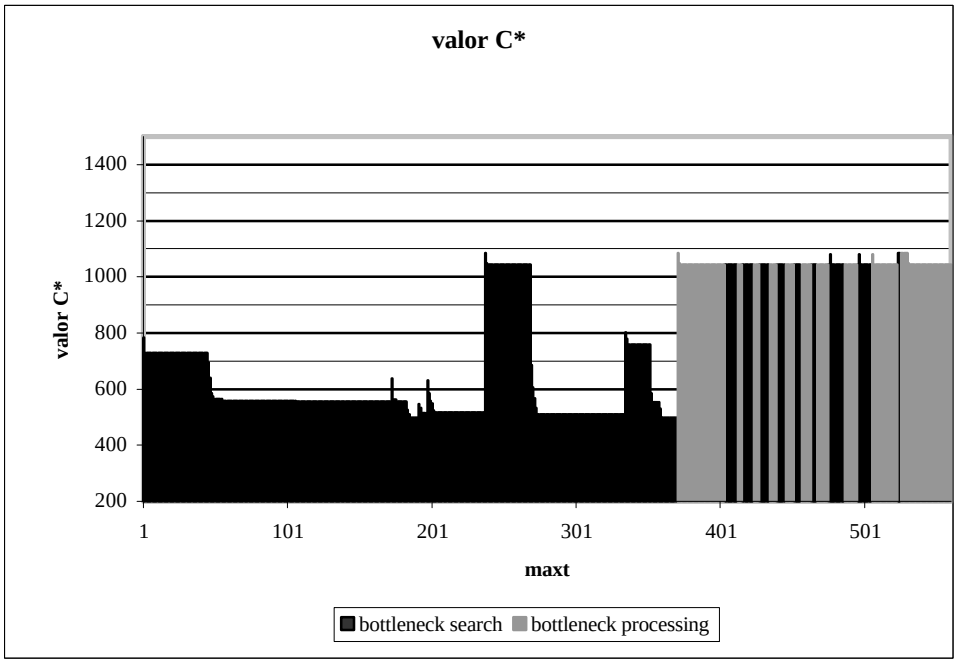


FIGURA: Perfil del recorrido del SBP sobre el conjunto de soluciones. Se grafica el makespan de la mejor solución conocida conforme se van creando nodos. Se distingue entre las tareas de identificación del bottleneck y su procesamiento.

5.3.1 Parámetros del SBP

También en el SBP existen ciertos valores que son preseteados por el usuario de acuerdo, al menos hasta donde nosotros conocemos, a su propio criterio y según sus necesidades particulares. A continuación presentamos una pequeña lista de estos, que si bien no es exhaustiva, da una idea de su naturaleza.

- *número máximo de hijos para un nodo de un nivel determinado.* Esto es, un límite para la cantidad de hijos que un nodo en un nivel dada puede tener y responde a la necesidad de acotar el número de elementos que genera el GLS cuando construye un árbol. Nosotros, por lo general, ajustamos el máximo número de hijos por una función lineal decreciente con el índice de nivel en el cual el nodo se halla.

- *número de máximo de árboles que genera el GLS.* Nosotros hemos adoptado el valor $k=\log_2 (\#M_0 * \#J)$ según recomendación de Balas-Vazacopoulos.

- *nivel crítico.* Es el nivel a partir del cual comenzamos a guardar las mejores soluciones del árbol. Nosotros utilizamos el valor $k=\log_2 (\#M * \#J)$ de B-V.

- *número máximo de soluciones buenas a guardar.* Instrumentamos $k=\log_2 (\#M_0 * \#J)$ de B-V.

Pero además de las variaciones introducidas por la asignación de uno u otro valor a cada parámetro, hay disponibles diversas versiones del SB producidas por los mismos autores. En efecto, la versión más conocida, la cual hemos mostrado en el capítulo precedente y notada SB-GLS/1 se tornó una base para nuevos desarrollos que mantienen la idea original de trabajar ignorando algunas máquinas y reducir un problema de n máquinas a uno de k máquinas, con k menor que n por supuesto.

Uno de estos, es el SB-GLS/2 que consiste en partir de todas las máquinas secuenciadas y a cada paso tomar una máquina por vez, remover los arcos correspondientes a su secuencia, optimizar las n−1 máquinas restantes aplicando GLS y luego secuenciar la máquina removida y aplicar GLS a la totalidad de las máquinas.

Otro es el SB-RGLSk que descansa en repetir k veces lo siguiente: una vez secuenciadas todas las máquinas a través del SB-GLS/1 se remueven $[\ n^{1/2}\ ]$ máquinas seleccionadas al azar, de tal manera que la i-ésima máquina secuenciada tenga probabilidad $K(2/3)^i$, se aplica GLS sobre las restantes y se completa el schedule aplicando SB-GLS/1.

Evidentemente, no hay límite para la cantidad de posibles recombinaciones más que la imaginación. Pero, como es imperativo poner un alto, nos detendremos aquí y únicamente indicaremos los resultados de los dos primeros procedimientos.

5.4 Comparación con otros algoritmos

Mostramos algunos resultados obtenidos por otros autores sobre la eficiencia del TSA, el SBP y otros varios algoritmos, que incluimos como referentes para tener un marco comparativo. Hemos hecho una selección de estos algoritmos a nuestro criterio y hemos privilegiando los que son ampliamente conocidos por su performance y por su relevancia histórica y los hemos indicado según la tradición vigente en la literatura de investigación operativa.

Estos algoritmos son:

- (AC) shuffle approximation procedure de Applegate-Cook de 1991
- (DP) genetic algorithm de Dorndorf y Pesch de 1995
- (LAL) simulated annealing de Laarhoven de 1992
- (BC) taboo search de Barnes-Chambers de 1995
- (DT) taboo search de Dell'Amico-Trubian de 1993
- (NS) taboo search de Nowicki-Smutnicki de 1993
- (SB) shifting bottleneck procedure de Balas-Vazacopoulos de 1998

El primer análisis se refiere a la calidad de la solución encontrada. Para esto hemos tomado una decena de problemas y hemos configurado un cuadro en el cual exhibimos el error relativo de los algoritmos al intentar alcanzar la mejor solución encontrada.

	1	2	3	4	5	6	7	8	9	10
AC	0.8	0	0	2.9	1.8	0.2	1.2	0	1.4	0.2
DP	0.8	0.1	0.1	1.5	0.9	1.1	0.2	0.2	1.7	0.7
LAL	2.3	0	0.7	1.2	0.3	0.2	2.1	1.5	3.0	0.2
BC	0.5	0	0.1	0.9	0.4	0.5	0.3	0.2	0	0.4
DT	0.5	0	0	0.4	0.3	0.2	1.6	0.3	0	0.4
NS	0	0	0	0.2	0	0.1	0	0.2	0.2	0
SB-GLS/1	0	0.1	1.1	0.4	0	0.1	0.5	0	0.2	0.1
SB-GLS/2	0	0	0	0	0.1	0	0.2	0	0.1	0

TABLA: Calidad (porcentual) de soluciones encontradas por diversos algoritmos.

Ahora, vamos a mostrar el costo computacional de hallar esas mismas soluciones en términos de CI-CPU time (computer-independent CPU time, Dongarra y Vaessens, 1994). Así, hemos de observar la relación costo-beneficio que reporta cada algoritmo.

	1	2	3	4	5	6	7	8	9	10
AC	61	19	182	720	1315	5853	3915	$>10^5$	$>10^5$	3734
DP	171	26	124	321	327	201	561	537	458	1653
LAL	740	111	788	525	4187	2312	5023	5206	6123	5112
BC	190	347	960	1019	1525	3022	2784	2171	1115	3689
DT	390	47	260	59	498	596	633	642	314	161
NS	930	655	842	1200	2063	3112	2871	3200	2033	3513
SB-GLS/1	32	666	29	112	260	97	140	270	151	163
SB-GLS/2	113	655	184	514	1032	366	793	1083	831	914

TABLA: Tiempo de ejecución de por diversos algoritmos medidos en CI-CPU.

5.5 Perspectivas futuras

Como nota final trataremos algunas posibilidades de optimizar estos algoritmos a través de la sofisticación de las técnicas que utilizan subyacentemente ambos.

El cálculo del camino crítico y del makespan es un punto muy importante en cuanto a su influencia sobre la eficiencia del procedimiento utilizado para tratar el problema. En este texto hemos elegido una modificación del algoritmo de Ford, que es un O(#V #R), pero existen muchos otros con rendimientos superiores. Estos se aprovechan de la estructura particular del job shop para hacer un cómputo eficiente. Por ejemplo, es común el cálculo de camino crítico y makespan de todas las soluciones de un entorno y se aprovecha esto observando que la realización de un move o intercambio involucra la modificación de muy pocas ramas en relación al resto, entonces si π' pertenece al entorno de π

- Muchos caminos de $G(\pi)$ permanecen inalterados en $G(\pi')$ y no hay necesidad de computarlos nuevamente.

- Si ordenamos topológicamente los vértices del grafo $G(\pi)$ con el consiguiente costo, podemos obtener fácilmente un ordenamiento de la misma característica en $G(\pi')$ sin necesidad del mismo costoso procedimiento que para $G(\pi)$. La utilidad de esto se descubre cuando advertimos que algoritmos muy eficientes para el cálculo de caminos óptimos exigen de orden topológico.

Una instancia casi obligada por la asiduidad del recurso, es la presencia de algoritmos que aproximan el makespan y el camino crítico con la finalidad de ganar en velocidad. También hay algoritmos que calculan tan sólo una cota inferior del makespan para las soluciones de un entorno (Taillard, 1989) realizando una política de *branch and bound*.

Otro punto donde se han introducido mejoras, específicamente en el ámbito del TSA es en determinar el largo adecuado de la lista tabú.

- En un principio, se fijaba una longitud determinada standard para todo tipo de problemas.

- El siguiente paso fue asignar una longitud para cada problema en particular, la cual se calcula antes de la ejecución del algoritmo.

- Luego Dell'Amico y Trubian (1993) desarrollaron una estrategia que permite un ajuste dinámico de la longitud de la lista durante el proceso de búsqueda adaptándose de acuerdo a las condiciones puntuales. El largo de la lista varía, con límites superior e inferior, decreciendo en respuesta a moves provechosos y creciendo en respuesta a moves no provechosos. Esto obedece a la idea de focalizar la búsqueda en áreas donde la presencia de buenas soluciones es abundante.

5.6 Nota final

La idea original que nos guió a lo largo de este texto era dar una perspectiva global de uno de los problemas NP-hard de la optimización combinatoria y a través de él observar algunas técnicas y estrategias típicas que sazonan este campo. Resulta sorprendente, el juego constante de simpleza y complejidad que se desprende tanto de problemas como de algoritmos, a veces solapado, a veces manifiesto.

Hemos de detenernos en algún punto y lo haremos precisamente aquí.

Bibliografía

Eugeniuz Nowicki and Czeslaw Smutnicki. A fast taboo search algorithm for the job shop problem. Management science, vol 42, num 6, jun 1996, págs 797-813.

Egon Balas and Alkis Vazacopoulos. Guided local search with shifting bottleneck for job shop scheduling. Management science, vol 44, num 42, feb 1998, págs 262-275.

Emile Aarts and Jan Karel Lenstra. Local search in combinatorial optimization. Wiley & Sons, 1997.

Ferdinando Pezzella ed Emanuela Merelli. A tabu search method guided by shifting bottleneck for the job shop scheduling problem. Paper de publicación electrónica , Istituto di Informatica, Università degli Studi di Ancona (Italia), 1998.

John Chambers and Wesley Barnes. New tabu search results for the job shop scheduling problem. Paper de publicación electrónica, Department of computer science, University of Austin (USA), 1996.

A Agnetis. Introduzione ai problemi di scheduling. Paper de publicación electrónica, Dipartimento di Ingegneria dell'Informazione, Università di Siena (Italia), 2000.

Brian Kernighan and Dennis Ritchie. El lenguaje de programación C. Prentice Hall.

Javier García de Jalón y otros. Aprenda C++. Publicación electrónica, Escuela superior de ingenieros industriales, Universidad de Navarra (España), 1998.

Dos ratones y un minino: