Búsqueda de local en la programación de *flowshop*

Universidad de los Andes
Isabella Bermúdez Gutiérrez
201923469
i.bermudez@uniandes.edu.co

Resumen

Este documento pretende aplicar dos búsquedas locales al problema de programación de flowshop permutado. Esta aplicación consiste en la implementación, experimentación, comparación y análisis de resultados obtenidos en cada una de las búsquedas. Al final se analizan distintos puntos de mejora que se podrían implementar en futuras ocasiones para mejorar las soluciones de las heurísticas planteadas.

I. INTRODUCCIÓN

En la programación *flowshop* todos los trabajos siguen una misma secuencia en las máquinas, pero cada uno maneja un diferente tiempo de procesamiento en cada una de ellas. El problema consiste en definir el orden (rango, prioridad, etc.) de un conjunto de trabajos a medida que avanzan de una máquina a otra. Por lo tanto, el problema a resolver implica la determinación de la posición relativa del trabajo i con respecto a todos los demás trabajos. Estos tipos de problemas ocurren en muchos contextos diferentes, pues existen siempre que se puede elegir el orden en el que se pueden realizar una serie de tareas (trabajos). En general, este tipo de problemas busca minimizar el tiempo de finalización del último trabajo, también denominado *makespan* (C_{max}). En este sentido, la pregunta a resolver es: ¿Cuál es el orden en el que las tareas deben ser programadas para minimizar el C_{max}? Con esto en mente, este artículo pretende probar dos búsquedas locales en múltiples instancias de este problema con m máquinas y n trabajos. Específicamente, se utilizarán las diez instancias de gran porte planteadas por Taillard (1993) (tai500_20_1-tai500_20_10, 500 trabajos y 20 máquinas). Al final, los resultados serán analizados y comparados entre sí para identificar mejoras aplicables a futuros experimentos.

II. DESCRIPCIÓN DEL PROBLEMA

Formalmente, el problema *flowshop* puede definirse de la siguiente manera. Sea un conjunto de n trabajos y un conjunto de m máquinas, se tienen los tiempos de procesamiento de cada trabajo i en cada máquina j, denominado como t_{ij}. Cada trabajo i sigue la misma secuencia en las m máquinas. En este sentido, se busca la secuencia que minimice el máximo valor de los tiempos de terminación de los trabajos. La formulación del problema se formaliza en las Ecuaciones 1-8.

Conjuntos

$M = máquinas = \{1,2,\dots,m\}$
$N = trabajos\ (posiciones\ de\ procesamiento) = \{1,2,\dots,n\}$

Parámetros

$$t_{ij} = tiempo\ de\ procesamiento\ del\ trabajo\ i \in N\ en\ la\ máquina\ j \in M.$$

Variables de decisión

$$x_{ik} = \begin{cases} 1, si\ el\ trabajo\ i \in N\ está\ asignado\ a\ la\ posición\ k \in N. \\ 0, dlc. \end{cases}$$

$$s_{kj} = tiempo\ en\ el\ que\ un\ trabajo\ en\ la\ posición\ k \in N\ comienza\ su\ procesamiento\ en\ la\ máquina\ j \in M.$$

$$\min \quad s_{nm} + \sum_{i=1}^{N} t_{im} x_{in} \tag{1}$$

$$s.a., \quad \sum_{i=1}^{n} x_{ik} = 1 \quad \forall\ k \in N \tag{2}$$

$$\sum_{k=1}^{n} x_{ik} = 1 \quad \forall\ i \in N \tag{3}$$

$$s_{k+1,1} = s_{k1} + \sum_{i=1}^{N} t_{i1} x_{ik} \quad \forall\ k \in N\ |\ k \leq N-1 \tag{4}$$

$$s_{1,j+1} = s_{1j} + \sum_{i=1}^{N} t_{ij} x_{i1} \quad \forall\ j \in M\ |\ j \leq M-1 \tag{5}$$

$$s_{11} = 0 \tag{6}$$

$$s_{k,j+1} \geq s_{kj} + \sum_{i=1}^{N} t_{ij} x_{ik} \quad \forall\ j \in M\ |\ j \leq M-1, k \in N\ |\ k \geq 2 \tag{7}$$

$$s_{k+1,j} \geq s_{kj} + \sum_{i=1}^{N} t_{ij} x_{ik} \quad \forall\ k \in N\ |\ k \leq N-1, j \in N\ |\ j \geq 2 \tag{8}$$

La función objetivo (1) minimiza el tiempo de finalización del último trabajo. Es decir, minimiza el C_{max}. La restricción (2) garantiza que solo haya un trabajo asignado a cada posición. La restricción (3) garantiza que cada trabajo tenga asignada una única posición. Las restricciones (4), (5) y (6) aseguran que no hay tiempo de espera en la máquina 1 ni en el trabajo 1. La restricción (7) asegura que el comienzo del trabajo i en la máquina $(j+1)$ es posterior a su hora de finalización en la máquina j. Finalmente, la ecuación (8) garantiza que el comienzo del trabajo $(i+1)$ en una máquina es posterior a la hora de finalización de un trabajo i en la misma máquina.

En la literatura para la resolución de la programación del *flowshop* se han presentado múltiples algoritmos que minimizan el *makespan*. En la mayoría, se asume que los trabajos están simultáneamente disponibles y que los tiempos de procesamiento son determinísticos. Estos supuestos también se tomarán en cuenta para la aplicación de la metodología que se va a presentar. Hasta ahora, ha habido tanto heurísticas, como métodos exactos, para la resolución del problema. Aunque los métodos exactos encuentran la solución óptima, estos requieren de mucha memoria y

tiempo computacional para problemas muy grandes. Por este motivo, las heurísticas son metodologías económicas, eficientes y útiles para encontrar buenas soluciones de este tipo de problemas.

El primer algoritmo fue propuesto por Johnson (1954), quien halló una metodología para encontrar la secuencia óptima para n trabajos y 2 máquinas. Luego, Palmer (1965) propuso una heurística en la que la secuenciación se realiza con base al cálculo del índice de la pendiente, el cual es organizado de manera decreciente para obtener el orden óptimo.

Después, el algoritmo de Johnson fue extendido por Campbell, et al (1970), el cual cuenta con cierto número de iteraciones antes de encontrar la solución final. Estos autores plantearon un proceso en el que se generan $m - 1$ problemas de dos máquinas artificiales, los cuales son resueltos por el método de Johnson. Esta heurística es conocida como CDS.

Luego, Gupta (1971) plantea una nueva técnica para el cálculo del índice de la pendiente con el propósito de utilizarlo para el ordenamiento de las tareas. Después, Nawaz, et al. (1983) desarrollaron una metodología basada en el tiempo total de procesamiento de las tareas. En este sentido, la prioridad de los trabajos en el cronograma se hace por el concepto de que el trabajo con mayor tiempo en proceso tiene la mayor ventaja sobre los demás. Este algoritmo es conocido como el NEH.

Más adelante, Nagar, et al. (1996) utilizaron dos métodos: *branch and bound* y el algoritmo genético. Nowicki y Smutnicki (1996) implementaron la búsqueda tabú para resolver el problema, mientras Neppalli, et al. (1996) utilizaron el algoritmo evolutivo básico para el problema de las dos máquinas. Asimismo, usaron el enfoque de algoritmos genéticos para llevar a cabo lo mismo.

Desde aquí, muchos otros autores han presentado otras metodologías para acercarse a la mejor solución. Modrak, et al. (2009) hizo la comparación entre los diversos algoritmos heurísticos en función del valor del makespan. Las heurísticas comparadas fueron el algoritmo de Palmer, el algoritmo de Gupta, el algoritmo CDS, el algoritmo NEH y el algoritmo heurístico MOD. El NEH usa iteraciones máximas entre todos los demás (mucha memoria y tiempo computacional), mientras que Palmer solo usa una iteración para llegar al resultado definitivo. Sin embargo, aunque el algoritmo de Palmer es muy rápido, se encontró que este carece de precisión y los resultados obtenidos con este no coinciden con el mejor resultado entre todos los demás algoritmos heurísticos.

III. METODOLOGÍA

Como se mencionó anteriormente, se utilizarán dos metodologías de búsqueda de local para la resolución del problema. Para ambas se utiliza como solución inicial la secuencia ordenada de uno hasta 500 {1, 2, 3, …, 500}.

El primer método se basa en un algoritmo de intercambio, también llamado *swap*. El pseudocódigo de este se ilustra en el *Algoritmo 1*. Este consiste en tomar un trabajo en cierta posición e intercambiarlo por otro. En cada iteración se encuentran dos posiciones aleatorias para cambiar en la mejor secuencia encontrada hasta el momento (líneas 6 y 7). Al realizar el cambio, si el C_{max} es menor al mejor C_{max} encontrado (línea 11), entonces las iteraciones se inicializan nuevamente en 0 (línea 12), se actualiza la mejor solución y C_{max} (líneas 13 y 14), y se vuelve a hacer el proceso. Si al hacer el cambio la solución no es mejor, se repite el algoritmo hasta que se cumpla con el número

de iteraciones impuestas desde un inicio. Esto quiere decir que el algoritmo parará solo hasta que se cumpla un número de iteraciones en las que la secuencia no haya presentado ninguna mejora.

Algoritmo 1. Búsqueda local basada en intercambio aleatorio

```
procedimiento Busqueda_Swap ()
Input: tiempos de procesamiento de los n trabajos en las m máquinas, número de iteraciones máximas, solución inicial (orden lexicográfico).
Output: secuencia que minimiza el makespan, C_max.
(1) mejor_solucion = [1, 2, …, n]
(2) Cmax_best = makespan (mejor_solucion, t)
(3) encontro = True
(4) iteraciones = 0
(5) while iteraciones < iteracionesMax:
(6)     posicionUno = random [1,500]
(7)     posicionDos = random [1,500]
(8)     solucion = swapPosiciones (mejor_solucion, posicionUno, posicionDos)
(9)     Cmax = makespan (solucion, t)
(10)    iteraciones +=1
(11)    if Cmax < Cmax_best then
(12)        iteraciones =0
(13)        mejor_solucion = solucion
(14)        Cmax_best = Cmax
(15) Return mejor_solucion, Cmax
End Busqueda_Swap
```

La segunda metodología está inspirada en la propuesta por Laha y Chakraborty (2009), quienes desarrollaron un algoritmo heurístico constructivo basado en el principio de búsqueda local por medio de la inserción de trabajos. En comparación con el primero propuesto, este resulta ser mucho más exhaustivo, por lo que evalúa muchas más posiciones y recae en la estocasticidad de los números aleatorios.

En este sentido, a continuación se explica el algoritmo a utilizar para la resolución del problema:

1. Para cada trabajo i se calcula el tiempo total de procesamiento T_i, el cual está dado por:

$$T_i = \sum_{j=1}^{m} p_{ij} \quad \forall\ i = 1,2,\dots,n$$

   Con esto, se organiza una secuencia en orden descendente según estos tiempos totales de procesamiento.

2. Se realiza un mecanismo de desplazamiento hacia adelante (*forward shift mechanism*) con la secuencia encontrada en el Paso 1. Asimismo, se aplica el mecanismo de desplazamiento hacia atrás (*backward shift mechanism*). Este procedimiento generaría $2(n-1)$ secuencias posibles y se escoge la que genere un menor C_{max}. A continuación, se muestran unos ejemplos de cómo se aplican estos mecanismos si, a modo de ejemplo, la secuencia del primer paso hubiera sido $\{2,5,3,1,4,6\}$.

Forward shift mechanism: se generarían las siguientes secuencias por evaluar $\{5,2,3,1,4,6\}, \{5,3,2,1,4,6\}, \{5,3,1,2,4,6\}, \{5,3,1,4,2,6\}, \{5,3,1,4,6,2\}$.

Backward shift mechanism: se generarían las siguientes secuencias por evaluar $\{5,2,3,1,4,6\}, \{3,2,5,1,4,6\}, \{1,2,5,3,4,6\}, \{4,2,5,3,1,6\}, \{6,2,5,3,1,4,6\}$.

En este sentido, se evaluaría el C_{max} para las 10 secuencias generadas y se escogería la que represente el menor valor de este.

3. Fije $k = 1$. Seleccione los dos primeros trabajos de la secuencia del Paso 2. Obtener la mejor de las dos secuencias parciales de estos dos trabajos y considérela como la secuencia actual.

4. Fijar $k = k + 1$. Se generan secuencias parciales, cada una de $2k$ trabajos, insertando los dos siguientes (es decir, el par k-ésimo de) trabajos como un bloque de la secuencia del paso 2 en cada una de las $2k - 1$ posiciones posibles de la secuencia actual. Se selecciona la mejor secuencia parcial como la secuencia actual (la secuencia actual tiene ahora $2k$ tareas).

   A continuación, el primer trabajo de la k-ésima pareja se coloca en todas las posiciones posibles de la secuencia actual, produciendo $2k - 1$ secuencias parciales. Si la mejor de estas $2k - 1$ secuencias parciales es mejor que la secuencia actual, se establece como secuencia actual. En caso contrario, la secuencia actual permanece igual. El segundo trabajo del par k se coloca en todas las posiciones posibles de la secuencia actual, produciendo otro conjunto de $2k - 1$ secuencias parciales. Si la mejor de estas $2k - 1$ secuencias parciales es mejor que la secuencia actual, se utiliza para actualizar la secuencia actual.

5. Se repite el proceso del Paso 4 hasta que ya no haya más trabajos por programar.

En el *Algoritmo 2* se muestra el pseudocódigo de esta metodología.

Algoritmo 2. Búsqueda local basada en inserción exhaustiva.

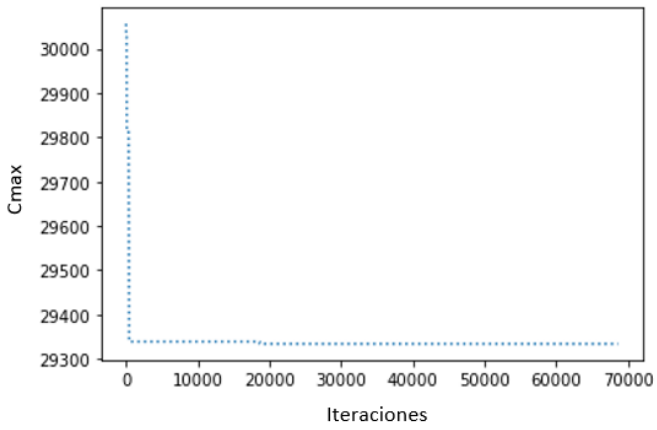
```
procedimiento Busqueda_Insercion ()
Input: tiempos de procesamiento t de los n trabajos en las m máquinas, número de iteraciones máximas, solución inicial (orden lexicográfico).
Output: secuencia que minimiza el makespan, Cmax.
(1) mejor_solucion = [1, 2, …, n]
(2) Cmax_best = makespan (mejor_solucion, t)
(3) T = lista de trabajos organizada de mayor a menor según suma de tiempos de procesamiento.
(4) for i in len (T):
(5)     Lista = inserción (T, 0, i)
(6)     Cmax_lista = makespan(lista,t)
(7)     If Cmax_lista < Cmax_best then
(8)         Mejor_solucion = lista
(9)         Cmax_best = Cmax_lista
(10) for j in len (T):
(11)    Lista = inserción (T, j, 0)
```

```
(12)        Cmax_lista = makespan(lista,t)
(13)        If Cmax_lista < Cmax_best then
(14)              Mejor_solucion = lista
(15)              Cmax_best = Cmax_lista
(16)   pareja = mejor secuencia de los primeros dos trabajos de la mejor_solucion.
(17)   for k in mejor_secuencia:
(18)        subsecuencia = pareja.copy
(19)        Cmax = 9999
(20)        for i in len(pareja):
(21)             Solucion = pareja.insert(k,i)
(22)             Cmax_temp = makespan (solucion, t)
(23)             If Cmax_temp < Cmax then
(24)                  subsecuencia = solucion
(25)                  Cmax = Cmax_temp
 (26)  Return subsecuencia, Cmax
End Busqueda_Insercion
```

IV. EXPERIMENTOS COMPUTACIONALES Y ANÁLISIS DE RESULTADOS

La implementación de estas heurísticas se realizó en el lenguaje de programación Python versión 4.1.5. Para la evaluación de estas se utilizaron las instancias propuestas por Taillard (1993), las cuales fueron precisamente generadas para probar algoritmos para el problema flowshop permutado con el criterio del *makespan*. Existen 12 conjuntos de 10 instancias, que van desde 20 trabajos y 5 máquinas hasta 500 trabajos y 20 máquinas, donde $n \in \{20, 50, 100, 200, 500\}$ y $m \in \{5, 10, 20\}$. En concreto, el propósito de este documento es evaluar las heurísticas planteadas con las instancias más grandes, de manera que los experimentos fueron realizados únicamente con las 10 instancias de 500 trabajos y 20 máquinas.

Como se mencionó anteriormente, el primer algoritmo de *swap* depende de una cantidad de iteraciones, las cuales, hasta cierto punto, van a determinar la calidad de la solución encontrada y el tiempo de cómputo consumido. Para identificar la cantidad de iteraciones útiles para correr el modelo, inicialmente se impusieron 50000 iteraciones. En la *Gráfica 1* se evidencia que realmente no es necesario indicar tantas, pues resulta que cuando el algoritmo encuentra una buena solución (óptimo local), ya se queda con esta y, por más iteraciones que se impongan, la solución no va a mejorar. Por este motivo, todas las instancias se corrieron con 10000 iteraciones. Con esto en mente, en la *Tabla 1* se resumen los resultados encontrados con el método *swap*.



Gráfica 1. Método *swap* con 50000 iteraciones sin mejora.

Tabla 1. Resultados algoritmo *swap* aleatorio.

Id	n	m	BKS	C_{max}^{Inic}	Gap_{BKS}^{Inic}	C_{max}^{Swap}	Gap_{BKS}^{Swap}	Gap_{Inic}^{Swap}	CPU (min)
ta111	500	20	26040	30121	15,7%	29398	12,9%	-2,4%	4.4
ta112	500	20	26500	31202	17,7%	29980	13,1%	-3,9%	1.6
ta113	500	20	26371	30447	15,5%	29664	12,5%	-2,6%	2.1
ta114	500	20	26456	30355	14,7%	29747	12,4%	-2,0%	4.1
ta115	500	20	26334	30099	14,3%	29595	12,4%	-1,7%	2.5
ta116	500	20	26469	30946	16,9%	29865	12,8%	-3,5%	3.9
ta117	500	20	26389	30792	16,7%	29401	11,4%	-4,5%	3.5
ta118	500	20	26560	31034	16,8%	29735	12,0%	-4,2%	4.3
ta119	500	20	26005	30634	17,8%	29293	12,6%	-4,4%	3.0
ta120	500	20	26457	30148	14,0%	29628	12,0%	-1,7%	3.8

Analizando cada una de las instancias, es posible decir que en ninguno de los casos se genera una solución muy cercana a las ya encontradas (BKS). Esto se debe a que este algoritmo de búsqueda local no es muy preciso, pues depende mucho de la estocasticidad, debido a que se generan números aleatorios para realizar los cambios en el vecindario. Numéricamente, en promedio, los resultados de este método de inserción presentaron un gap de 12.42%. Adicionalmente, este presenta un tiempo de corrida promedio de 3.32 minutos.

Ahora bien, en la *Tabla 2* se muestran los resultados encontrados con la segunda búsqueda de local planteada. En este caso, es posible ver que la calidad de la solución resulta ser mucho mejor, pues el gap promedio se reduce a tan solo ser del 2.91%. No obstante, esta calidad se consigue a un costo computacional un poco más alto que el del primer algoritmo, puesto que en promedio consume un tiempo de 17.84 minutos. Esto se debe a que esta heurística resulta ser mucho más exhaustiva que la primera, de manera que evalúa muchas más posibilidad para poder llegar a la mejor solución posible.

Id	n	m	BKS	C_{max}^{Inic}	Gap_{BKS}^{Inic}	C_{max}^{Inser}	Gap_{BKS}^{Inser}	Gap_{Inic}^{Inser}	CPU (min)
ta111	500	20	26040	30121	15,7%	26856	3,1%	-10,8%	17,5
ta112	500	20	26500	31202	17,7%	27413	3,4%	-12,1%	17,3
ta113	500	20	26371	30447	15,5%	27186	3,1%	-10,7%	17,2
ta114	500	20	26456	30355	14,7%	27244	3,0%	-10,2%	18
ta115	500	20	26334	30099	14,3%	27098	2,9%	-10,0%	17,5
ta116	500	20	26469	30946	16,9%	27233	2,9%	-12,0%	18,3
ta117	500	20	26389	30792	16,7%	26988	2,3%	-12,4%	19,2
ta118	500	20	26560	31034	16,8%	27341	2,9%	-11,9%	17,9
ta119	500	20	26005	30634	17,8%	26707	2,7%	-12,8%	17,2
ta120	500	20	26457	30148	14,0%	27197	2,8%	-9,8%	18,3
				Promedio	16,01%		2,91%	-11,28%	17,84

Dada la gran diferencia que ambos algoritmos presentan en la calidad de la solución encontrada, se puede decir que el segundo método es mucho mejor que el primero, pues aún con la diferencia que presentan en el tiempo computacional, los valores de C_{max} alcanzados son mucho más valiosos con esta última. En la *Tabla 3* se observan los resultados finales encontrados.

Tabla 3. Resumen de resultados.

Método	Gap_{BKS}	$Gap_{Inicial}$	CPU promedio (min)
Swap	12.42%	-3.09%	3.32
Inserción	2.91%	-11.28%	17.84

V. CONCLUSIONES

El flowshop permutado ha sido un problema muy estudiado en la literatura desde hace mucho tiempo. Este documento presentó un enfoque basado en dos algoritmos de búsqueda de local, aplicando la metodología de *swap* e inserción. Los resultados encontrados no superaron a los métodos ya estudiados por otros autores, pero alcanzaron soluciones factibles, acordes con las restricciones del problema. La dificultad computacional restringe la cantidad de iteraciones y exhaustividad que puede utilizarse en el algoritmo y, por ende, la calidad de la solución.

Dentro de los dos métodos presentados, se identificó que el método de inserción alcanza soluciones mucho mejores que con el método de swap. Aunque este representa tiempos computacionales mayores, la calidad de las soluciones refleja una mejora significativa en la metodología utilizada. En este caso, el método *swap*, a diferencia del de inserción, recae mucho en la estocasticidad utilizada, lo cual hace muy difícil encontrar una buena solución. Por lo tanto, se llega a la conclusión de que la exhaustividad del segundo algoritmo promete mucho mejores soluciones que la estocasticidad del primero.

REFERENCIAS

Campbell, H. G., Dudek, R. A., & Smith, M. L. (1970). A heuristic algorithm for the n job, m machine sequencing problem. *Management science*, *16*(10), B-630.

Cheng, T. E., & Lin, B. M. (2009). Johnson's rule, composite jobs and the relocation problem. *European Journal of Operational Research*, *192*(3), 1008-1013.

Gupta, J. N. (1971). A functional heuristic algorithm for the flowshop scheduling problem. *Journal of the Operational Research Society*, *22*(1), 39-47.

Johnson, S. M. (1954). Optimal two-and three-stage production schedules with setup times included. *Naval research logistics quarterly*, *1*(1), 61-68.

Modrak, V., Semanco, P., & Kulpa, W. (2013). Performance measurement of selected heuristic algorithms for solving scheduling problems. In *2013 IEEE 11th International Symposium on Applied Machine Intelligence and Informatics (SAMI)* (pp. 205-209). IEEE.

Nagar, A., Heragu, S. S., & Haddock, J. (1996). A combined branch-and-bound and genetic algorithm based approach for a flowshop scheduling problem. *Annals of Operations Research*, *63*(3), 397-414.

Nawaz, M., Enscore Jr, E. E., & Ham, I. (1983). A heuristic algorithm for the m-machine, n-job flow-shop sequencing problem. *Omega*, *11*(1), 91-95.

Neppalli, V. R., Chen, C. L., & Gupta, J. N. (1996). Genetic algorithms for the two-stage bicriteria flowshop problem. *European journal of operational research*, *95*(2), 356-373.

Nowicki, E., & Smutnicki, C. (1996). A fast tabu search algorithm for the permutation flow-shop problem. *European Journal of Operational Research*, *91*(1), 160-175.

Palmer, D. S. (1965). Sequencing jobs through a multi-stage process in the minimum total time—a quick method of obtaining a near optimum. *Journal of the Operational Research Society*, *16*(1), 101-107.

Wu, C. C., & Lee, W. C. (2009). A note on the total completion time problem in a permutation flowshop with a learning effect. *European Journal of Operational Research*, *192*(1), 343-347.