Flow Shop Sequencing

Gustavo Aguilar[1], Juan Barrerra[2], Juan Trujillo[3]

Resumen— **El documento versa sobre el problema flow shop scheduling, presentando una definicion general de los problemas tipo flow shop, el proceso de modelamiento, estado del arte y ejemplos concretos donde el flow shop scheduling puede ser utilizado para optimización de procesos. Se propone resolver el problema utilizando dos algoritmos de optimización clásicos: enumeración exhaustiva y branch and bound. Se analizan los resultados obtenidos y se discute la efectividad de estos métodos para el problema.**

Palabras clave— **Conjunto de máquinas, secuencia de trabajos, minimizar tiempo total de trabajo, enumeración exhaustiva, branch and bound.**

I. Introducción

EN la programación de problemas tipo Flow Shop se busca conseguir una secuenciación adecuada para cada trabajo que entra en la línea de flujo y para su respectivo procesamiento en un conjunto de máquinas determinado, las cuales cumplen con una operación especifica dentro del proceso. Junto con ello se desea mantener un flujo continuo de tareas de procesamiento con un mínimo tiempo de inactividad y un mínimo tiempo de espera.

Flow Shop como modelo de planificación de tareas posee las siguientes características [4]:

- Cada máquina realiza una sola operación y para un trabajo a la vez.
- Las operaciones requieren una sola visita (ejecución) para ser completadas (en caso de que este trabajo no utilice la correspondiente máquina se denomina que el tiempo es cero).
- Los trabajos pasan por cada máquina una sola vez.
- El orden de las máquinas es siempre el mismo.

Como ventajas del uso de un sistema de configuración Flow Shop se pueden mencionar [4]:

- Manejo de materiales reducido.
- Escasa existencia de trabajos en curso.
- Minimos tiempos de fabricación.
- Simplificación de los sistemas de programación y control de la producción.

.dz

II. Estado del arte

El Flow Shop Scheduling es uno de los problemas más estudiados en la literatura de Investigación Operativa [10]. En la literatura relacionada, la minimización de la duración máxima, C_{max}, ha sido comúnmente elegido por los investigadores como el objetivo a optimizar. La mayoría de los investigadores se han centrado en implementar métodos aproximados para encontrar buenas soluciones sin tiempos de cálculo excesivos.

Ha habido una gran cantidad de artículos publicados con algoritmos y procedimientos. Entre los métodos más relevantes, se encuentran la heurística NEH (Nawaz, Enscore Jr., Ham, 1983), la cual es una de las más eficiente entre las heurísticas constructivas y el Iterated Local Search (Stütze, 1998), la cual se presenta como la metaheurística más eficiente para el problema.

Otros métodos más recientes, como el como el Iterated Greedy (IG) de Ruiz y Stützle (2007), han mejorado algunos de los mejores algoritmos existentes. Sin embargo, el nuevo estado de la técnica sigue sin estar claro debido a la falta de un marco homogéneo para realizar la comparación entre algoritmos, los cuales han sido comparado bajo diferentes condiciones:

- Probado en diferentes condiciones computacionales (diferentes lenguajes de programación, diferentes computadores, sistemas operativos, etc.).
- Comparación de algoritmos con diferentes usos de tiempo de CPU.
- Uso de diferentes benchmarks

Además, se debe hacer un esfuerzo especial al comparar heurísticas eficientes con las mejores metaheurísticas bajo el mismo criterio de detención, ya que el tiempo de CPU requerido por algunas heurísticas es relativamente alto en comparación con algunas metaheurísticas.

III. Modelamiento

Consideremos que se tiene un conjunto de m máquinas y n trabajos, cada trabajo comprende un conjunto de m operaciones, donde deben ser procesadas en diferentes máquinas. Además todos los trabajos deben procesarse en el mismo orden a través de las máquinas, no existiendo restricciones de procedencia entre los diferentes trabajos. El problema consistirá en encontrar la secuencia de trabajos que minimice el tiempo total de trabajo.
Para encontrar un modelo apropiado, es bien sabido que existen diferentes formas de abodar la problemática, ya sean con métodos de aproximación o bien heúristicos, como por ejemplo, tabu search o genetic algorithms. Sin embargo, si se considera un tamaño de m, n tal que permita procesarlo en un computador ordinario, entonces se puede aplicar Mixed Interger Programming (MIP) como para dar un primer acercamiento respecto de este tipo

[1]Dpto Ingeniería Informática, Universidad Santiago de Chile, e-mail: gustavo.aguilar@usach.cl
[2]Dpto Ingeniería Informática, Universidad Santiago de Chile, e-mail: juan.barrerab@usach.cl
[3]Dpto Ingeniería Informática, Universidad Santiago de Chile, e-mail: juan.trujillo@usach.cl

problemas, e indentificar los componentes necesarios y posteriormente optimizar la función objetivo.

A. Definiciones

De acuerdo a la descripción del problema, tenemos las siguientes definiciones en la tabla I.

Definición	Descripción
J	Conjunto de trabajos 1,...,n.
M	Conjunto de máquinas 1,...,m.
O_p	Conjunto de operaciones 1,...,m.
J_i	El trabajo $i-ésimo$ en la permutación del conjunto de trabajos.
p_{ik}	Tiempo de procesamiento del trabajo $J_i \in J$ en la máquina k.
v_{ik}	Tiempo de espera de la máquina k antes de que comience el el trabajo J_i $(\forall i \in J)(\forall$k$\in$M$)$.
w_{ik}	Tiempo de espera del trabajo J_i después de finalizar la operación de la máquina k, mientras espera por la máquina $k+1$ a que sea liberada.

Tabla I: Definiciones del problema

B. Variables de decisión

Dado que cada trabajo J_i debe tener asignado una máquina mientras realiza una operación, entonces se define las permutaciones de un modo binario, tal que:

$$x_{ij} = \begin{cases} 1 & \text{si } j \text{ es asignado a la } i-ésima \\ & \text{permutación, esto es cuando } J_i = j \\ 0 & \text{cualquier otro caso} \end{cases} \tag{1}$$

En otras palabras, (1) indica que si la variable tiene valor 1, entonces una máquina está ejecutando la operación de algún trabajo j.

C. Restricciones

$$\forall i \in J : \sum_{j=1}^{n} x_{ij} = 1 \tag{2}$$

$$\forall j \in J : \sum_{i=1}^{n} x_{ij} = 1 \tag{3}$$

Las restricciones (2) y (3), limitan a que las máquinas puedan ejecutar sólo un trabajo a la vez, y que un trabajo sólo pueda ser ejecutado por una máquina a la vez.

$$\forall k \in M - \{m\} : w_{1k} = 0 \tag{4}$$

De la restricción (4), indica que para el trabajo J_1, todos los tiempos de espera a que se desocupe alguna máquina $k+1$, luego de terminar una máquina k, son 0. Esto es evidente ya que no hay ningún trabajo que le antecede y que haya ocupado alguna máquina $k+1$.

$$\forall k \in M - \{1\} : v_{1k} = \sum_{r=1}^{k-1} \sum_{i=1}^{n} p_{ir} x_{1i} \tag{5}$$

Cuando se observa una máquina k del primer trabajo (i.e $i = 1$), su tiempo de espera (v_{1k}) va estar dado por los tiempos de procesamiento de las máquinas que le anteceden, exceptuando la máquina 1, que en cuyo caso es 0 por definición, por lo que la restricción (5) garantiza que la máquina se ejecute en el orden correcto para el trabajo J_1.

$$(\forall i \in J - \{n\})(\forall k \in M - \{m\}) :$$

$$v_{i+1,k} + \sum_{j=1}^{n} p_{jk} x_{i+1,j} + w_{i+1,k} = w_{ik} + \sum_{j=1}^{n} p_{j,k+1} x_{ij} + v_{i+1,k+1} \tag{6}$$

La restricción (6) nos proporciona la consistencia y viabilidad entre los tiempos de inactividad de la máquina y el trabajo [1].

D. Función objetivo

Teniendo que m como la última máquina del conjunto J, nos interesa optimizar el tiempo de ejecución de todos los trabajos de esta, en conjunto con el tiempo de espera [1][2][3], es decir:

$$C_{max} = \sum_{i}^{n} (v_i m + \sum_{j=1}^{n} p_{jm} x_{ij}) \tag{7}$$

IV. Algortimos

En el presente laboratorio se han trabajado dos algoritmos clásicos para resolver la problemática propuesta de FSS, correspondientes a Enumeración Exhaustiva (EE) y Ramificación y Poda (RP, Branch and Bound en inglés).
Para el primer método se desarrolla en Javascript con IDE Visual Studio Code, y para Branch and Bound el problema se modela usando Julia y JuMP, con el solver GLPK y el ambiente Jupyter Notebook (en VM cloud-service).

A. Enumeración Exhaustiva

El objetivo de este algoritmo es poder poblar una matriz de tiempos de entrada y salida de para cada máquina y trabajo, con el fin de encontrar el último par de tiempos de la última máquina, según una secuencia dada, como se ilustra en la siguiente tabla (II), donde nos interesa obtener el OUT de la máquina corresponde MN, es decir x^N_{N2}.

	Máquinas					
	M1		M2		...MN	
J	IN	OUT	IN	OUT	IN	OUT
J1	X^1_{11}	$x1_{12}$	x^1_{21}	x^1_{22}	x^1_{N1}	x^1_{N2}
J2	X^2_{11}	$x2_{12}$	x^2_{21}	x^2_{22}	x^2_{N1}	x^2_{N2}
JN	X^N_{11}	xN_{12}	x^N_{21}	x^N_{22}	x^N_{N1}	x^N_{N2}

Tabla II: Matriz de tiempos de entrada y salida de cada máquina para cada trabajo de alguna secuencia en particular.

De lo anterior, se obtiene el tiempo que corresponde a una secuencia de trabajos en particular, por lo que

se quiere obtener una secuencia S_i cuyo tiempo x_{N2}^N sea el mínimo del conjunto de las secuencias para $i = N!$ permutaciones.

A continuación se detalla el pseudo código que permite poblar la matriz anteriormente descrita, la que deriva en encontrar el tiempo total de las ejecuciones de las operaciones (ver [4] el código completo en Javascript).

Listing 1: FSS Enumeración Exhaustiva - Pseudo código

```
Set matrix as {
    j1: [tiempos_operaciones_trabajo1],
    j2: [tiempos_operaciones_trabajo2],
    ...,
    jN: [tiempos_operaciones_trabajoN]
}
Set keys as [j1, j2,...,jN]
Set jobsQ as keys.length
Set machines as matrix.j1.length // Asumiendo que todos los trabajos realizan las mismas operaciones.
Set combinations as [
    [j1,j2,...,jN],
    [j2,j1,...,jN],
    ...,
    [jN,...,j1,j2]
] // N! Permutaciones

For k to combinations.length {
    Set jobsKeys = combinations[k] // Seleccionamos la permutación de trabajos
    For j to jobsQ {
        Set jobsTiming as Empty_array // es un arreglo de par de números, que contienen el tiempo de entrada y salida de una máquina en un trabajo determinado.
        For i to machines {
            // primer trabajo en primera máquina
            If = 0 and j = 0 {
                jobsTiming add [0, job1_machine1_time]
            }
            // primer trabajo en máquinas posteriores
            Else If j = 0 {
                // Tomar el tiempo del par derecho de la máquina anterior
                Set lastOut as jobsTiming[jobsTiming.length - 1][1]}
                // Tiempo de salida del máquina J
                Set afterOut as lastOut + job_1_machinei_time
                jobsTiming add [lastOut, afterOut] // Se agrega el par de tiempos
                de entrada y salida de la máquina J del trabajo 1.
            }
            // segundos trabajos en adelante y de la segunda máquina en adelante
            Else {
                // Tiempo de salida de la máquina anterior en el mismo trabajo
                Set outMachineBefore as jobJ_machineI-1_out
                // Tiempo de salida de la misma máquina pero del trabajo anterior
                Set outSameMachineJobBefore as jobJ-1_machineI_out
                // Si el tiempo de salida del trabajo anterior (j-1) de la misma máquina (m), es superior al tiempo del mismo trabajo (j) pero de una máquina anterior (m-1), quiere decir, que aún no está la máquina disponible para su uso, por lo que el tiempo de salida correponde al outSameMachineBefore.
                If outSameMachineJobBefore > outMachineBefore {
                    Set afterOut = outSameMachineJobBefore + jobJ_machinei_time
                    jobsTiming add [outSameMachineJobBefore, afterOut]
                }
                // Caso en que la máquina m ya está disponible, sin embargo la máquina m-1, aún no termina su trabajo
                else{
                    Set afterOut = outMachineBefore + jobJ_machinei_time
                    jobsTiming add [outMachineBefore, afterOut]
                }
            }
        }
    }
}
```

B. Ramificación y Poda (Branch and Bound)

Para este método, se implementa el IDE de Jupyter Notebook, con el lenguaje de programación Julia y JuMP para su implementación, con GLPK (GNU Linear Programming Kit), donde de forma natural aplica la técnica de BB, para problemas de Mixed Integer Programming (MIP).
A continuación el pseudo código que permite encontrar el tiempo de optimización de la secuencia de trabajos (ver completo en [5])

Listing 2: FSS Branch Bound - Pseudo código

```
Set modelo as Model(parametros_GPLK)
// No es necesario que Jobs.length = Machines.length
Set matriz as [ J1_M1 J1_M2 ... J1_MN; J2_M1 J2_M2 ... J2...MN; ...; JN_M1 ... JN_MN]
Set variable_x as matriz_JxJ
Set variable_w as matriz_JxM
Set variable_v as matriz_JxM

Set objetivo as sum(varible_v) + sum(matriz* variable_x)

// Observar las restricciones del apartado anterior.
For i to N {
    Set restriccion1
}
For j to N {
    Set restriccion2
}
For k to M {
    Set restriccion3
}
For i in N-1{
    Set restriccion5
}

CALL Procedure optimize!
```

C. Datos

En ambos algoritmos se realizan pruebas con dataset con diferentes tamaños de población de trabajos y máquinas, y que a su vez fueron comparadas entre los métodos, y con la literatura. En algunos

casos se encontraron diferencias, como se detallará en el apartado de resultados y discusión.

D. Rendimiento y Errores

Para el problema de FSS, cuando se tiene un dataset muy grande (i.e 20x20), bajo los métodos utilizados, las ejecuciones comienzan a presentar problemas en el ambiente de trabajo, tales como; desborde de memoria y congelamiento. Para el caso de EE, se tiene que el máximo de trabajos permitidos fue de 9 (para 17 trabajos) y para BB el máximo fue 6.
Por otra parte, para 3 trabajos o menos, la cantidad de máquinas se llegó a probar por más de 1500000, sin tener problemas de rendimiento o desbordamiento de memoria. Para 8 trabajos y 200 máquinas, se encontró desborde de memoria. En la siguiente sección, se podrá ver con mayor detalle lo anteriormente señalado.

V. Resultados y Análisis

Trabajos	Máquinas
4	$\sim$ 700000
5	$\sim$ 100000
6	$\sim$ 20000
7	$\sim$ 2010
8	$\sim$ 200
9	17
10+	-

Tabla III: Tabla de límites de desbordes de memoria para método de EE, en ambiente local con CPU Intel i7-8665U, RAM 16GB, Win10-x64.

De la tabla III, podemos observar que a medida que aumenta la cantidad de trabajos, el número de máquinas que producen un desborde de memoria disminuye.

Para 4 trabajos, se requieren aproximadamente 700.000 máquinas para desbordar la memoria, y si se aumenta a 5 trabajos, la cantidad de máquinas disminuye significativamente, a aproximadamente 100.000.

Cuando se consideran 9 trabajos, la cantidad de máquinas solo puede aumentar hasta 17. Para 10 o más trabajos, se produce un desborde de memoria independiente del número de máquinas. Esto es debido a que en el algoritmo de EE, se debe almacenar todas las permutaciones de los trabajos ensamblados en la operación, por lo que se tendrían 10! o más permutaciones en memoria, por lo que se comprueba empíricamente la complejidad del problema de O(n!) [6].

De la tabla IV, podemos observar que para instancias muy pequeñas (3x3, 4x5, 4x3), la solución basada en branch and bound dio mejores resultados en cuanto a tiempo respecto a la solución basada

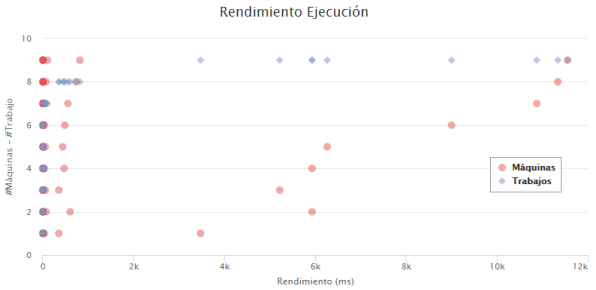

Fig. 1: Rendimiento de las ejecuciones, para diferentes cantidad de máquinas y trabajos, con límite de 9 trabajos.

DS (JxM)	TO-EE(TP[s])	TO-BB(TP[s])	TO-F
3x3	44 (0.074)	44 (0.001)	44
4x5	44 (0.075)	44 (0.018)	49
4x3	270 (0.074)	269.99 (0.020)	270
6x3	396 (0.089)	396 (15.781)	-
7x5	663 (0.183)	Freeze	-
5x20	1278 (0.081)	Freeze	1278
10x20	Error	Freeze	-
20x20	Freeze	Freeze	-

Tabla IV: Tabla de resultados de los métodos EE y BB. TO: Tiempo Optimizado, TP: tiempo procesado, F: Fuente [7]).

en numeración exhaustiva. Sin embargo, para instancias más grandes (7x5, 5x20), la enumeracion exhaustiva dio mejores resultados, y el branch and bound incluso no fue capaz de terminar de computar. Para las instancias más grandes (10x20, 20x20), ninguno de los dos métodos fue capaz de entregar una solución. Para el caso de los datasets pequeños se obtuvieron mediante los experimentos los mismos resultados que en los papers de estudio utilizados como referencia [7], por tanto se puede estar conforme con los resultados obtenidos para esos casos de espacio muestral.

En la figura 1 se puede observar el comportamiento de los tiempos de rendimiento (ejecución del algortimo) para cierta cantidad de trabajos y máquinas, donde se obtiene que para un set de pocos trabajos y máquinas, el tiempo es $\sim$ 0 [ms], como también se puede ver en la tabla IV. Además se observa que para un set de 9 trabajos, en todas sus configuraciones de máquinas, se logra obtener un tiempo de rendimiento con mayor valor.
Respecto a lo evidenciado en este experimento, es importante destacar que para el caso 4x5 de la tabla IV, en ambos métodos se obtuvo un resultado exacto, sin embargo, de acuerdo a la fuente, se obtuvo un valor de 49 unidades de tiempo de operación. Este último se ha desarrollado con un algoritmo heurístico de NEH polinómico, lo cual arroja un resultado aproximado y no exacto, a diferencia de lo obtenido en los métodos clásicos.

VI. Discusión

Mediante la aplicación de los algoritmos utilizados en el presente trabajo (enumeración exhaustiva y branch and bound), se ha confirmado lo que establece

la literatura: la resolución del Flow Shop Scheduling mediante el uso de soluciones computacionales basadas en métodos de optimizacion clásicos permiten resolver eficientemente el problema para instancias pequeñas. Sin embargo, a medida que el tamaño de las instancias aumenta, se hace evidente que este tipo de métodos no son suficientes.

Se tiene que diferentes métodos clásicos consideran diferentes rendimientos para distintos según el tamaño de la instancia. Por ejemplo, en el caso de los métodos utilizados en este trabajo, para instancias muy pequeñas es mejor usar branch and bound, y para instancias un poco más grandes, es mejor utilizar enumeración exhaustiva. Sn embargo, ninguno de los dos métodos es capaz de entregar un resultado para instancias más grandes (más de 10 trabajos).

Si se desea obtener el valor óptimo para instancias grandes, es necesario revisar otros métodos de solución.

VII. Conclusiones

A partir del trabajo realizado, se pueden establecer las siguientes conclusiones:

1. Se hace relativamente sencillo modelar el problema de Flow Shop Scheduling utilizando lenguajes de programación de alto nivel, ya sea orientado a objetos (como Javascript) u homoíconicos (como Julia + Jump).
2. Es posible resolver el problema de Flow Shop Scheduling utilizando soluciones computacionales basadas en métodos de optimización clásicos, sin embargo, esto es factible sólo para instancias pequeñas, y para instancias grandes, otros métodos de solución deben ser considerados.
3. Sería posible obtener soluciones mediante estos métodos tradicionales para instancias mayores con un poder de cómputo y memoria superiores.
4. Mediante las experimentaciones y sus resultados es posible obtener los rangos de instancias en los cuales los métodos de solucion expuestos son eficientes y el límite de ambos.

Referencias

[1] Willem J. Selen and David D. Hott, *A Mixed-Integer Goal-Programming Formulation of the Standard Flow-Shop Scheduling Problem*, Palgrave Macmillan Journals on behalf of the Operational Research Society, 1986.

[2] Débora Pretti Ronconia Sergio Gomez Moralesa, *Formulações matemáticas e estratégias de resolução para o problema job shop clássico*, 2015.

[3] J. Christopher Beck Wen-Yang Ku, *Mixed Integer Programming Models for Job Shop Scheduling: A Computational Analysis*, Computers Operations Research, 73, 165-173., 2016.

[4] Gustavo Aguilar Morita., *FSS solución en Javascript, por método de enumeración exhaustiva. https://github.com/naotoam/magister/blob/master/Optimización en Ingeniería/solucionPreliminar.js*, 2021.

[5] Juan Barrera, *FSS solución Julia+JuMP, método BB. https://github.com/naotoam/julia_optimization/blob/main/FlowShop.ipynb*, 2021.

[6] Miloš Šeda, *Mathematical Models of Flow Shop and Job Shop Scheduling Problems*, World Academy of Science, Engineering and Technology, 2007.

[7] Gustavo Aguilar Morita., *Ejemplos de datasets https://github.com/naotoam/magister/blob/master/Optimización en Ingeniería/datasets.js*, 2021.

[8] Rafael Mellado Silva, *Aplicación del problema Flow-Shop Scheduling a la programación de reparación de equipos médicos*, Pontificia Ubiversidad Católica de Valparaíso,33,35,36., 2014.

[9] Aslıhan Yarıkcan Gözde Yücel Özgün Öner Adalet Göksu Akgün, Gizem Karakaş, *An Application of Permutation Flowshop Scheduling Problem in Quality Control Processes*, Department of Industrial Engineering, Yaşar University, İzmir, Turkey, 2019.

[10] Sündüz Dağ, *An Application On Flowshop Scheduling*, Alphanumeric Journal, Bahadir Fatih Yildirim, vol. 1(1), pages 47-56, December, 2013.

[11] Inyong Ham Muhammad Nawaz, E Emory Enscore, *A heuristic algorithm for the m-machine, n-job flow-shop sequencing problem*, Omega, Volume 11, Issue 1, 1983, Pages 91-95, ISSN 0305-0483,, 1983.

[12] Thomas Sttzle, *Applying iterated local search to the permutation ow shop problem*, Darmstadt, University of Technology Department of Computer Science, Intellectics Group, 1998.

[13] R. Ruiz and T. Stützle, *simple and effective iterated greedy algorithm for the permutation flowshop scheduling problem*, European Journal of Operational Research, 177(3):2033–2049,, 2007.

[14] Paul Baradie Mohie. Arisha, Amr Young, *Flow Shop Scheduling Problem: a Computational Study*, 2002.