Flow Shop Sequencing

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

Resumen— **El presente documento trata sobre el problema de flow shop scheduling, presentando una definición general de los problemas tipo flow shop, se revisa el estado del arte, se describe el proceso de modelamiento del problema y se propone resolver el problema utilizando dos enfoques de solución. En primer lugar, se utilizan dos algoritmos de optimización clásicos: enumeración exhaustiva y branch and bound. Luego, se propone resolver el problema utilizando una solución metaheurística basada en tabu search. Se analizan los resultados obtenidos de ambos enfoques y se compara la efectividad de ambos tipos de métodos para la resolución del problema.**

Palabras clave— **Planificación de trabajos, tiempo de procesamiento, metaheurísticas.**

I. Introducción

EN la programación de problemas tipo Flow Shop se busca conseguir una secuencia 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.
- Mínimos 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 del problema

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.

A. Definiciones

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

[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

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. Resolución con métodos clásicos

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_{N2}^{N}.

	Máquinas					
	M1		M2		...MN	
J	IN	OUT	IN	OUT	IN	OUT
J1	X_{11}^{1}	$x1_{12}$	x_{21}^{1}	x_{22}^{1}	x_{N1}^{1}	x_{N2}^{1}
J2	X_{11}^{2}	$x2_{12}$	x_{21}^{2}	x_{22}^{2}	x_{N1}^{2}	x_{N2}^{2}
JN	X_{11}^{N}	xN_{12}	x_{21}^{N}	x_{22}^{N}	x_{N1}^{N}	x_{N2}^{N}

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

23 errors9 warnings
Click to hide the PDF

                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. Resolución con metaheurísticas

En el presente informe, se propone también resolver el problema mediante una solución metaheurística. Para esto, se implementó un algoritmo basado en tabú search.

A. Algoritmo

A continuación se presenta el algoritmo implementado para la metaheurística de TS.

Listing 3: Algoritmo Tabu Search

```
Funcion TS_QAP(params[]*)
    s* = solución_inicial
    fit* = fit_inicial
    // matrices cuadradas de largo según s*
    tabu_corto = matriz()
    tabu_largo = matriz()
    Repetir hasta tiempo_exploración_espacio
        ji = Seleccionar trabajo pivote i
        Iterar hasta intensidad_maxima
            Repetir Hasta tiempo_exploració
                n_local
                // búsqueda local, contiene el
                    tipo de operador
                vecinos = BusquedaLocal()
                // p contiene el movimiento que
                    fue permitido y
                // el valor fit que se obtuvo
                p = BuscarVecinosTabu(params2
                    [])
                fit* = p.fit
                s* = p.sol
                tabu_corto[x,y] =
                    tiempo_prohibicion
            Fin repetir
    Fin repetir
Fin TS_QAP

funcion BuscarVecinosTabu(params[]*)
    min_sv = ordernarSolucionesDescendientes()
    Iterar Hasta min_sv
        // guardar movimiento en memoria de
            largo plazo
        tabu_largo[x,y)]=+ 1
    Iterar Hasta lmin_sv
        // si no está prohibido por memoria de
            corto plazo y largo plazo
        Si tabu_corto[x,y] == 0 & tabu_largo[x,y
            ] >= umbral_largo_plazo
            return {fit, sol, tabu_largo}
        Otro
            // si está prohibido por memoria de
                largo plazo
            Si tabu_largo[x,y] >=
                umbral_largo_plazo
                min *= penalizar
            Si min < fit
                temporal = min
    return {fit, sol, tabu_largo}
Fin BuscarVecinosTabu

Nota params[]* se hace referencia a todos los
    parámetros de entradas necesarios para
    ajustar la experimentación.
```

B. Parámetros de ajuste

A continuación, se describen los parámetros que utilizará la metaheurística implementada en la presente experiencia:

Variable	Descripción
Tiempo Local	Corresponde al número de iteraciones que va a realizar para buscar un conjunto de vecinos.
Tiempo Espacio	Corresponde al número de iteraciones para realizar una búsqueda diversificada.
Intensidad	Valor que indica que el tamaño o tasa de cambio del operador.
Memoria Corta	Tiempo de iteraciones no permitidas para considerar un movimiento.
Memoria Larga	Tiempo de iteraciones como umbral, en la que se comenzará a prohibir el movimiento, hasta el fin del algoritmo.
Penalización	Valor porcentual por la cual se va a decrementar el valor fit encontrado.
NVecinos	Corresponde al número de vecinos a considerar en la búsqueda local.

Tabla III: Descripción de parámetros de ajustes para el algoritmo de Tabu Search.

C. Operadores utilizados

A continuación, se describen los operadores a partir de los cuales se definirá la vecindad en el problema y se realizarán las pruebas respectivas.

C.1 Swap

Para este operador se aplica un pequeño algoritmo, con el fin de generar una colección de soluciones vecinas, de forma ordenada e intensificada.

Listing 4: Algoritmo Swap

```
    Iterar Hasta intensidad_maxima
        Iterar Hasta n_vecinos
            // i posición pivote y la intensidad
                , corresponde contra que posici
                ón.
            // se va a realizar el cambqio de
                elemento.
            vecino = swap(i, i+intensidad)
        Fin Iterar
    Fin Iterar
```

Un ejemplo, es considerar un arreglo de soluciones como en (8)

$$s = [a, b, c, d, e, f] \tag{8}$$

Luego para la primera iteración, se tendrá una intensidad 1, por lo que se tendrán soluciones vecinas como en (9).

$$s1 = [b, a, c, d, e, f] s2 = [a, c, b, d, e, f] s3 = [a, b, d, c, e, f] ... \tag{9}$$

Posteriormente para una intensidad mayor, se va a considerar un swap de largo 2 o más, como en el

ejemplo (12).

$$s1 = [c, b, a, d, e, f] s2 = [a, d, c, b, e, f] s3 = [a, b, e, d, c, f] \quad (10)$$

C.2 Insert

El segundo operador que será utilizado será el operador insert, el cual opera de la siguiente manera:

1. Se elige un elemento, el cual se mueve a otra posición cualquiera de la lista.
2. Dependiendo de en que sentido se mueva el elemento, el resto de los elementos se acomodan en el sentido contrario con el fin de ocupar el espacio vacío.

A continuación se muestran dos ejemplos de la utilización del operador insert, aplicados al arreglo (8):

1. Si se mueve el segundo elemento (b) a la quinta posición, este es el resultado:

$$s1 = [a, c, d, e, b, f] \quad (11)$$

2. Si se mueve el cuarto elemento (d) a la primera posición, este es el resultado:

$$s2 = [d, a, b, c, e, f] \quad (12)$$

VI. Resultados y Análisis

A. Métodos clásicos

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

Tabla IV: 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 IV, 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].

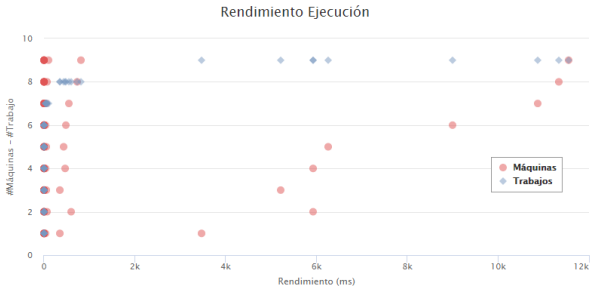

Fig. 1: Rendimiento de las ejecuciones, para diferentes cantidad de máquinas y trabajos, con límite de 9 trabajos.

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

De la tabla V, 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 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 V. 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 V, 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.

B. Metaheurística

A continuación se presenta una recopilación de los resultados obtenidos para cada uno de los operadores de la meta heurística de Tabu Search.
En donde: Tloc = Tiempo local, Tesp = Tiempo espacio, Int = Intensidad, MC = memoria corto plazo, ML = Memoria largo plazo, Pen = Penalización, Nvec = Número de vecinos, T. Ex. = Tiempo Excedido.

En la tabla VI, se presentan los resultados con el operador swap para una data de 20x20. Como se puede observar, los valores del fitness oscilan entre 2485 y 2636.

Tloc	Tesp	Int	MC	ML	Nvec	Fit
3	10	10	5	10	10	2644
3	10	10	5	10	20	2546
3	10	10	5	10	40	2598
10	10	10	5	10	25	2636
20	10	10	5	10	25	2628
40	10	10	5	10	25	2485
3	10	10	5	10	25	2546
3	10	20	5	10	25	T.exc
3	10	40	5	10	25	T.exc
3	10	10	10	10	25	2546
3	10	10	20	10	25	2546
3	10	10	40	10	25	2630
3	10	10	5	12	25	2631
3	10	10	5	20	25	2627
3	10	10	5	40	25	2616
3	10	15	5	10	25	2546
3	10	20	5	10	25	2535
3	10	40	5	15	25	2535

Tabla VI: Tabla de resultados Tabu Search con operador swap, para un dataset de 20x20, para diferentes parámetros de entrada.

En la tabla VII, se presentan los resultados con el operador insert para una data de 20x20. Como se puede observar, los valores del fitness oscilan entre 2685 y 2867.

De las tablas VI y VII, podemos notar que para diferentes configuraciones, el fitness del método varía entorno al $\sim 7\,\%$ para ambos operadores.
En general los procesos se ejecutaron entre 2 a 5 segundos por cada prueba, sin embargo, para ciertas configuraciones, el algoritmo quedaba congelado o el tiempo de espera se hacía muy prolongado, como lo fue en los casos de intensidades altas.
Se observa una mejora al momento de modificar la memoria de corto plazo para el caso del operador swap ($\sim 3\,\%$), esto es, aumentar la restricción por iteraciones en una búsqueda local, por otro lado para el operador insert, no se observa una mejora significativa. En general, las mejoras radicales se obtienen cuando se intensifica la búsqueda acorde al parámetro *int*, y para tiempos pequeños en cuanto a la búsqueda local y espacial. Aumentar el número de vecinos como cota superior, no se evidenció como parámetro relevante de mejora.

Tloc	Tesp	Int	MC	ML	Nvec	Fit
3	10	10	5	10	10	2770
3	10	10	5	10	20	2786
3	10	10	5	10	40	2847
10	10	10	5	10	25	2812
20	10	10	5	10	25	2780
40	10	10	5	10	25	2810
3	10	10	5	10	25	2746
3	10	20	5	10	25	T.exc
3	10	40	5	10	25	T.exc
3	10	10	10	10	25	2685
3	10	10	20	10	25	2671
3	10	10	40	10	25	2712
3	10	10	5	12	25	2780
3	10	10	5	20	25	2763
3	10	10	5	40	25	2812
3	10	15	5	10	25	2852
3	10	20	5	10	25	2867
3	10	40	5	15	25	2834

Tabla VII: Tabla de resultados Tabu Search con operador insert, para un dataset de 20x20, para diferentes parámetros de entrada.

En este apartado se consideraron diversos tamaños de dataset, 5x20, 10x20, 20x20 y 50x10. En cada uno de estos se varían los parámetros antes descritos, como por ejemplo el tamaño de la vecindad de búsqueda, tiempo de búsqueda, tamaños de memoria de corto y largo plazo, etc. A continuación se mostrarán algunos resultados relevantes de la experimentación.

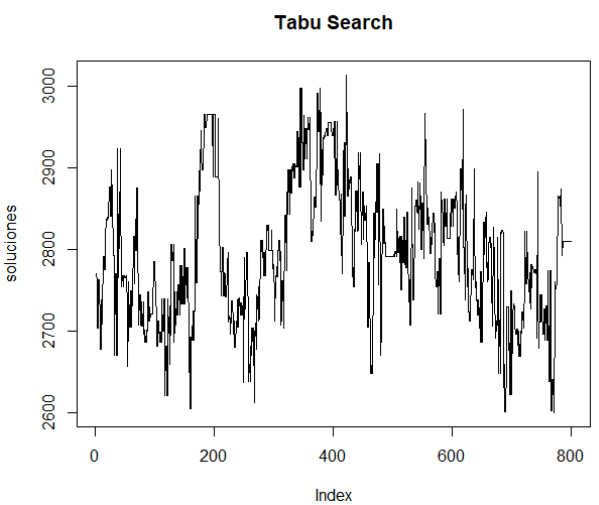

Fig. 2: Perfil de soluciones para TS con operador swap, para un dataset de 20x20, con parámetros de configuración: $tiempoLocal = 5$, $tiempoEspacio = 8$, $intensidad = 20$, $mCorto = 5$, $mLargo = 50$, $penalizacion = 0{,}5$ y $nVecinos = 20$. Mejor solución encontrada = 2600.

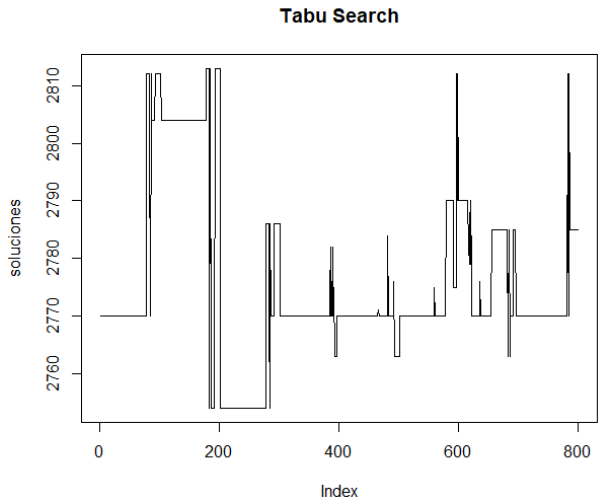

Fig. 3: Perfil de soluciones para TS con operador insert, para un dataset de 20x20, con parámetros de configuración: $tiempoLocal = 5$, $tiempoEspacio = 8$, $intensidad = 20$, $mCorto = 5$, $mLargo = 50$, $penalizacion = 0{,}5$ y $nVecinos = 20$. Mejor solución encontrada = 2754.

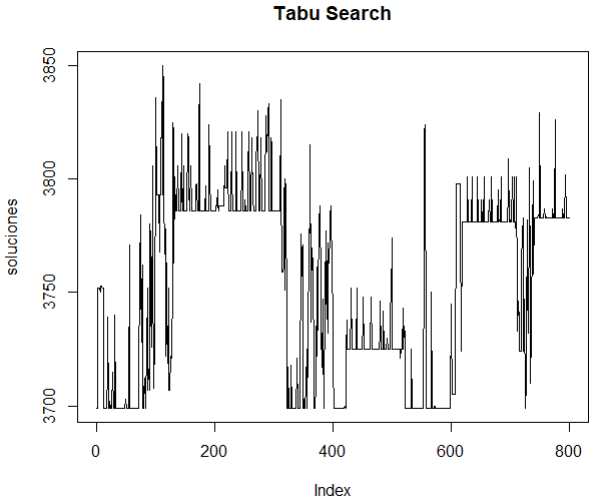

Fig. 5: Perfil de soluciones para TS con operador insert, para un dataset de 50x10, con parámetros de configuración: $tiempoLocal = 5$, $tiempoEspacio = 8$, $intensidad = 20$, $mCorto = 5$, $mLargo = 50$, $penalizacion = 0{,}5$ y $nVecinos = 20$. Mejor solución encontrada = 3699.

De acuerdo a la experimentación en E. Taillard, "Benchmarks for basic scheduling problems", el dataset de 20x20, se tiene que el rango de resultados por cota superior e inferior es de 2297 y 1911 respectivamente, por lo que de acuerdo a lo obtenido en esta experimentación se tiene una diferencia del 23 % para el operador swap y un 31 % para el operador insert.

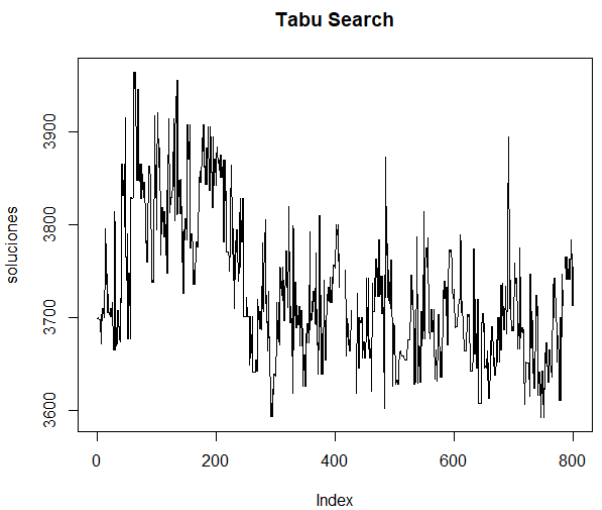

Fig. 4: Perfil de soluciones para TS con operador swap, para un dataset de 50x10, con parámetros de configuración: $tiempoLocal = 5$, $tiempoEspacio = 8$, $intensidad = 20$, $mCorto = 5$, $mLargo = 50$, $penalizacion = 0{,}5$ y $nVecinos = 20$. Mejor solución encontrada = 3593.

De acuerdo a la experimentación en E. Taillard, "Benchmarks for basic scheduling problems", el dataset de 50x10, se tiene que el rango de resultados por cota superior e inferior es de 3025 y 2907 respectivamente, por lo que de acuerdo a lo obtenido en esta experimentación se tiene una diferencia del orden de 20 % aproximadamente.

VII. Discusión

Mediante la aplicación de los métodos de optimización clásicos 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 este tipo de métodos 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, que operan con rendimientos diversos sujetos a los tamaños 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. Sin embargo, ninguno de los dos métodos es capaz de entregar un resultado para instancias más grandes (más de 10 trabajos).

En cambio, haciendo uso de una metaheurística como tabu search, es posible obtener soluciones computacionales rápidas para instancias grandes (por ejemplo, 20 trabajos y 20 máquinas).

La variación de los diferentes parámetros seleccionados en el presente algoritmo han demostrado ejercer diversa influencia en los resultados obtenidos, teniendo algunos parámetros que varían de manera significante el fitness de la solución, y por otro lado, algunos que no ejercen gran variación.

A pesar de que una metaheurística, de acuerdo a su definición, podría no entregarnos la solución óptima al problema, es posible obtener soluciones factibles, bien aproximadas y de buena calidad que podrían ser suficientes para apoyar la toma de decisiones en un escenario real, por ejemplo en la industria.

Por lo tanto, para el problema del flow shop scheduling, al trabajar con instancias grandes, se prefiere

el uso de metaheurísticas por sobre la resolución por métodos de optimización clásicos, debido a su gran ventaja respecto a la utilización de recursos y tiempo de ejecución.
Es importante destacar que para este experimento, el algoritmo tabu search fue diseñado y construído en base a los conocimientos de programación de los redactores de este documento, por lo que de cierta forma, está sujeto a posibles errores de lógica, de optimización o algún otro tipo de factor no trivial de identificar, que pueda influenciar en la calidad de los resultados obtenidos.

A. Propuesta de mejoras

En la experimentación, al momento de ejecutar el algoritmo de Tabu Search desarrollado en R, cabe destacar que se produjeron diversos errores de ejecución, relacionadas con el control de borde de las matrices y arreglos de los datos, control de decisiones no apropiados con la lógica, movimientos incoherentes, entre muchos otros elementos.
Además se logra percibir que el algoritmo en ocasiones no explora lo suficientemente bien, como también se logra evidenciar que se repite la misma búsqueda reiteradas ocasiones, a pesar de variar los parámetros de entrada.
Para el lector, se sugiere considerar en dar un mejor enfoque en el modo que se construye la lista tabu de los movimientos restringidos, en especial la memoria de corto plazo. La dificultad se presenta al momento de realizar movimientos que intercalan elementos que salen del largo de arreglo y se debe reescribir la posición, como por ejemplo, considerar una solución del tipo presentado en (8), luego de acuerdo al algoritmo, si se intensifica el swapping al punto de exceder el largo de la solución, se debe reescribir la posición a un lugar coherente. Este tipo de consideraciones no son triviales, dado que el movimiento 1-3 no es lo mismo que 3-1, por lo que es importante añadir este componente de forma de ampliar los espacios de búsqueda.
Ahora bien, otro punto a considerar es el método de restricción a utilizar, se podría probar con crear listas tabú respecto a soluciones o rango de soluciones, en vez de la prohibición de movimientos.
Por otro lado, se podría mejorar el modo de operar sobre la representación de una solución, dado que en el algoritmo utilizado se opera sobre una matriz como en la representación en la tabla II.

VIII. Conclusiones

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

1. El Flow Shop Sheduling es un problema importante en el área de investigación de operaciones. A pesar de la gran literatura existente sobre el tema, el FSS sigue siendo un tópico vigente el día de hoy en cuanto a la búsqueda de mejores soluciones.
2. 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 o R) u homoíconicos (como Julia + Jump).
3. Es posible resolver instancias pequeñas del problema de Flow Shop Scheduling utilizando soluciones computacionales basadas en métodos de optimización clásicos. En el presente documento, se planteó utilizar enumeración exhaustiva y branch and bound. Sin embargo, se demostró que estos métodos se hacen insuficientes al aumentar el tamaño de trabajos y máquinas.
4. Para instancias más grandes, se hace necesario resolver el problema usando metaheurísticas. En el presente trabajo, se diseñó un algoritmo basado en tabu search, el cual fue probado con dos operadores diferentes (swap e insert) y cambiando los parámetros para variar el proceso de búsqueda por el espacio de soluciones. El algoritmo fue capaz de entregar buenas soluciones en tiempos cortos.
5. Se logra evidenciar que los algoritmos metaheurísticos, son capaces de resolver problemas de alta complejidad y que pueden reducir significativamente los tiempos de procesados, sin embargo, el diseño e implementación de esta, debe ser revisada con mayor expertiz y ser sometida a test para evaluar la calidad, no sólo de los resultados, si no más bien de la mismo desarrollo, con el objeto de validar el instrumento de medición.

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 per-*

mutation 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.