Trabajo Fin de Máster
Máster en Organización Industrial y Gestión de Empresas

Metaheurísticas para el problema Distributed Flow shop con permutación y objetivo Total Core Idle Time

Autor: Marta Ariza Gamero
Tutor: Paz Pérez González

Dpto. de Organización Industrial y Gestión de Empresas I
Escuela Técnica Superior de Ingeniería

Sevilla, 2023

Trabajo Fin de Máster
Máster en Organización Industrial y Gestión de Empresas

Metaheurísticas para el problema Distributed Flow shop con permutación y objetivo Total Core Idle Time

Autor:
Marta Ariza Gamero

Tutor:
Paz Pérez González
Profesor titular

Dpto. de Organización Industrial y
Gestión de Empresas I
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla
Sevilla, 2023

Trabajo Fin de Máster: Metaheurísticas para el problema Distributed Flow shop con permutación y objetivo Total Core Idle Time

Autor: Marta Ariza Gamero

Tutor: Paz Pérez González

El tribunal nombrado para juzgar el Proyecto arriba indicado, compuesto por los siguientes miembros:

Presidente:

Vocales:

Secretario:

Acuerdan otorgarle la calificación de:

Sevilla, 2023

El Secretario del Tribunal

ÍNDICE

1 INTRODUCCIÓN

El primer capítulo del proyecto en estudio busca contextualizar el problema en estudio, dando a conocer qué es la Programación de la Producción y su función en la producción así como la terminología empleada en el resto del documento. Expone, además, la estructura del estudio y objetivo del mismo.

1.1 Contexto

La Programación de la Producción es el proceso de toma de decisiones que asigna recursos a tareas en periodos de tiempo determinados y busca optimizar uno o varios objetivos (Pinedo, 2012). Los recursos en el marco de la fabricación se encuentran en forma de máquinas en un entorno de trabajo y las tareas son los trabajos en un proceso productivo, es decir, la Programación de la Producción es la asignación de máquinas para la fabricación de un conjunto de trabajos. En otros ámbitos, los recursos se pueden considerar pistas de aterrizajes en un aeropuerto o unidades de procesamiento en centros informáticos y las tareas serían los despegues/aterrizajes o ejecuciones de programas, respectivamente.

La Programación de la Producción es una decisión a corto plazo que forma parte de la Gestión de la Producción (*Figura 1*). Cada una de las decisiones, bien sea a corto, medio o largo plazo, busca la mayor calidad en el producto/servicio al mismo tiempo que se minimizan costes y plazos de entrega. Al más alto nivel en la Gestión de la Producción está la Planificación Estratégica de la Red en la que se toman decisiones estratégicas para establecer los proveedores, la localización y entornos de las plantas, entre otras. Las decisiones estratégicas son aquellas de mayor impacto en la actividad y definirán la capacidad y distribución de recursos con los que cuenta a lo largo de su vida. Tras las decisiones estratégicas a largo plazo, se plantean decisiones tácticas a medio plazo: la Planificación Agregada determina en plazos de semanas o meses los recursos necesarios (Planificación de las necesidades de material) para satisfacer la demanda estimada (Planificación de la Demanda) por grupos o familias sin entrar en mayor detalle. Partiendo de las limitaciones y alcances definidos en el Plan Agregado de Producción, y conociendo las Necesidades de Materia Prima y la Planificación de la producción se estudia la Programación de la Producción. A pesar de ser una decisión a corto plazo y tener un ciclo de vida reducido, es una decisión compleja, relativamente estructurada y con bastante relevancia. Algunos autores hablan de ella como un proceso de programación continuo.

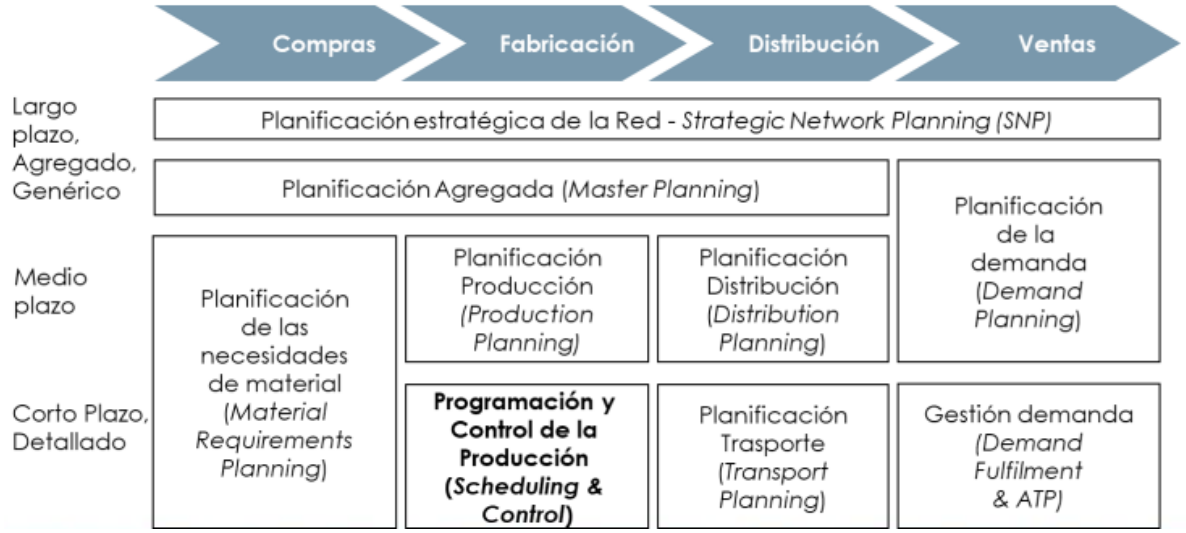


Figura 1. Gestión de la Producción (Framinan et al., 2014)

Un programa (*schedule*) es el resultado de las decisiones tomadas en la Programación. Dado un periodo temporal, el programa define el momento en el que una máquina comienza y finaliza un trabajo. En la mayoría de los casos, cuando un trabajo empieza a ser procesado, la operación no se puede interrumpir, por lo tanto, como el tiempo de la operación es conocido, solo es necesario conocer el instante en el que el trabajo entra o sale de dicha máquina. Sin embargo, no todos los programas existentes pueden ser ejecutados en la realidad. Si cumple con las limitaciones, restricciones y características del proceso productivo, entonces el programa es admisible (*feasible schedule*).

La variedad de programas admisibles en un modelo es amplia, por lo que hay que establecer un criterio para elegir entre ellos. En la mayoría de las tomas de decisiones, el programa admisible elegido es semiactivo (*semi-active schedule*), es decir, los trabajos no se pueden adelantar sin cambiar el orden en que alguna máquina procese cualesquiera de los trabajos.

Para definir un modelo de programación es necesario tener cierta información básica que limite el área en estudio como la distribución de la planta, el número de trabajos a procesar, el tiempo de procesamiento en máquinas y el número de máquinas con las que trabajar. Siguiendo la notación de (Framinan et al., 2014):

- $N=\{1,..,n\}$ denota al conjunto de trabajos o tareas a procesar. Los subíndices j, k hacen referencia a los trabajos.
- $M=\{1,...,m\}$ denota al conjunto de máquinas o recursos donde se procesan los trabajos. El subíndice i hace referencia a las máquinas.
- El tiempo de proceso, p_{ij}, es el tiempo que la máquina i tarda en procesar el trabajo j. También se puede ver como el tiempo que la máquina i está ocupada por el trabajo j.

Los modelos de programación, comúnmente, se definen mediante la tripleta α | β | γ propuesta por (Graham et al., 1979).

El entorno α es un campo único que indica la disposición de las máquinas y el número de ellas en la planta. Los entornos más comunes son una máquina (α = 1), máquinas paralelas, diferenciándose entre ellas por los tiempos de proceso (idénticas α = P_m, uniformes α = Q_m o no relacionadas α = R_m) y máquinas en serie o taller, diferenciándose por la ruta que siguen por cada una de las máquinas (taller de flujo regular si tienen la misma ruta α = F_m, taller de trabajo si la ruta es propia del trabajo α = J_m, taller abierto si no hay ruta α = O_m).

Este estudio se centra en el taller de flujo distribuido o *Distributed Flow shop* α = DF_m. En él un conjunto de fábricas idénticas $F=\{1,...,f\}$ se comportan como talleres de flujo iguales e independientes. El capítulo 1.1 Entorno *Distributed Flow shop* entra en más detalle del entorno.

El campo β indica las limitaciones de las máquinas o los trabajos para ser procesados. Pueden entrar una o varias restricciones, e incluso puede no existir ninguna más allá de las generales (capítulo 2.1.Restricciones del problema). Las restricciones relacionadas con tiempos son las más frecuentes en la literatura, ya sean ociosos en máquinas (β = *no-idle*) o de espera de los trabajos entre máquinas (β = *no-wait*). Para el estudio se ha tenido en cuenta que el taller sea regular de permutación (β = *prmu*), es decir, la secuencia de trabajos en cada una de las máquinas es el mismo.

El objetivo γ es la función a optimizar que, en general, están enfocadas al tiempo, coste y calidad buscando el equilibrio entre estos (*Figura 2*).

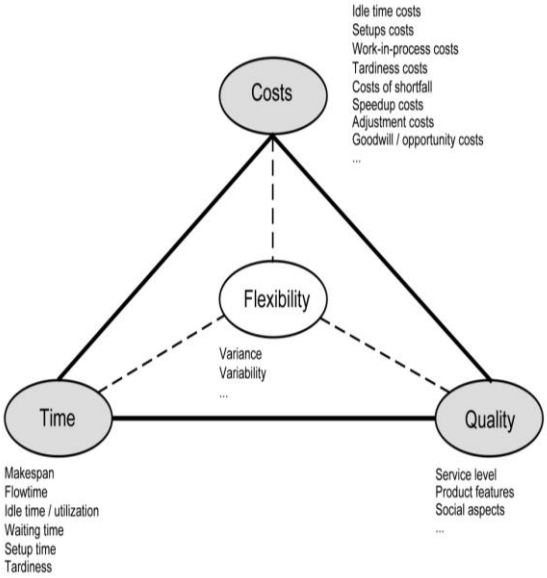


Figura 2. Clasificación de los objetivos (Framinan et al., 2014)

El objetivo más extendido es el *makespan* $\gamma = C_{max}$, es decir, el tiempo que tarda un programa desde que comienza a procesar el primer trabajo hasta que finaliza el último trabajo secuenciado. La suma de los tiempos de finalización de cada trabajo se conoce como *Total Completion Time* $\gamma = \Sigma C_j$ y a la suma de tiempos que un trabajo está en el sistema como *Total Flow Time* , estando ambos objetivos bastante extendidos. Este estudio en concordancia con el *makespan* y la restricción *no-idle*, busca minimizar el *Core Idle Time*, definido en el capítulo 2.2, Objetivo Core Idle Time.

1.2 Estructura del documento

El problema *Distributed Permutation Flow sho*p con objetivo *Total Core Idle Time,* $DF_m \mid prmu \mid \sum CIT$, utilizando la tripleta anteriormente definida, es el objeto de análisis de este documento. Para ello, se ha introducido la Programación de la Producción y la nomenclatura empleada. En el capítulo 2, Descripción del problema, se explica en detalle el entorno, restricciones y objetivo del problema después de esta pequeña contextualización. En el capítulo 3, Revisión de la literatura, se hace un análisis del estado-del-arte en cuanto al problema en cuestión y las distintas conclusiones obtenidas en los artículos relacionados con el fin de que sean adaptados a problema en estudio. Se clasifican los algoritmos para la resolución de los problemas en dos capítulos y se detalla cómo han sido adaptados: Heurísticas constructivas para la resolución del problema y Metaheurísticas para la resolución del problema. En el penúltimo capítulo, Resultados obtenidos, se analiza y discuten cada una de las soluciones aportadas para llegar a ciertas conclusiones en el último capítulo, Conclusiones.

1.3 Objetivo del proyecto

Este Proyecto busca identificar métodos más eficientes para la del problema $DF_m \mid prmu \mid \sum CIT$ partiendo de los métodos desarrollados anteriormente en el Trabajo Fin de Grado (TFG) (Ariza Gamero, 2021). En él se estudiaron un total de veintiséis heurísticas constructivas, expuestas en *Tabla 1*, generadas a partir de los métodos de resolución adaptados en la literatura que habían generado los mejores resultados para entornos similares, además de nuevas heurísticas propuestas. Finalmente, en el TFG se concluyó que la adaptación de una de las heurísticas constructivas más usadas en la literatura sobre programación de la producción, conocida como NEH (R_1, A_4), genera los valores más eficientes para la función objetivo. Estos resultados serán comparados en el proyecto con las cuatro heurísticas planteadas, siendo una de ellas una mejora del NEH (R_1, A_4).

Este proyecto propone métodos aproximados avanzados, metaheurísticas, con el objetivo de proporcionar los mejores valores del $\sum CIT$ de los obtenidos con las heurísticas constructivas expuestas en *Tabla 1*.

Heurísticas constructivas estudiadas en TFG	***Nuevas heurísticas constructivas planteadas***
DLR	DLR-DNEH
DNEH	NEH2
NEH (R_1, A_i)	NEH2_en
NEH (R_2, A_i)	NEH (R_1, A_4)_en
DNEH (R_1, A_i)	
DNEH (R_2, A_i)	

Tabla 1. Heurísticas constructivas estudiadas y nuevas planteadas

2 DESCRIPCIÓN DEL PROBLEMA

El problema en estudio se detalla a continuación, indicando el entorno, sus restricciones y el objetivo a optimizar.

1.1 Entorno Distributed Flow shop

El taller de flujo o *Flow shop* es un entorno tipo de taller en el que las máquinas se encuentran en serie y cada uno de los trabajos del taller pasa por cada una de las máquinas en el mismo orden: primero por la máquina 1, seguido de la máquina 2 y así hasta completar las *m* máquinas que componen el taller.

El entorno *Distributed Flow shop* es una generalización del *Flow shop* clásico como consecuencia de los entornos multi-fabrica. En un entorno Distributed se encuentran un conjunto de *F* fábricas idénticas y cada una de ellas se comporta como un taller de flujo donde el número de máquinas y tiempos de procesos no difieren entre ellas. Los entornos multi-fábrica han ido dejando atrás a los entornos con una sola fábrica (Fernandez-Viagas & Framinan, 2015). En ellos, se reducen riesgos y costes al mismo tiempo que se genera calidad en el producto.

En cualquier contexto multi-fábrica, cada trabajo *j* puede ser asignado en cada una de las *F* fábricas ya que la distribución es la misma y el número y orden de máquinas no varía al igual que el tiempo de proceso p_{ij}. Exige además que, si un trabajo es asignado a una fábrica, no sale de ella hasta que no ha sido procesado al completo. En estos problemas, la asignación trabajo-fábrica forma parte de la toma decisión del problema al igual que las secuencias en cada una de las fábricas.

2.1 Restricciones del problema

El programa dado para el modelo en estudio debe cumplir con la restricción de permutación β = *prmu* explicada anteriormente en el capítulo 1.1: los trabajos pasan por cada una de las máquinas en el mismo orden en todas las fábricas del entorno.

No obstante, además de la restricción *prmu,* el modelo se limita por las suposiciones generales de (Naderi & Ruiz, 2010) para los entornos *Flow shop:*

- Todos los trabajos son independientes y están al principio del horizonte temporal de la programación.
- Las máquinas siempre están disponibles (*no breakdowns*).
- Cada máquina solo puede procesar un trabajo en el mismo momento.
- Cada trabajo solo puede ser procesado por una máquina en el mismo momento.
- Cuando un trabajo empieza a ser procesado en una máquina, no se puede interrumpir hasta que no ha finalizado (*no preemption*).
- Los tiempos de preparación y transporte están incluidos en los tiempos de proceso o son despreciables.
- El buffer entre máquinas se considera infinito.

2.2 Objetivo Core Idle Time

En entornos tipo taller es común tener en cuenta consideraciones que no permitan tiempos ociosos (*idle time*) en las máquinas cuando se trata de máquinas caras que forman parte del diseño del *layout* (Maassen et al., 2020).

El tiempo ocioso de una planta es un indicador de la eficiencia del sistema productivo: un flujo continuo en un entorno es sinónimo que nada interrumpe el proceso productivo, minimizando así el desperdicio temporal y económico que supone cualquier parada en el sistema.

No obstante, la restricción β=*no-idle,* que no permite tiempos ociosos en las maquinas, es una limitación muy fuerte que conviene relajar siempre que sea posible y no sea limitante para el problema en estudio, como ocurre en la industria alimentaria. El *idle time* de una máquina se puede dividir en los siguientes tres componentes, según ocurran en el trascurso temporal del proceso productivo:

- *Front Idle Time* de la máquina *i,* FIT_i: tiempo que la máquina *i={2,…, m}* espera a procesar el primer trabajo programado.
- *Core Idle Time* de la máquina *i,* CIT_i: tiempo que una máquina *i={2,…, m}* espera entre dos trabajos consecutivos.
- *Back Idle Time* de la máquina *i,* BIT_i: tiempo que la máquina *i={1,…, m-1}* espera a que todos los trabajos de procesados.

En la *Figura* 3 se encuentra un ejemplo donde se identifican, mediante un diagrama de Gantt, los distintos tiempos ociosos de un taller de flujo. Cabe destacar que en la primera máquina del entorno solo se puede aparecer BIT_1, ya que el programa considera que todos los trabajos se encuentran disponibles al principio del horizonte temporal y se trabaja con programas semiactivos. La última máquina del taller, por otro lado, no puede generar BIT_m ya que el programa finaliza cuando el último trabajo es procesado en la última máquina.

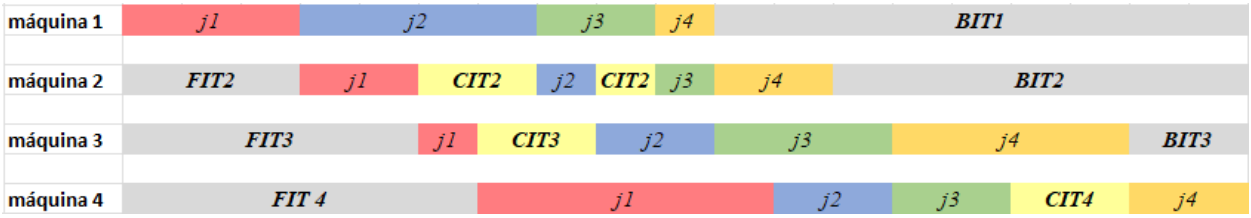


Figura 3. Idle time en un entorno Flow shop

Si la restricción *no-idle* hubiera sido impuesta desaparecería el CIT_i de las máquinas, pero seguiría BIT_i y FIT_i ya que no es posible eliminar estos tiempos ociosos. Sin embargo, el tiempo anterior y posterior en el que una máquina empieza a trabajar es tiempo que, aunque no genere valor al proceso productivo, es tiempo que las máquinas precisan para su mantenimiento y limpieza a diferencia del CIT_i que es tiempo desperdiciado en producción. Este proyecto considera la minimización del CIT_i para todas las máquinas como objetivo sostenible siendo este la relajación del problema estudiado en la literatura por (Ruiz et al., 2009) con objetivo *makespan.* Además, (Maassen et al., 2020) demuestra que los métodos eficientes empleados para este problema son aplicables al problema objeto de estudio generando soluciones de calidad.

2.3 Cálculo del objetivo Total Core Idle Time para el problema propuesto

.Para una mayor comprensión del cálculo del objetivo, se expone a continuación un ejemplo sencillo del DPFSP con $f = 2$ fábricas, $m = 4$ máquinas, $n = 7$ trabajos obtenido de (Ariza Gamero, 2021). Los tiempos de proceso p_{ij} se encuentran en la *Tabla* 2.

		trabajo j						
		1	2	3	4	5	6	7
máquina i	1	3	4	2	1	2	3	3
	2	2	1	1	2	4	3	3
	3	1	2	3	4	4	5	1
	4	5	2	2	2	2	1	2

Tabla 2. Tiempos de proceso para el ejemplo dado

El programa del ejemplo se muestra en *Figura 4*: en la fábrica 1 se secuencia los trabajos (*j1, j2, j3, j4*) mientras que en la fábrica 2 se secuencian (*j5, j6, j7*). Este programa genera un *makespan* de 19 u.t., instante en el que finaliza el trabajo *j4* en la máquina 4 de la fábrica 1 siendo este el trabajo crítico. Además, teniendo en cuenta que se trabaja con programas semiactivos el *Total Completion Time*, es decir, la suma de los tiempos de finalización de los trabajos es igual al *Total Flow Time*, ya que todos los trabajos están disponibles al principio del horizonte temporal de la programación. El cálculo se encuentra en *Tabla* 4: en este programa el *Total Flow Time* es 104 u.t.

Tras conocer el *makespan* del problema y *Total Flow Time*, ahora se pasa a analizar el cálculo del objetivo del problema. En primer lugar, la fábrica 1 el trabajo *j1* pasa inmediatamente de la máquina 1 a la máquina 2 y así sucesivamente ya que se está trabajando con programas semiactivos. Lo mismo ocurre con el trabajo *j5* en la fábrica 2 ya que es el primer trabajo secuenciado. De esta manera, se puede apreciar que el primer trabajo secuenciado en una fábrica nunca va a generar *Core Idle Time* si se trabaja con programas semiactivos. Por este mismo motivo, la primera máquina de un taller en serie no genera *Core Idle Time*. Se ve cómo en el resto de las máquinas se generan tiempos ociosos debido a que las máquinas esperan que los trabajos terminen sus procesos anteriores.

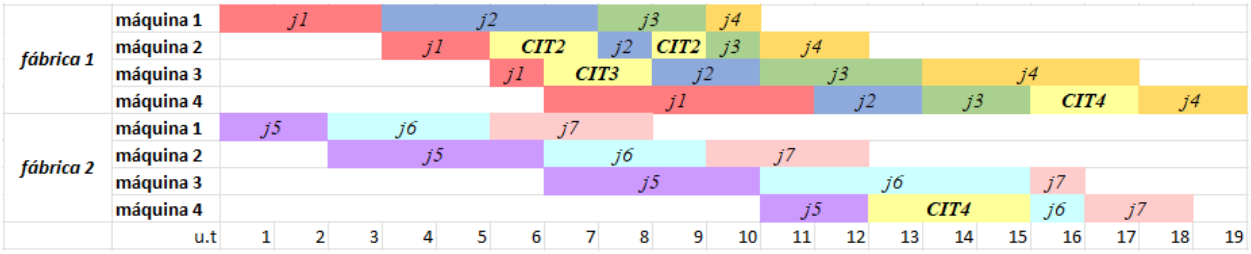


Figura 4. Diagrama de Gantt para el programa dado

El *Core Idle Time* de una fábrica es el tiempo ocioso generado por cada una de sus máquinas. En el programa dado, se puede ver en la Tabla 4 que la fábrica 1 genera 7 u.t. mientras que la fábrica 2 genera 3 u.t. Es decir, se ha generado un total de 10 u.t en las que las máquinas están ociosas.

trabajo j	Completion time j
j1	11
j2	13
j3	15
j4	19
j5	12
j6	16
j7	18
Total Completion Time	104

Tabla 3. Cálculo Total Completion Time para el programa dado

	máquina 1	máquina 2	máquina 3	máquina 4	CIT por fábrica
fábrica 1	0	3	2	2	7
fábrica 2	0	0	0	3	3
				CIT total	10

Tabla 4. Cálculo del Total Core Idle Time para el programa dado

Si se secuencian los trabajos (*j5, j1, j6, j3*) en la fábrica 1 y los trabajos (*j2, j4*) en la fábrica 2, la máquina 4 de la fábrica 2 solo transcurren 2 u.t. en la que esté ociosa. Sin embargo, ahora el *makespan* aumenta respecto al programa anterior: pasan a ser 22 u.t. ; al igual que aumenta el *Total Flow Time*, que son 111 u.t.. El programa se aprecia en la Figura 5.

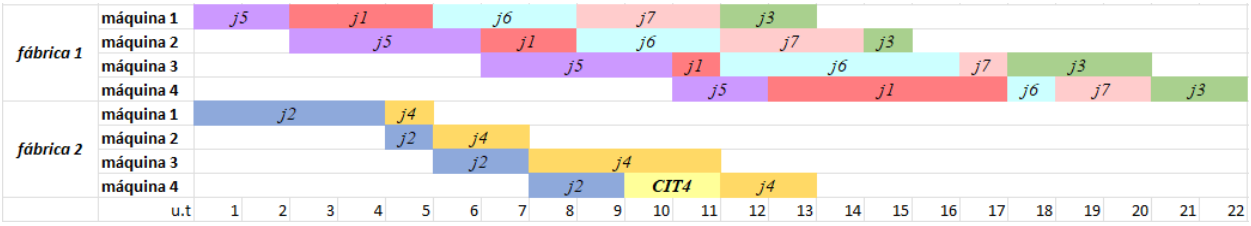


Figura 5. Diagrama de Gantt para un nuevo programa

3 REVISIÓN DE LA LITERATURA

El entorno tipo taller de flujo regular o *Flow shop* es uno de los entornos de fabricación más antiguos y revisados en la literatura (Johnson, 1954). En él, todos los trabajos siguen una misma ruta de fabricación, pasando primero por la máquina 1, seguido de la máquina 2 y continúa hasta la máquina *m* siendo esta la última que completa el taller. En este capítulo se revisará el estudio de este entorno y la evolución a lo largo de la historia.

Esta revisión se centra en el flujo regular de permutación (*prmu*), conocido como *Permutation Flow shop*. En el problema *Permutation Flow shop* (PFSP) (Nawaz & Enscore, 1983), además de que los trabajos tienen una ruta determinada, el orden por el que los trabajos pasan por cada una de las máquinas se exige que sea el mismo.

Los entornos tipo taller se han visto limitado por distintas restricciones, además de las suposiciones generales del taller (Graham et al., 1979). Los modelos pueden exigir que no existan tiempos ociosos entre las máquinas (*no-wait*) (Gangadharan & Rajendran, 1993) o que los trabajos no puedan esperar a ser procesados una vez entran en el flujo (*no-idle*) (Woollam, 1986).

Como consecuencia de la deslocalización, del aumento de la competencia y de avances en la logística, los modelos con una única fábrica han desaparecido gradualmente con los años. Es así como, adaptándose a la producción real, la literatura ha ido de la mano de la evolución industrial y el problema *Distributed Permutation Flow shop* (DPFSP) (Naderi & Ruiz, 2010) , comparado con otros entornos, se ha convertido en uno de los más estudiados a pesar de ser bastante reciente. En el DPFSP, la asignación de trabajos a fábricas se convierte en una decisión de peso en el problema. Todos los trabajos pueden ser asignados a todas las fábricas, ya que cada una de ellas se convierte en un entorno de flujo independiente al resto en los que los tiempos de fabricación no varían.

El DPFSP ha sido revisado con los objetivos clásicos de la literatura como es el *makespan*. (Naderi & Ruiz, 2014) plantea para el objetivo *makespan* un modelo *Scatter Search* y de manera paralela (Fernandez-Viagas & Framinan, 2015) plantea un algoritmo *Iterated Greedy*, que al ser estudiado de manera simultánea en su momento no fueron comparados. No obstante, no se ha determinado el mejor método para resolver el problema ya que a pesar de ser que fueran analizados a posteriori por (Ruiz et al., 2019), junto al *Iterated Greedy Two Stage* propuesto en el problema, se generan resultados similares sin obtener un método claro.

El *Total Completion Time* es otro de los objetivos clásicos revisados en la literatura. El mejor método de resolución lo obtiene (Pan et al., 2019) al comparar el *Evolutionary Algorithm de* (Fernandez-Viagas et al., 2018) junto a sus cuatro metaheurísticas (*Iterated Greedy*, *Iterated Local Search*, *Scatter Search* and *Artificial Bee Colony*). Finalmente se concluyó que el *Iterated Local Search* generaba las mejores soluciones.

No obstante, el estudio del problema ha crecido exponencialmente (Perez-Gonzalez & Framinan, 2024) y, además de los objetivos anteriormente citados, entre la literatura se encuentran objetivos como el *Tardiness* (Khare & Agrawal, 2021) o el *makespan* con restricciones *no-wait* (Lin & Ying, 2016) y *no-idle* (Ying et al., 2017). Además, de la gran variedad de problemas multiobjetivos que buscan la eficiencia energética o sostenibilidad del entorno.

Los problemas con objetivos sostenibles desean minimizar costes, ya sean salariales (Alaghebandha et al., 2019) o de inventario (Fathollahi-Fard et al., 2021), entre otros. Dentro de los objetivos sostenibles, se encuentran

aquellos orientados a temas medioambientales, como es la minimización de residuos (Fathollahi-Fard et al., 2021) o los que minimizan el impacto social negativo de la actividad (Lu et al., 2021).

Actualmente una de las líneas de investigación del DPFS continúa con objetivos de eficiencia energética (TEC). Estos objetivos quieren dar solución al gran consumo energético que supone la industria: para minimizar el *makespan* del taller, hay que aumentar la velocidad de las máquinas y en consecuencia aumenta el consumo energético. El problema para equilibrar el coste del consumo-tiempo, disminuye la velocidad de aquellos trabajos que no determinan el *makespan*, minimizando el TEC.

El primer problema de la línea de investigación en cuanto a consideraciones energéticas es el conocido como *Energy-Efficient Distributed No-idle Flow shop Scheduling Problem* (Chen et al., 2019). Este escenario no permite tiempos ociosos entre máquinas a través de la restricción *no-idle* y mediante el algoritmo colaborativo de optimización (COA), un método aproximado, se minimiza el *makespan* obteniendo como resultado la secuencia de cada fábrica y la velocidad de las máquinas en las mismas. A partir de este, otros problemas similares han sido planteados en la literatura, como (J. J. Wang & Wang, 2020) que omite la restricción *no-idle* o (G. Wang et al., 2020) que considera los tiempos de preparación de las máquinas.

4 HEURÍSTICAS CONSTRUCTIVAS PARA LA RESOLUCIÓN DEL PROBLEMA

Las heurísticas constructivas utilizadas para la resolución del $DF_m \mid prmu \mid \sum CIT$ son algoritmos planteados en la literatura para el mismo entorno pero con distintos objetivos, ya sea *makespan* o *Total Flow Time* y que han generado muy buenos resultados. Esta investigación comenzó en el TFG (Ariza Gamero, 2021) donde se propusieron las heurísticas constructivas del capítulo 1.3, Objetivo del proyecto.

En primer lugar, se presentaron las heurísticas DLR, como adaptación de la heurística LR (Liu & Reeves, 2001), y DNEH, como adaptación de la NEH (Nawaz & Enscore, 1983), de (Pan et al., 2019) ya que ambas emplean un índice de ordenación de trabajos que depende de tiempo ocioso de la máquina. La DLR construye el programa añadiendo los trabajos de uno en uno al final de cada fábrica, y la DNEH incluye los trabajos en las secuencias buscando el menor incremento en la función objetivo al probar en cada posición de cada fábrica.

También, partiendo de la NEH, se adaptaron la NEH(R_1, A_i) y NEH(R_2, A_i) propuesta por (Fernandez-Viagas et al., 2016) considerando los dos tipos de representaciones y las seis reglas de asignación, siendo ambas necesarias para poder generar el programa. L a denominada representación R_1 da una única secuencia de trabajos que se asignará en función de la regla de asignación mientras que la representación R_2 da tantas secuencias como fábricas sin que sea necesaria la regla de asignación, ya que forma parte de la codificación.

Las seis reglas de asignación adaptadas, denominadas como A_i, son las siguientes:

- A_1*:* Asigna el trabajo a insertar al final de la fábrica que tiene menor tiempo ocioso.
- A_2*:* Asigna el trabajo a insertar al final de la fábrica que tiene menor tiempo ocioso.
- A_3*:* Asigna el trabajo en la mejor posición de la fábrica que genera menor tiempo ocioso en la fábrica tras su inserción.
- A_4*:* Asigna el trabajo en la mejor posición de la fábrica que genera menor valor de la función objetivo tras su inserción.
- A_5: Asigna el trabajo en la mejor posición de la fábrica que genera menor valor de la función objetivo tras su inserción sin evaluar la fábrica que tiene mayor tiempo ocioso.
- A_6: Asigna el trabajo en la mejor posición de la fábrica que genera menor valor de la función objetivo tras su inserción. Esta regla evalúa el subconjunto *F'* formado por las (*f*/2) fábricas con menores tiempo de flujo total.

Debido a los buenos resultados de esta última heurística y considerando el índice de ordenación de la DLR y la NEH, se presentó una última heurística denominada DNEH(R_1, A_i) y DNEH(R_2, A_i). El proyecto concluye que las mejore heurísticas planteadas son NEH(R_1, A_4) y DNEH(R_1, A_4) dependiendo de los valores de *f*, *n* o *m.*

4.1 Adaptación de la heurística DLR-DNEH

La heurística constructiva DLR-DNEH fue propuesta por (Pan et al., 2019) para el problema $DF_m \mid prmu \mid \sum F_j$. Esta heurística híbrida surge a partir de las heurísticas DLR, inspirada por la clásica LR (Liu & Reeves, 2001), y DNEH, inspirada por el clásico NEH (Nawaz & Enscore, 1983) descritas en este mismo artículo: se genera una solución parcial de *x* trabajos según la DLR y se completa la secuencia con la DNEH. El parámetro *x* determina el porcentaje de trabajos que son secuenciados uno a uno por la DLR. El procedimiento se muestra en *Figura 6*.

Ambas heurísticas, tanto DLR como DNEH ordenan los trabajos de forma decreciente en función del índice

$IF_{j,0}$ (*Ecuación 3*), cuyo interés será explicado a continuación, y asigna los *f* primeros trabajos a las *f* fábricas que componen el entorno siendo este criterio también utilizado para la heurística híbrida. Sin embargo, la heurística DLR selecciona la fábrica con menor CIT y asigna a la última posición de esta fábrica, n_{k*}, el trabajo que genere menor valor del índice $IF_{j,\mathrm{n}_{k*}}$ mientras que la heurística DNEH toma los trabajos con menores $IF_{j,0}$ (*Ecuación 4*) y los asigna a aquella fábrica y a aquella posición que genere menor impacto en el objetivo.

$$IF_{j,n_{k*}} = ({}^{n}/_{f} - n_{k*} - 2)IT_{j,n_{k*}} + C_{m,j}$$

Ecuación 1. Índice IF (j, n_{k*}*)*

$$IT_{j,n_{k*}} = \sum_{i=2}^{m} \frac{m \cdot max\{C_{i-1,j} - C_{i,[n_{k*}],}0\}}{i + n_{k*} \cdot (m-1)/({}^{n}/_{f} - 2)}$$

Ecuación 2. Cálculo Idle Time para posición n_{k*}

El interés de esta heurística para el $\sum CIT$ se encuentra en el índice que ordenación $IF_{j,\mathrm{n}_{k*}}$. Este índice tiene en cuenta, por un lado, el *makespan* de la máquina al asignar el trabajo y, por el otro, el tiempo ocioso calculado gracias al $IT_{j,n_{k*}}$(*Ecuación 2*). No obstante, cuando un único trabajo es asignado en una fábrica no se generan tiempos ociosos y el índice $IT_{j,0}$ (*Ecuación 4*) se simplifica notablemente.

Procedure DLR-DNEH (x)
Compute $IF_{j,0}$ for each *j* ∈ *N*
Generate job permutation λ = (λ_1, λ_2, …, λ_n) according to non-decreasing order of $IF_{j,0}$
π := (π_1, π_2, …, π_f), where π_k= (λ_k) , *k* =1,2,…,*f*
Remove jobs λ_1, λ_2, …, λ_k from λ
while sizeof (λ) > *x* **do** *// DLR loop*
    Find factory *k** that has the lowest core idle time
    Compute $IF_{j,\mathrm{n}_{k*}}$for each job *j* included in λ
    *j** := job leading to smallest $IF_{j,\mathrm{n}_{k*}}$
    Remove job *j* from λ
endwhile
while sizeof (λ) > 0 **do** *// DNEH loop*
    Extract the first job *j* from λ
    for *k* := 1 **to** *f* **do**
        Test job *j* at all the possible positions of π_k
        Δ_k = minimum increase total core idle time
        ξ_k = position resulting in Δ_k
    endfor
*k** := arg($\min_{k:=1,2,\ldots,f}(\Delta_k)$)
Insert job *j* at position ξ_k of Δ_k
endwhile
return π

Figura 6. Pseudocódigo adaptación DLR-DNEH

$$IF_{j,0} = ({}^{n}/_{f} - 2)IT_{j,0} + C_{m,j}$$

Ecuación 3. Cálculo IF (j,0)

$$IT_{j,0} = \sum_{i=2}^{m} \frac{m \cdot C_{i-1,j}}{i}$$

Ecuación 4. Cálculo Idle Time para posición 0

4.2 Adaptación de la heurística NEH2

El clásico NEH para el problema $F_m \mid prmu \mid C_{max}$ de (Nawaz & Enscore, 1983) ha sido adaptado en numerosas ocasiones, como en el capítulo 4.1. En el artículo (Naderi & Ruiz, 2010), al igual que en el clásico, se calcula por trabajo el tiempo total que tarda en ser procesado al pasar por todas las máquinas y estos trabajos se van asignando uno a uno según el menor tiempo calculado. No obstante, los trabajos no se incluyen al final de la secuencia de la fábrica, si no que se prueba en cada una de las posibles posiciones de cada fábrica y se secuencia en aquella posición que ha generado menor impacto en el CIT. En la *Figura 7* se muestra el pseudocódigo del método.

Procedure NEH2

Calculate $P_j = \sum_{i=1}^{m} p_{ij}$, $\forall n \in N$

π^{LPT} := Sort the *n* jobs according to P_j in decreasing order

for *f* := 1 **to** *F* **do**

$\pi := (\pi_1, \pi_2, \ldots, \pi_F)$

for *step* := 1 **to** *n* **do**

j := $\pi^{LPT}[step]$

for *f* = 1 **to** *F* **do**

Test job *j* in all possible positions of π_f

CIT^f is the lowest *CIT* obtained

p^f is the position where the lowest *CIT* is obtained

endfor

endfor

end

Figura 7. Pseudocódigo adaptación NEH2

4.3 Improvement EN para NEH2

A partir del NEH2 explicado en el capítulo 4.2, (Ruiz et al., 2019) incluye una extensión o *improvement* a la heurística constructiva. Después de asignar el trabajo $j := \pi^{LPT}[step]$, de manera aleatoria, se extrae el trabajo anterior o posterior secuenciado, que se encuentra en la posición *h* y se inserta en cada posible posición de esta misma fábrica para obtener el menor *CIT* posible (*Figura 8.Pseudocódigo adaptación NEH2_en*).

Procedure NEH2_en

Calculate $P_j = \sum_{i=1}^{m} p_{ij}$, $\forall n \in N$

π^{LPT} := Sort the *n* jobs according to P_j in decreasing order

for *f* := 1 **to** *F* **do**

$\pi = (\pi_1, \pi_2, \ldots, \pi_F)$

for *step* := 1 **to** *n* **do**

$j := \pi^{LPT}[step]$

for *f* := 1 **to** *F* **do**

Test job *j* in all possible positions of π_f

CIT^f is the lowest *CIT* obtained

p^f is the position where the lowest *CIT* is obtained

endfor

f_{min} = arg ($\min_{F=1,2,\ldots,f}(CIT^f)$)

Insert job *j* in $\pi_{f_{min}}$ at the position $p^{f_{min}}$ resulting in the lowest *CIT*

Extract at random job *h* from the position $p^{f_{min}}$-1 or $p^{f_{min}}$+1 from $\pi_{f_{min}}$

Test job *h* in all possible positions of $\pi_{f_{min}}$

Insert job *h* in $\pi_{f_{min}}$ at the position resulting in the lowest *CIT*

endfor

end

Figura 8.Pseudocódigo adaptación NEH2_en

4.4 Improvement EN para NEH (R_1, A_4)

Como se ha indicado en la introducción del capítulo, en el TFG al comparar el índice $RDI^h{}_I$ (*Ecuación 5*), se demostró que la NEH (R_1, A_4) genera los mejores resultados de los planteados en (Fernandez-Viagas et al., 2018) para el $DF_m \mid prmu \mid \sum F_j$. En este estudio se presenta esta heurística constructiva y mejorándola gracias al *improvement en*. En *Figura 9* se muestra el pseudocódigo de la heurística constructiva adaptada.

Procedure NEH (R_1*,* A_4*)_en*

Ω := $(\Omega_1, \Omega_2, \ldots, \Omega_n)$ jobs sorted according to non-increasing order to sums of processing times (Ω := $\{\omega_1, \omega_2, \ldots, \omega_n\}$)

Π := (ω_1)

for *k*:=2 **to** *n* **do**

$\Pi^{\omega_k{}^l} :=$ Partial sequence formed by inserting job ω_k in position *l* of partial sequence Π where *l*=1, 2, …, *k*

Compute each partial sequence $\Pi^{\omega_k{}^l}$ by using assigment rule A_4 to allocate jobs to factories at the end of that factory. Let *l**be the index *l* whose total core idle time is minimal and A_4 the rule that assign the jobs to the factory yielding the lowest total CIT after inserting the job in the best position

Π :=$\Pi^{\omega_k{}^l}$

Extract at random job *h* from the position *l* -1 or *l* +1 from Π

Test job *h* in all possible positions of Π and compute each sequence by using assigment rule A_4

Insert job *h* in Π at the position resulting in the lowest total *CIT*

endfor

return Π

Figura 9.Pseudocódigo adaptación NEH (R_1*,* A_4*)_en*

5 METAHEURÍSTICAS PARA LA RESOLUCIÓN DEL PROBLEMA

De manera similar al capítulo anterior, Heurísticas constructivas para la resolución del problema, se han tomado las mejores metaheurísticas de la literatura hasta el momento para el entorno en estudio con objetivos clásicos y han sido adaptadas para el objetivo $\sum CIT$.

5.1 Adaptación de la metaheurística ILS

El artículo de (Pan et al., 2019) presenta para el problema $DF_m \mid prmu \mid \sum F_j$ las metaheurísticas basadas en población *Scatter Search* (SS) y D*iscrete Artificial Bee Colony* (DABC) y vecindad, *Iterated Local Search* (ILS) e I*terated Greedy* (IG) que han generado buenos resultados para problemas similares. Este estudio concluye demostrando que el ILS genera los mejores resultados tras comparar las cuatro metaheurísticas planteadas junto a las mejores hasta el momento.

En términos generales, un ILS parte de una solución inicial dada por una heurística constructiva, a la que se genera una búsqueda local para obtener una vecindad y aplicar alguna perturbación al mejor de los vecinos generados para salir de óptimos locales.

La solución inicial se obtiene de la heurística constructiva DLR-DNEH(0.2), explicada en Adaptación de la heurística DLR-DNEH y la búsqueda local empleada se denomina R*eferences Local Search* (RLS), explicada en la *Figura 10*. En el RLS, se extraen los trabajos programados de manera consecutiva y se reinsertan en cada posición de cada fábrica para obtener obteniendo una vecindad. Si de una de las soluciones de la vecindad obtiene un valor menor para el $\sum CIT$ que la inicial propuesta, entonces esta se consideraría este el mejor vecino y continuaría la búsqueda a partir de esta. Este proceso es iterativo hasta que todos los trabajos de una misma solución se han reinsertado sin conseguir mejoras en el objetivo.

El proceso de perturbación ayuda a escapar de óptimos locales y amplía la zona de búsqueda a partir de las soluciones generadas por la perturbación. Para mejorar el proceso de perturbación, el ILS genera ϖ perturbaciones a la mejor solución hasta el momento para continuar con la mejor de ellas y obtener una vecindad gracias al RLS. Este algoritmo escapa de óptimos locales al utilizar el operador de inserción: aleatoriamente extrae un trabajo de una fábrica para ser insertado de nuevo de manera aleatoria en otra posición de otra fábrica.

Si la solución generada tras la perturbación y la búsqueda local de esta, se acepta esta solución como la mejor hasta el momento obteniendo entonces un nuevo campo de búsqueda. Sin embargo, si no mejora se puede aceptar la solución según el criterio *Simulated Annealing* (SA) bastante extendido en la literatura. La solución es aceptada si mejora probabilísticamente en función del parámetro conocido como temperatura *t.* La temperatura depende del tiempo de proceso de los trabajos en las máquinas, el número de máquinas y trabajos por fábrica y el parámetro β que debe ser ajustado.

El ILS queda definido cuando se ajustan los parámetros que definen la heurística.

- α es la heurística que genera una solución inicial.
- ϖ es el número de perturbaciones a realizar sobre una solución para obtener una vecindad.
- τ es el número de soluciones generadas con el operador de perturbación de una solución.
- Ξ es el proceso de perturbación.
- β es el factor de temperatura.

La *Figura 11* muestra el pseudocódigo de la adaptación del ILS y los valores de los parámetros que la definen.

Procedure RLS (π, λ**)**

Counter := 0

j := 1

while *Counter* < *n* **do**

Find job λ_j from π

$\pi^{-\lambda_j}$:= partial solution generated by extracting job λ_j from π

π' := best solution generated after reinserting job λ_j to $\pi^{-\lambda_j}$

if *CIT*(π') < *CIT*(π) **then**

π := π'

Counter := 0

else

Counter := *Counter*+1

endif

j := (*j*+1)/*n* +1

endwhile

return π

Figura 10. Pseudocódigo adaptación RLS

Procedure ILS (*α, ϖ, τ, Ξ, β*)

π := constructive heuristic α (α := *DLR-DNEH(*0.2*)*)

Perform *RLS* to π

π := π* // best solution so far

$t := \beta * \frac{\sum_{j=1}^{n}\sum_{k=1}^{m} p_{k,j}}{10*n*m}$

while (*CPU* < 0.01**n***m*) **do**

for *i*=1 **to** ϖ **do**

$\pi'(i)$:= performing τ times operator Ξ to π

endfor

π'' := best solution among {$\pi'(1)$,…, $\pi'(\varpi)$}

Perform *RLS* to π"

if *CIT*(π") < *CIT*(π) **then**

π := π"

elseif *rand*( ) < *exp*((*CIT*(π)-*CIT*(π")) /t) **then**

π* := π"

endif

endwhile

return π*

Figura 11. Pseudocódigo adaptación ILS

5.2 Adaptación de la metaheurística IG2S

El *Iterated Greedy* presentado por (Ruiz et al., 2019) sigue el esquema de la mayoría de los IG propuestos en la literatura: inicialización, búsqueda local, destrucción, reconstrucción y criterio de aceptación. Sin embargo, el interés de esta metaheurística cae en una última nueva fase de búsqueda de soluciones que se realiza en la fábrica que genera mayor CIT. La *Figura 12* indica el esquema que sigue la metaheurística planteada.

En primer lugar, se genera una solución inicial mediante la heurística constructiva *NEH2_en*, anteriormente explicada en Improvement EN para NEH2, para obtener una vecindad mediante la búsqueda local *LS3.* Al mejor vecino generado tras la búsqueda es considerado como la mejor solución hasta el momento para posteriormente, realizar una búsqueda iterativa en la que la solución es parcialmente destruida en el proceso de *Destruction* y regenerada de nuevo en *Reconstruction.* Una vez más gracias al *LS3,* se obtiene el mejor vecino de la solución obtenida tras *Destruction* y *Reconstruction*. Si el $\sum CIT$ ha mejorado o probabilísticamente se acepta por el SA, entonces se acepta la nueva solución.

En la mayoría de los *IG*, la búsqueda comenzaría tras el criterio de aceptación generando de nuevo una solución a partir de *Destruction-Reconstruction-LS3.* No obstante, en estos casos el método emplea gran parte de su tiempo en la búsqueda de soluciones que no mejoran la función objetivo. El IG2S destina una parte del tiempo computacional a mejorar la fábrica que genera mayor *CIT* y que, por tanto, no minimiza el objetivo. A continuación, se detalla cada una de las fases del proceso iterativo y de la segunda búsqueda de soluciones generada a partir de la fábrica con mayor CIT.

Procedure I2GS

π_0 := constructive heuristic *NEH2_en* //Initial Solution

π := *LS3* (π_0) //LocalSearch

t := $T * \frac{\sum_{j=1}^{n}\sum_{k=1}^{m} p_{k,j}}{10*n*m}$

while (*CPU* < *ρ**0.01**n***m*) **do**

π_D, π_R := *Destruction* (π)

π' := *Reconstruction* (π_D, π_R)

π" := *LS3* (π')

if *CIT*(π") < *CIT*(π) **then** //Acceptance Criteria

π := π"

elseif rand( ) < *exp*((*CIT*(π)-*CIT*(π")) /t) **then**

π* := π"

endif

endwhile

while (*CPU* < (1-*ρ*)*0.01**n***m*) **do**

endwhile

f_{max} := arg($\max_{k:=1,2,\ldots,f}(CIT^f)$)

π_{2S} := *TwoStage* (π, f_{max})

if (*CIT*(π_{2S}) < *CIT*(π)) //Acceptance Criteria

π := π_{2S}

endif

enwhile

return π

Figura 12. Pseudocódigo adaptación IG2S

El proceso de búsqueda local LS3 presentado en la *Figura 13* toma trabajos aleatorios de la fábrica como mayor *CIT,* sin repetición, para ser reinsertado en cada una de las posiciones de las fábricas del entorno. Si el *CIT* total mejora en algunos de las soluciones generadas por la vecindad, entonces este trabajo se considera el mejor hasta el momento y una nueva búsqueda comienza. El proceso solo finaliza cuando todos los trabajos de la fábrica con mayor CIT han sido reinsertados sin obtener soluciones mejores hasta el momento.

En la fase de *Destruction* se generan dos soluciones parciales: una a partir de los trabajos que van a ser reinsertados y otra de los trabajos que continúan en la fábrica y no van a ser reinsertados. El pseudocódigo se presenta en *Figura 14*. Se extraen de la fábrica con mayor *CIT* (*d*/2) trabajos y (*d*/2) del resto de fábricas. Todos los trabajos se extraen de manera aleatoria y se introducen secuencialmente en la solución parcial π_R. La solución parcial de trabajos que continúan secuenciado se denomina π_D.

Procedures LS3

$\pi := \{ \pi_1, \pi_2, \ldots, \pi_F \}$

$sumCIT^* = \sum_{k=1}^{f} CIT\ (\pi_k)$

$CIT_{max} = \max_{k:=1,2,\ldots,f} (CIT(\pi_f))$

f_{max} = arg(CIT_{max})

Count := 0

while (*Count* < $|\pi_{f_{max}}|$) **do**

Randomly extract, without repetition, a job *j* from position *k* of $\pi_{f_{max}}$

Test job *j* in all possible positions of π_f, except position *k* of $\pi_{f_{max}}$

$sumCIT^{f_{min}}$ is the lowest *sumCIT* obtained by inserting job *j* in the position p^f of factory f_{min}

if ($sumCIT^{f_{min}}$ < *sumCIT**) **then**

Place job *j* at position p^f of factory f_{min}

$CIT_{max} = \max_{k:=1,2,\ldots,f} (CIT(\pi_f))$

f_{max} = arg(CIT_{max})

Count := 0

elseif

Return job *j* to position *k* of f_{max}

Count := *Count* + 1

endif

endwhile

return π

Figura 13. Pseudocódigo adaptación LS3

En la segunda etapa, de manera similar al *IG* explicado, la secuencia de la fábrica con mayor CIT es parcialmente destruida y reconstruida para aplicar una búsqueda local con *improvement en*. Sin embargo, en esta fase para aceptar una secuencia el *CIT* debe haber disminuido. El pseudocódigo de la etapa se define en *Figura 16*.

En la fase de *Reconstruction* todos los trabajos de π_R son insertados en la solución parcial π_D. Para ello, se toman de manera secuencial, y hasta que no queden trabajos, los trabajos de π_R y se introducen en cada posición de cada fábrica de π_D consiguiendo el menor incremento en el objetivo. Esta fase tiene en cuenta el *improvement en* utiizado en Improvement EN para NEH2: de manera aleatoria toma el trabajo anterior o posterior al secunciado y lo reprograma en la fábrica para obtener el menor *CIT* de esta misma fábrica, tal y como se indica en la *Figura 15*.

Los parámetros que definen el problema son los siguientes:

- d son los trabajos extraídos para obtener π_R.
- d_2 son los trabajos extraídos para obtener π_R en *TwoStage.*

- ρ es el porcentaje en tanto por uno del tiempo computacional en la segunda etapa.
- T es el factor de temperatura.

Procedure Destruction

$\pi := \{ \pi_1, \pi_2, \ldots, \pi_F \}$

$f_{max} := \arg(\max_{k:=1,2,\ldots,f}(CIT(\pi_f))$

for j=0 **to** (d/2) **do**

Randomly extract a job *j* from $\pi_{f_{max}}$

Insert job *j* to π_R

endfor

for j=0 **to** (d/2) **do**

Randomly extract a job *j* from π_f $j \in F - \{f_{max}\}$

Insert job *j* to π_R

endfor

π_D := partial sequence with jobs no removed from π

return π_D, π_R

Figura 14. Pseudocódigo adaptación Destruction

Procedure Reconstruction

while ($|\pi_R| > 0$)

for *f* := 1 **to** F **do**

Test job *j* in all possible positions of π_f

CIT^f is the lowest *CIT* obtained

p^f is the position where the lowest *CIT* is obtained

endfor

$f_{min} := \arg(\min_{k:=1,2,\ldots,f} CIT(\pi_f))$

Extract at random job *h* from the position $p^{f_{min}}$-1 or $p^{f_{min}}$+1 from $\pi_{f_{min}}$

Test job *h* in all possible positions of $\pi_{f_{min}}$

Insert job *h* in $\pi_{f_{min}}$ at the position resulting in the lowest *CIT*

endwhile

return π

Figura 15. Pseudocódigo adaptación Reconstruction

Procedure TwoStage

$\pi := \{ \pi_1, \pi_2, \ldots, \pi_F \}$

$f_{max} :=$ arg($\max_{k:=1,2,\ldots,f}(CIT(\pi_f))$

for j=0 **to** (d_2) **do** //Destruction

Randomly extract a job *j* from $\pi_{f_{max}}$

Insert job *j* to π_R

endfor

π_D := partial sequence with jobs no removed from $\pi_{f_{max}}$

while ($|\pi_R| > 0$) //Reconstruction

Randomly extract a job *j* from π_R

Test job *j* in all possible positions of π_D

p^D is the position of π_D where the lowest *CIT* is obtained

Insert job *j* at the position p^D resulting in the lowest *CIT*

Extract at random job *h* from the position $p^{f_{min}}$-1 or $p^{f_{min}}$+1 from π_D

Test job *h* in all possible positions of π_D

Insert job *h* in π_D at the position resulting in the lowest *CIT*

endwhile

$\pi_{f_{max}} := \pi_D$

$CIT^{f_{max}} :=$ CIT($\pi_{f_{max}}$

while (*Count* < $|\pi_{f_{max}}|$) **do** //LocalSearch

Randomly extract, without repetition, a job *j* from position *k* of $\pi_{f_{max}}$

Test job *j* in all possible positions of $\pi_{f_{max}}$

p^f is the position where the lowest *CIT* is obtained

CIT^f is the lowest *CIT* obtained

if ($CIT^f < CIT^f$) **then** //AcceptanceCriteria

Insert job *j* in position p^f resulting in the lowest *CIT*

endif

endwhile

Figura 16.Pseudocódigo adaptación TwoStage

6 RESULTADOS OBTENIDOS

Las cuatro heurísticas constructiva descritas en el capítulo 4.Heurísticas constructivas para la resolución del problema, y las cuatro metaheurísticas del capítulo 5. Metaheurísticas para la resolución del problema han sido evaluadas usando 72 instancias de (Naderi & Ruiz, 2010). Este conjunto de instancias presenta valores de f ϵ {2, 3, 4, 5, 6, 7}, m ϵ {2, 3, 4, 5, 10, 20} y n ϵ {4, 6, 8, 10, 12, 14, 16, 20, 50, 100}. Las instancias utilizadas, clasificadas en *Small* y *Large*, en función del número de trabajos, se muestran en las *Tabla 5. Instancias SmallTabla 5* y *Tabla 6*, respectivamente. Para ello, se han codificados los algoritmos en lenguaje de programación C mediante el programa CodeBlocks y la librería *schedule* (*Librería SCHEDULE – Grupo de Investigación Organización Industrial*). El código utilizado en la programación se encuentra en el Anexo Código programación en C.

Small		
f	**n**	**m**
2	4	2
2	4	3
2	4	4
2	4	5
2	6	4
2	6	5
2	8	2
2	8	5
2	10	5
2	12	5
2	14	5
2	16	2
3	4	4
3	4	5
3	6	2
3	6	3
3	8	5
3	10	2
3	10	3
3	12	3
3	14	4
3	16	2
3	16	3
4	4	2
4	4	3
4	6	2
4	6	3
4	8	2
4	10	2
4	12	4
4	14	2
4	14	4
4	14	5
4	16	4

Tabla 5. Instancias Small

Large		
f	**n**	**m**
2	20	5
2	20	10
2	50	10
2	50	20
2	100	5
2	100	10
3	20	5
3	20	10
3	20	20
3	50	5
3	50	10
3	100	10
3	100	20
4	20	10
4	50	20
4	100	5
5	20	10
5	20	20
5	50	5
5	50	20
5	100	5
5	100	10
5	100	20
6	20	5
6	20	10
6	20	20
6	50	5
6	50	10
6	50	20
6	100	5
6	100	10
6	100	20
7	20	5
7	20	10
7	20	20
7	50	10
7	50	20
7	100	5

Tabla 6. Instancias Large

La comparación de las heurísticas ha sido realizada por el índice $RDI^h{}_I$ (Ecuación 5), donde I es la instancia a evaluar y h la heurística evaluada. El índice muestra la desviación de una instancia evaluada mediante una heurística respecto al mejor valor obtenido de esa instancia para todas las heurísticas evaluadas cuando se busca minimizar el objetivo. Si se obtienen valores próximos a 1, el valor obtenido se aproxima al peor de los evaluados siendo el 1 la peor solución. Por el contrario, los valores próximos a 0 significan valores cercanos al mejor resultado obtenido, siendo 0 el mejor valor obtenido. Los resultados obtenidos de cada instancia por cada heurística están a disposición del tribunal bajo solicitud, así como el índice $RDI^h{}_I$.

$$RDI^{h}{}_{I} = \frac{FO^{h}{}_{I} - \min_{H} FO_I}{\max_{H} FO_I - \min_{H} FO_I} \qquad H := DLR - DNEH, NEH2, NEH2\_en, NEH(R_1, A_4)\_en$$

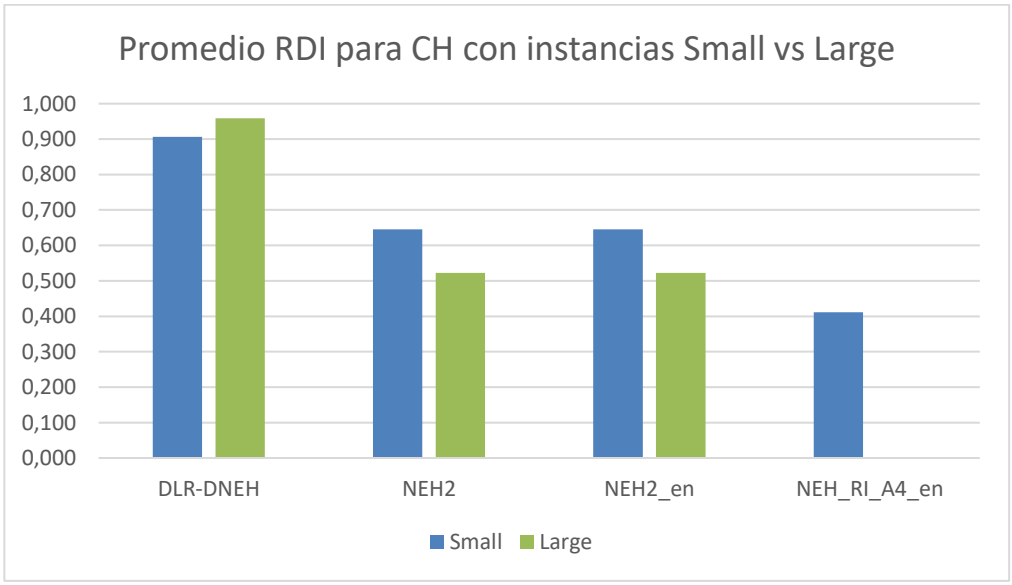
Ecuación 5. Índice RDI

En primer lugar, se estima para el problema en estudio la mejor heurística constructiva (CH) evaluada. El promedio del índice para cada *h* se muestra en la Tabla 7. Tras el análisis se concluye que los mejores resultados se obtienen con la *NEH (R_1, A_4)* al introducir el improvement *en.* Esta CH presenta los mejores resultados para 58 de las instancias estudiadas, a diferencia de la *DLR-DNEH* que no obtiene el mejor valor para ninguna de las instancias evaluadas.

Promedio RDI	
DLR-DNEH	0,934
NEH2	0,581
NEH2_en	0,580
NEH_RI_A4_en	0,194

Tabla 7. Promedio RDI por CH

En la *Gráfica 1* se han representado los $RDI^{h}{}_{I}$ diferenciando entre las instancias consideradas como S*mall* y *Large*. Se observa que la CH a utilizar no depende del número de trabajos a evaluar, la DLR-DNEH dará los peores resultados para cualquiera de las casuísticas, seguida de la NEH2 y la NEH2 con improvement *en*. No obstante, al comparar el promedio $RDI^{h}{}_{I}$ obtenido para las instancias *Small* y las instancias *Large* de la *NEH (R_1, A_4)* con improvement *en*, se puede ver que para aquellas instancias que contengan entre 20 y 100 trabajos, esta heurística dará para el mejor resultado de los estudiados según el índice ya que el promedio es 0.



Gráfica 1.Promedio RDI para CH

Por otro lado, se han evaluados las metaheurísticas ILS e IG2S considerando las soluciones iniciales indicas en los capítulos 5.1 y 5.2, respectivamente y, tras conocer la mejor de las heurísticas constructivas, se ha considerado la mejor de ellas como solución inicial para ambas metaheurísticas con el objetivo de conocer el impacto de la solución inicial en el problema en estudio. Se denomina como metaheurísticas originales a las detalladas en 5.1 y en 5.2 y las metaheurísticas con la mejor CH como metaheurísticas mejoradas.

Los parámetros que definen ambas heurísticas se encuentran en la *Tabla 8* para ILS y en la *Tabla 9* para IG2S.

Parámetros ILS	
ϖ	20
τ	3
Ξ	*insertion*
β	0.7

Tabla 8. Valores de los parámetros para ILS

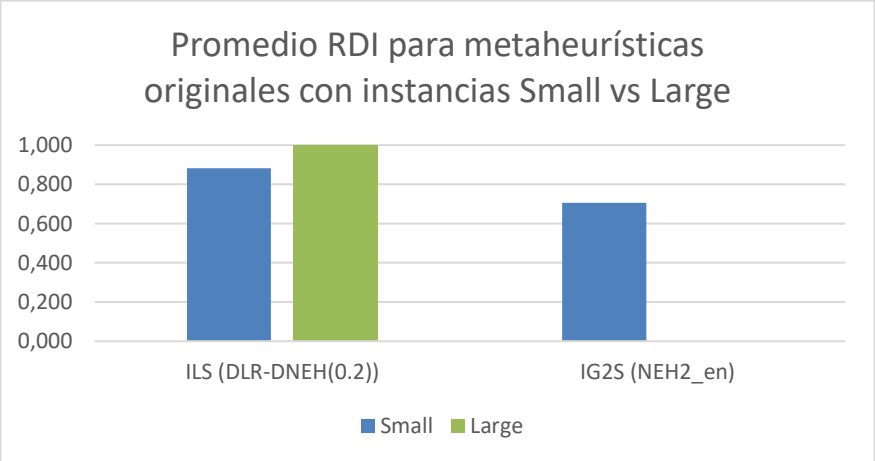
Parámetros IG2S	
d	5
d_2	6
ρ	0.95
T	0.2

Tabla 9. Valores de los parámetros para IG2S

Se ha establecido un tiempo de ejecución (CPU) de 0.01 x *n* x *m* segundos para ejecutar las 72 instancias después de obtener la solución inicial, donde el tiempo no ha sido considerado. Al comparar los resultados obtenidos en las heurísticas iniciales detalladas en los capítulos 5.1 y 5.2, se concluye que la IG2S obtiene los mejores resultados (*Tabla 10).* Además, los mejores resultados se van a obtener siempre con la IG2S cuando se consideran 20 trabajos o más tal y como se observa en la *Gráfica 2*, para las instancias *Large* el índice es 0. Sin embargo, la gran diferencia entre ambas heurísticas para el problema en estudio se encuentra precisamente en la solución inicial.

Promedio RDI	
ILS (DLR-DNEH(0.2))	0,944
IG2S (NEH2_en)	0,333

Tabla 10. Promedio RDI para ILS (DLR-DNEH(0.2)) e IG2S (NEH2_en)



Gráfica 2. Promedio RDI metaheurísticas originales para instancias Small vs Large

Al generar la solución inicial mediante la CH *NEH* (R_1, A_4) con improvement *en* obtenemos los valores de la

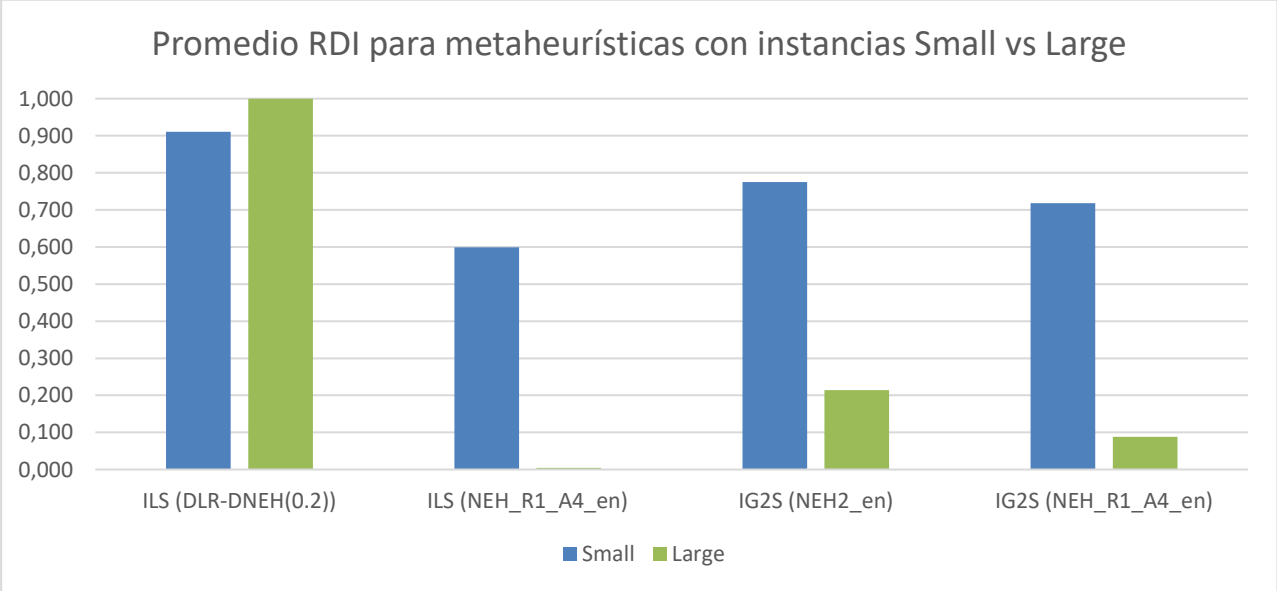
Tabla 11, que evalúa el $RDI^{h}{}_{I}$ considerando las cuatro metaheurísticas: las dos originales y las dos mejoradas . Dado que la peor de las heurísticas constructivas era la *DLR-DNEH(0.2),* el peor valor se obtiene con la ILS original, sin embargo, esta es la mejor de las metaheurísticas si se considera la *NEH* (R_1, A_4) con improvement *en.* De manera similar ocurre con la I2GS: el índice mejora considerablemente. Al desglosarlo en instancias *small* y *large,* se aprecia en la *Tabla 12* y en *Gráfica 3* de manera más visual que ambas heurísticas obtienen mejores resultados, el $RDI^{h}{}_{I}$ es 0, si se incluye la mejor CH como solución inicial, estando en primer lugar la ILS, seguida de la IG2S para ambos.

Promedio RDI	
ILS (DLR-DNEH(0.2))	0,958
ILS (NEH_R1_A4_en)	0,285
IG2S (NEH2_en)	0,479
IG2S (NEH_R1_A4_en)	0,385

Tabla 11. Promedio RDI para metaheurísticas originales

Promedio RDI	Instancias Small	Instancias Large
ILS (DLR-DNEH(0.2))	0,911	1,000
ILS (NEH_R1_A4_en)	0,599	0,004
IG2S (NEH2_en)	0,775	0,214
IG2S (NEH_R1_A4_en)	0,718	0,088

Tabla 12. Promedio RDI para metaheurísticas originales y metaheurísticas mejoradas con instancias small vs large



Gráfica 3. Promedio RDI para metaheurísticas con instancias Small vs Large

7 CONCLUSIONES

El entorno *Distributed Flow shop* con permutación y objetivo *Total Core Idle Time* empezó a ser estudiado en el TFG (Ariza Gamero, 2021), proponiéndose un total de 26 heurísticas constructivas. Las heurísticas fueron programas en lenguaje de programación C y se evaluaron, por cada heurística, 72 instancias extraídas de (Naderi & Ruiz, 2010). Es decir, se obtuvieron 1872 valores para ser comparados entre ellos con el fin de obtener el mejor método de resolución. La *NEH* (R_1, A_4) adaptada y la nueva *DNEH* (R_1, A_4) concluyeron ser las mejores heurísticas constructivas de las propuestas.

Este proyecto continúa esta línea de investigación considerando métodos aproximados avanzados como son las metaheurísticas. A través de una revisión del estado-del-arte, se ha considerado que los mejores métodos para ser adaptados al problema en estudio son el *Iterated Local Search, ILS,* de (Pan et al., 2019) para el objetivo Total Flow Time y el *Iterated Greedy with Two Stage IG2S* de (Ruiz et al., 2019) para el objetivo *makespan*.

También en esta ocasión se han programado las metaheurísticas en lenguaje C y se ha estudiado la influencia de la solución inicial en estas. Además de las heurísticas DLR-DNEH(02) para el *ILS* y NEH2 con *improvement en* para el *IG2S*, se han programado las heurísticas *NEH2* y *NEH* (R_1, A_4) con *improvement en* para conocer la que presenta menor valor del índice $RDI^h{}_I$ y ser considerada la solución inicial para las metaheurísticas.

Por un lado, al comparar las heurísticas constructivas se ve que la *NEH* (R_1, A_4) con *improvement* claramente obtiene los mejores resultados para las distintas instancias evaluadas, obteniendo siempre la mejor cuando se trabaja con 20, 50 o 100 trabajos hasta el momento. La *DLR-DNEH(0,2)* solo para un número muy pequeño de instancias obtiene los mejores valores, siendo esta la peor de todas ellas.

Por otro lado, al comparar las metaheurísticas, la *IG2S* resulta mejores valores para la función objetivo que la *ILS*. Sin embargo, si ambas parten de la misma solución inicial tras emplear el método constructivo *NEH* (R_1, A_4) con *improvement,en*, la ILS es bastante mejor que la *IG2S*. Esto demuestra que la inicialización de las metaheurísticas es importante hasta el punto de obtener los peores resultados si se parte de una mala solución inicial, como es la heurística *DLR-DNEH(0.2)*.

BIBLIOGRAFÍA

Alaghebandha, M., Naderi, B., & Mohammadi, M. (2019). Economic lot sizing and scheduling in distributed permutation flow shops. *Journal of Optimization in Industrial Engineering*, *12*(1), 103–117. https://doi.org/10.22094/JOIE.2018.542997.1510

Ariza Gamero, M. (2021). *Análisis del problema Distributed Flow shop con permutación y objetivo Total Core Idle Time Trabajo Fin de Grado* (P. Pérez González, Ed.).

Chen, J. fang, Wang, L., & Peng, Z. ping. (2019). A collaborative optimization algorithm for energy-efficient multi-objective distributed no-idle flow-shop scheduling. *Swarm and Evolutionary Computation*, *50*. https://doi.org/10.1016/j.swevo.2019.100557

Fathollahi-Fard, A. M., Woodward, L., & Akhrif, O. (2021). Sustainable distributed permutation flow-shop scheduling model based on a triple bottom line concept. *Journal of Industrial Information Integration*, *24*. https://doi.org/10.1016/J.JII.2021.100233

Fernandez-Viagas, V., Dios, M., & Framinan, J. M. (2016). Efficient constructive and composite heuristics for the Permutation Flowshop to minimise total earliness and tardiness. *Computers and Operations Research*, *75*, 38–48. https://doi.org/10.1016/j.cor.2016.05.006

Fernandez-Viagas, V., & Framinan, J. M. (2015). A bounded-search iterated greedy algorithm for the distributed permutation flowshop scheduling problem. *International Journal of Production Research*, *53*(4), 1111–1123. https://doi.org/10.1080/00207543.2014.948578

Fernandez-Viagas, V., Perez-Gonzalez, P., & Framinan, J. M. (2018). The distributed permutation flow shop to minimise the total flowtime. *Computers and Industrial Engineering*, *118*, 464–477. https://doi.org/10.1016/j.cie.2018.03.014

Framinan, J. M., Leisten, R., & Ruiz García, R. (2014). *Manufacturing Scheduling Systems An Integrated View on Models, Methods and Tools*.

Gangadharan, R., & Rajendran, C. (1993). Heuristic algorithms for scheduling in the no-wait flowshop. *International Journal of Production Economics*, *32*(3), 285–290. https://doi.org/10.1016/0925-5273(93)90042-J

Graham, R. L., Lawler, E. L., Lenstra, J. K., & Kan, A. H. G. R. (1979). *OPTIMIZATION AND APPROXIMATION IN DETERMINISTIC SEQUENCING AND SCHEDULING: A SURVEY*. 287–326.

Johnson, S. M. (1954). Optimal two- and three-stage production schedules with setup times included. *Naval Research Logistics Quarterly*, *1*, 61–68.

Khare, A., & Agrawal, S. (2021). Effective heuristics and metaheuristics to minimise total tardiness for the distributed permutation flowshop scheduling problem. *International Journal of Production Research*, *59*(23), 7266–7282. https://doi.org/10.1080/00207543.2020.1837982

Librería SCHEDULE – Grupo de Investigación Organización Industrial. (n.d.). http://grupo.us.es/oindustrial/investigacion/software-y-librerias/libreria-schedule/

Lin, S. W., & Ying, K. C. (2016). Minimizing makespan for solving the distributed no-wait flowshop scheduling problem. *Computers and Industrial Engineering*, *99*, 202–209. https://doi.org/10.1016/j.cie.2016.07.027

Liu, J., & Reeves, C. R. (2001). Constructive and composite heuristic solutions to the P//∑Ci scheduling problem. *European Journal of Operational Research*, *132*(2), 439–452. https://doi.org/10.1016/S0377-2217(00)00137-5

Lu, C., Gao, L., Gong, W., Hu, C., Yan, X., & Li, X. (2021). Sustainable scheduling of distributed permutation flow-shop with non-identical factory using a knowledge-based multi-objective memetic optimization

algorithm. *Swarm and Evolutionary Computation*, *60*. https://doi.org/10.1016/J.SWEVO.2020.100803

Maassen, K., Perez-Gonzalez, P., & Günther, L. C. (2020). Relationship between common objective functions, idle time and waiting time in permutation flow shop scheduling. *Computers and Operations Research*, *121*. https://doi.org/10.1016/j.cor.2020.104965

Naderi, B., & Ruiz, R. (2010). The distributed permutation flowshop scheduling problem. *Computers and Operations Research*, *37*(4), 754–768. https://doi.org/10.1016/j.cor.2009.06.019

Naderi, B., & Ruiz, R. (2014). A scatter search algorithm for the distributed permutation flowshop scheduling problem. *European Journal of Operational Research*, *239*(2), 323–334. https://doi.org/10.1016/j.ejor.2014.05.024

Nawaz, M., & Enscore, E. E. (1983). A Heuristic Algorithm for the m-Machine, n-Job Flow-shop Sequencing Problem. In *JI of Mgmt Sci* (Vol. 11, Issue 1).

Pan, Q. K., Gao, L., Wang, L., Liang, J., & Li, X. Y. (2019). Effective heuristics and metaheuristics to minimize total flowtime for the distributed permutation flowshop problem. *Expert Systems with Applications*, *124*, 309–324. https://doi.org/10.1016/j.eswa.2019.01.062

Perez-Gonzalez, P., & Framinan, J. M. (2024). A review and classification on distributed permutation flowshop scheduling problems. In *European Journal of Operational Research* (Vol. 312, Issue 1, pp. 1–21). Elsevier B.V. https://doi.org/10.1016/j.ejor.2023.02.001

Pinedo, M. L. (2012). *Scheduling. Theory, Algorithms, and Systems*.

Ruiz, R., Pan, Q. K., & Naderi, B. (2019). Iterated Greedy methods for the distributed permutation flowshop scheduling problem. *Omega (United Kingdom)*, *83*, 213–222. https://doi.org/10.1016/j.omega.2018.03.004

Ruiz, R., Vallada, E., & Fernández-Martínez, C. (2009). Scheduling in flowshops with no-idle machines. *Studies in Computational Intelligence*, *230*, 21–51. https://doi.org/10.1007/978-3-642-02836-6_2

Wang, G., Gao, L., Li, X., Li, P., & Tasgetiren, M. F. (2020). Energy-efficient distributed permutation flow shop scheduling problem using a multi-objective whale swarm algorithm. *Swarm and Evolutionary Computation*, *57*. https://doi.org/10.1016/j.swevo.2020.100716

Wang, J. J., & Wang, L. (2020). A Knowledge-Based Cooperative Algorithm for Energy-Efficient Scheduling of Distributed Flow-Shop. *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, *50*(5), 1805–1819. https://doi.org/10.1109/TSMC.2017.2788879

Woollam, C. R. (1986). Flowshop with no idle machine time allowed. *Computers & Industrial Engineering*, *10*(1), 69–76. https://doi.org/10.1016/0360-8352(86)90028-8

Ying, K. C., Lin, S. W., Cheng, C. Y., & He, C. D. (2017). Iterated reference greedy algorithm for solving distributed no-idle permutation flowshop scheduling problems. *Computers and Industrial Engineering*, *110*, 413–423. https://doi.org/10.1016/j.cie.2017.06.025

ANEXO CÓDIGO PROGRAMACIÓN EN C

```
#include <schedule.h>
#include <stdio.h>
#include<conio.h>
#include<time.h>


MAT_INT loadDPFS (char *data, int *jobs, int *machs, int *facts);
void outputDPFS ( char *inputfile, char *outputfile, int time, int jobs, int machs, int facts, VECTOR_INT seq, MAT_INT seq_fabs, double objetive);
int CIT (VECTOR_INT seq, MAT_INT pt, int jobs, int m, int n);
int sumCIT (MAT_INT seqxfab, VECTOR_INT jobs_facts, MAT_INT pt, int jobs, int machs, int facts);
int A_4(VECTOR_INT *fab_insertion, MAT_INT seqxfab, VECTOR_INT jobs_facts, int job_to_insert, int f, int n, int m, MAT_INT pt);
void insertion (VECTOR_INT seq, VECTOR_INT *nueva_seq, int pos, int job_to_insert, int n);
VECTOR_INT matriz_a_vector(MAT_INT pi, int n, int f);
int makespan (VECTOR_INT seq, int n, int m, MAT_INT pt);
void copy_mat_int( MAT_INT original, MAT_INT destino, int filas, int cols);


void DLR_DNEH (VECTOR_INT *seq, VECTOR_INT *jobs_facts, MAT_INT *seq_fabs, MAT_INT pt, int n, int m, int f, float x);
void NEH2 (VECTOR_INT *seq, VECTOR_INT *jobs_facts, MAT_INT *seq_fabs, MAT_INT pt, int n, int m, int f);
void NEH2_en (VECTOR_INT *seq, VECTOR_INT *jobs_facts, MAT_INT *seq_fabs, MAT_INT pt, int n, int m, int f);
void NEH_R1_A4_en (VECTOR_INT *seq, VECTOR_INT *jobs_facts, MAT_INT *seq_fabs, MAT_INT pt, int n, int m, int f);
void ILS (VECTOR_INT *seq, VECTOR_INT *jobs_facts, MAT_INT *seq_fabs, MAT_INT pt, int n, int m, int f );
void RLS (MAT_INT *seqxfab, MAT_INT pt, VECTOR_INT *jobs_facts, int n, int m, int f);
void insertion_operator(MAT_INT *seqxfab, VECTOR_INT *jobs_facts, MAT_INT pt, int m, int n, int f);
void IG2S (VECTOR_INT *seq, VECTOR_INT *jobs_facts, MAT_INT *seq_fabs, MAT_INT pt, int n, int m, int f );
void LS3(MAT_INT *pi, MAT_INT pi_0, VECTOR_INT *jobs_facts_pi, VECTOR_INT jobs_facts_0, MAT_INT pt, int n, int m, int f);
void destruction (MAT_INT pi, VECTOR_INT jobs_facts_pi, MAT_INT pt, MAT_INT *pi_destruction, VECTOR_INT *jobs_facts_destruction, VECTOR_INT *pi_removed, int d, int n, int m, int f);
```

```
void reconstruction (MAT_INT pi_destruction,VECTOR_INT jobs_facts_destruction, MAT_INT *pi_reconstruction, VECTOR_INT *jobs_facts_reconstruction, VECTOR_INT pi_removed, MAT_INT pt, int n, int m, int f);



int main(int argc, char *argv[])
{
   int jobs, machs, facts;
   MAT_INT pt;

   pt=loadDPFS(argv[1],&jobs,&machs,&facts);

   VECTOR_INT seq= DIM_VECTOR_INT(jobs);
   setval_vector(seq,jobs,-1);
   MAT_INT seq_fabs=DIM_MAT_INT(facts,jobs);
   setval_matrix(seq_fabs,facts, jobs, -1);

   int objetive, time;
   VECTOR_INT jobs_facts=DIM_VECTOR_INT(facts);
   setval_vector(jobs_facts,facts,0);

   //METAHEURÍSTICAS

   clock_t clock (void);
   IG2S(&seq,&jobs_facts,&seq_fabs,pt,jobs,machs,facts);
   time=((double)clock()/CLK_TCK);
   printf("Secuencia IG2S\n");
   print_vector(seq,jobs);
   printf("Numero trabajos por fabrica");
   print_vector(jobs_facts,facts);
   printf("Matriz de trabajos por fabricas\n");
   print_matrix(seq_fabs,facts,jobs);
   objetive=sumCIT(seq_fabs,jobs_facts,pt,jobs,machs,facts);
   outputDPFS(argv[1],argv[2],time,jobs,machs,facts,seq,seq_fabs,objetive);

   clock_t clock (void);
   ILS(&seq,&jobs_facts,&seq_fabs,pt,jobs,machs,facts);
   time=((double)clock()/CLK_TCK);
```

```
printf("Secuencia ILS\n");
print_vector(seq,jobs);
printf("Numero trabajos por fabrica");
print_vector(jobs_facts,facts);
printf("Matriz de trabajos por fabricas\n");
print_matrix(seq_fabs,facts,jobs);
objetive=sumCIT(seq_fabs,jobs_facts,pt,jobs,machs,facts);
outputDPFS(argv[1],argv[2],time,jobs,machs,facts,seq,seq_fabs,objetive);



//HEURÍSTICAS CONSTRUCTIVAS

clock_t clock (void);
NEH_R1_A4_en(&seq,&jobs_facts,&seq_fabs,pt,jobs,machs,facts);
time=((double)clock()/CLK_TCK);
printf("Secuencia NEH_R1_A4_en\n");
print_vector(seq,jobs);
printf("Numero trabajos por fabrica");
print_vector(jobs_facts,facts);
printf("Matriz de trabajos por fabricas\n");
print_matrix(seq_fabs,facts,jobs);
objetive=sumCIT(seq_fabs,jobs_facts,pt,jobs,machs,facts);
outputDPFS(argv[1],argv[2],time,jobs,machs,facts,seq,seq_fabs,objetive);

clock_t clock (void);
NEH2_en(&seq,&jobs_facts,&seq_fabs,pt,jobs,machs,facts);
time=((double)clock()/CLK_TCK);
printf("Secuencia NEH2_en\n");
print_vector(seq,jobs);
printf("Numero trabajos por fabrica");
print_vector(jobs_facts,facts);
printf("Matriz de trabajos por fabricas\n");
print_matrix(seq_fabs,facts,jobs);
objetive=sumCIT(seq_fabs,jobs_facts,pt,jobs,machs,facts);
outputDPFS(argv[1],argv[2],time,jobs,machs,facts,seq,seq_fabs,objetive);
```

```
    clock_t clock (void);
    NEH2(&seq,&jobs_facts,&seq_fabs,pt,jobs,machs,facts);
    time=((double)clock()/CLK_TCK);
    printf("Secuencia NEH2\n");
    print_vector(seq,jobs);
    printf("Numero trabajos por fabrica");
    print_vector(jobs_facts,facts);
    printf("Matriz de trabajos por fabricas\n");
    print_matrix(seq_fabs,facts,jobs);
    objetive=sumCIT(seq_fabs,jobs_facts,pt,jobs,machs,facts);
    outputDPFS(argv[1],argv[2],time,jobs,machs,facts,seq,seq_fabs,objetive);


    clock_t clock (void);
    DLR_DNEH(&seq,&jobs_facts,&seq_fabs,pt,jobs,machs,facts,0.5);
    time=((double)clock()/CLK_TCK);
    printf("Secuencia DLR_DNEH\n");
    print_vector(seq,jobs);
    printf("Numero trabajos por fabrica");
    print_vector(jobs_facts,facts);
    printf("Matriz de trabajos por fabricas\n");
    print_matrix(seq_fabs,facts,jobs);
    outputDPFS(argv[1],argv[2],time,jobs,machs,facts,seq,seq_fabs,objetive

    free_vector(seq);
    free_vector(jobs_facts);
    free_matrix(pt,jobs);
    free_matrix(seq_fabs,facts);

    return 0;

}

//LECTURA DE ENTORNO DPFS
MAT_INT loadDPFS (char *data, int *jobs, int *machs, int *facts)
{
```

```
int temp_data;
register int i,j;
MAT_INT pt;

FILE *input;
input=fopen(data,"rt"); //Abre el archivo con el nombre dado y lee
if (input==NULL)
{
   printf("No se ha abierto el fichero\n");
}

//Lectura de número de trabajos, máquinas y fábricas
fscanf(input,"%d\t",&temp_data);
*jobs=temp_data;

fscanf(input,"%d\n",&temp_data);
*machs=temp_data;

fscanf(input,"%d\n",&temp_data);
*facts=temp_data;
//printf("JOBS:%d\t MACHINES: %d\nFACTORIES: %d\n",*jobs,*machs,*facts);

//Lectuta de tiempos de procesos
pt=DIM_MAT_INT(*jobs,*machs); //Tamaño de la matriz
for (j=0;j<*jobs;j++)
 {

 for (i=0;i<*machs;i++)
{
   fscanf(input,"%d\t",&temp_data);
   pt[j][i]=temp_data;
}
}
//print_matrix(pt,*jobs,*machs);
printf("Fin lectura de datos\n");

fclose(input);
```

```
   return pt;
}

//RESULTADOS ENTORNO DPFS
void outputDPFS ( char *inputfile, char *outputfile, int CPU, int jobs, int machs, int facts, VECTOR_INT seq,
MAT_INT seq_fabs, double objetive)
{
   FILE *output;
   register int j,k;

   output=fopen(outputfile,"at"); //Crea un archivo con el nombre dado en outputfile y escribe en él

   time_t timer;
   struct tm *tblock;

   timer = time(NULL);
   tblock = localtime(&timer);

   fprintf(output,"\n%s\t%s\n",inputfile,asctime(tblock));
   fprintf(output,"CPU(segundos) = %d\n",CPU);
   //Escritura valor FO
   fprintf(output,"FO=%f\n", objetive);

   //Escritura secuencia
   fprintf(output,"%s=(","Secuencia");
   for(j=0;j<jobs;j++)
   {
      fprintf(output,"\t%d",seq[j]);
   }

   //Escritura secuencia por fabrica
   fprintf(output,")\n%s","Secuencia por fabrica ");

   for(k=0;k<facts;k++)
   {
      fprintf(output,"\n%s %d","Fabrica",k);
      for(j=0;j<jobs;j++)
      {
```

```
            fprintf(output,"\t%d",seq_fabs[k][j]);
        }

    }
    printf("Fin escritura de resultados\n");
    fclose(output);
}

//CIT POR FÁBRICA
int CIT (VECTOR_INT seq, MAT_INT pt, int jobs, int m, int n)
{
    float objetive=0;
    int register i,j; //máquina-i, trabajo-j
    MAT_INT S=DIM_MAT_INT(n,m);
    setval_matrix(S,n,m,-1);
    MAT_INT ct=DIM_MAT_INT(n,m); // En una matriz se guardan los completion time de cada trabajo en
cada máquina
    setval_matrix(ct,n,m,0);
    if (seq[0]!=-1)
    {
        S[seq[0]][0]=0;
        for (j=1;j<jobs;j++) //En la primera máquina no hay CIT, un trabajo empieza cuando termina el anterior
        {
            S[seq[j]][0]+=S[seq[j-1]][0]+pt[seq[j-1]][0];
        }
        for (i=1;i<m;i++) //El primer trabajo secuenciado pasa a la siguiente máquina cuando termina el anterior
        {
            S[seq[0]][i]=S[seq[0]][i-1]+pt[seq[0]][i-1];
        }
        for (i=1;i<m;i++) //A partir de la segunda máquina y el segundo trabajo se generan los CIT
        {
            for (j=1;j<jobs;j++)
            {
                if(S[seq[j]][i-1]+pt[seq[j]][i-1]<S[seq[j-1]][i]+pt[seq[j-1]][i])
                    {
                        S[seq[j]][i]=S[seq[j-1]][i]+pt[seq[j-1]][i];
                    }
```

```
            else
            {
               S[seq[j]][i]=S[seq[j]][i-1]+pt[seq[j]][i-1];
            }
      }
   }

printf("Matriz start time\n");
print_matrix(S,n,m);

   for(j=0;j<jobs;j++) //Completion time por trabajos
   {
      for (i=0;i<m;i++)
      {
         ct[seq[j]][i]=S[seq[j]][i]+pt[seq[j]][i];
      }
   }
printf("Matriz completion time trabajos\n");
print_matrix(ct,n,m);


   for (i=1;i<m;i++) //Calculo CIT
   {
      for(j=1;j<jobs;j++)
      {
         if (S[seq[j]][i]>ct[seq[j-1]][i]) //Si la diferencia es positiva, entonces se va sumando
         {
            objetive+=S[seq[j]][i]-ct[seq[j-1]][i];
         }
      }
   }



}

free_matrix(S,n);
free_matrix(ct,n);
```

```
   printf("Calculo de CIT=%f\n",objetive);


   return objetive;
}


//FO SUM (CIT)
int sumCIT (MAT_INT seqxfab, VECTOR_INT jobs_facts, MAT_INT pt, int jobs, int machs, int facts)
{
   float objetive=0;
   int register i,j,f; //máquina-i, trabajo-j, fábrica-f
   MAT_INT C=DIM_MAT_INT(facts,machs); // En un matriz se van a guardar los completion time de cada máquina en cada fábrica
   setval_matrix(C,facts,machs,0);
   MAT_INT S=DIM_MAT_INT(jobs,machs);
   setval_matrix(S,jobs,machs,0);
   MAT_INT ct=DIM_MAT_INT(jobs,machs); // En una matriz se guardan los completion time de cada trabajo en cada máquina
   setval_matrix(ct,jobs,machs,0);


   for(f=0;f<facts;f++) //Para cada fábrica
   {
      if (seqxfab[f][0]!=-1)
      {
         S[seqxfab[f][0]][0]=0;
         for (j=1;j<jobs_facts[f];j++) //En la primera máquina no hay CIT, un trabajo empieza cuando termina el anterior
         {
            S[seqxfab[f][j]][0]+=S[seqxfab[f][j-1]][0]+pt[seqxfab[f][j-1]][0];
         }
            C[f][0]=S[seqxfab[f][j-1]][0]+pt[seqxfab[f][j-1]][0]; //Completion time de la máquina 0


         for (i=1;i<machs;i++) //El primer trabajo secuenciado pasa a la siguiente máquina cuando termina el anterior
         {
            S[seqxfab[f][0]][i]=S[seqxfab[f][0]][i-1] + pt[seqxfab[f][0]][i-1];
         }


         for (i=1;i<machs;i++) //A partir de la segunda máquina y el segundo trabajo se generan los CIT
```

```
        {
          for (j=1;j<jobs_facts[f];j++)
          {
            if(S[seqxfab[f][j]][i-1]+pt[seqxfab[f][j]][i-1]<S[seqxfab[f][j-1]][i]+pt[seqxfab[f][j-1]][i])
            {
              S[seqxfab[f][j]][i]=S[seqxfab[f][j-1]][i]+pt[seqxfab[f][j-1]][i]; //El trabajo anterior en la máquina tarda más que el mismo trabajo en la máquina anterior
            }
            else
            {
              S[seqxfab[f][j]][i]=S[seqxfab[f][j]][i-1]+pt[seqxfab[f][j]][i-1]; //El trabajo en la máquina anterior tarda más que el anterior en la máquina
            }
          }
          C[f][i]=S[seqxfab[f][j-1]][i]+pt[seqxfab[f][j-1]][i]; //Completion time de la máquina i
        }

      }


    printf("Matriz completion time maquinas\n");
    print_matrix(C,facts,machs);

    printf("Matriz start time\n");
    print_matrix(S,jobs,machs);

  }
  for(f=0;f<facts;f++)
  {
     for(j=0;j<jobs_facts[f];j++) //Completion time por trabajos
     {
       if(seqxfab[f][j]!=-1)
       {
          for (i=0;i<machs;i++)
          {
            ct[seqxfab[f][j]][i]=S[seqxfab[f][j]][i]+pt[seqxfab[f][j]][i];
          }
```

```
        }

      }

   }

   printf("Matriz completion time trabajos\n");
   print_matrix(ct,jobs,machs);



   for (i=1;i<machs;i++) //Calculo CIT
   {
      for(f=0;f<facts;f++)
      {
         for(j=1;j<jobs;j++)
         {
            if (seqxfab[f][j]!=-1 && seqxfab[f][j-1]!=-1 && S[seqxfab[f][j]][i]>ct[seqxfab[f][j-1]][i]) //Si la diferencia es positiva, entonces se va sumando
            {
               objetive+=S[seqxfab[f][j]][i]-ct[seqxfab[f][j-1]][i];
            }
         }

      }

   }

   free_matrix(C, facts);
   free_matrix(S,jobs);
   free_matrix(ct, jobs);

   printf("Calculo de sumCIT=%f\n",objetive);

   return objetive;
}

void insertion (VECTOR_INT seq, VECTOR_INT *nueva_seq, int pos, int job_to_insert, int n)
```

```
{
   copy_vector(seq,*nueva_seq,n);
   extract_vector(*nueva_seq,n,n-1); //Extrae el valor -1 de la ultima posicion de la secuencia
   insert_vector(*nueva_seq,n,job_to_insert, pos);
}

VECTOR_INT matriz_a_vector(MAT_INT pi, int n, int f)
{
   VECTOR_INT secuencia=DIM_VECTOR_INT(n);
   int j,k;
   int pos=0;

   for(k=0;k<f;k++)
   {
      for(j=0;j<n;j++)
      {
         if(pi[k][j]!=-1)
         {
            secuencia[pos]=pi[k][j];
            pos++;
         }

      }

   }
   return secuencia;
}

int A_4(VECTOR_INT *fab_insertion, MAT_INT seqxfab, VECTOR_INT jobs_facts, int job_to_insert, int f,
int n, int m, MAT_INT pt)
{
   VECTOR_INT copia_fila_fab= DIM_VECTOR_INT(n);
   VECTOR_INT dif_CIT_sum = DIM_VECTOR_INT(f);
   setval_vector(dif_CIT_sum,f,-1);
   VECTOR_INT pos_f = DIM_VECTOR_INT(f);
   setval_vector(pos_f,f,-1);
   MAT_INT copia_seqxfab=DIM_MAT_INT(f,n);
```

```
register int j,k;
int best_f=-1, best_pos, max_dif=-1, min_CIT_sum, CIT_sum,CIT_sum_ini;

CIT_sum_ini=sumCIT(seqxfab,jobs_facts,pt,n,m,f);
printf("min sumCIT de la fabrica se da antes de insertar el trabajo = %d\n", CIT_sum_ini);

for(k=0;k<f;k++)
{
   for (j=0;j<n;j++)
   {

      copia_fila_fab[j]=seqxfab[k][j];

   }

   if(jobs_facts[k]<n)
   {
      pos_f[k]=0;
      copy_mat_int(seqxfab,copia_seqxfab,f,n);
      insertion(copia_fila_fab,&(*fab_insertion),0,job_to_insert,n);
      jobs_facts[k]++;
      pasterowImatrix(copia_seqxfab,*fab_insertion,k,n);
      min_CIT_sum=sumCIT(copia_seqxfab, jobs_facts, pt,n,m,f);
      jobs_facts[k]--;

      for(j=1;j<=jobs_facts[k];j++)
      {
         copy_mat_int(seqxfab,copia_seqxfab,f,n);
         insertion(copia_fila_fab,&(*fab_insertion),j,job_to_insert,n);
         jobs_facts[k]++;
         pasterowImatrix(copia_seqxfab,*fab_insertion,k,n);
         CIT_sum=sumCIT(copia_seqxfab, jobs_facts, pt,n,m,f);
         jobs_facts[k]--;
         if (CIT_sum<=min_CIT_sum)
         {
            min_CIT_sum=CIT_sum;
            pos_f[k]=j;
```

```
            dif_CIT_sum[k]=CIT_sum_ini-min_CIT_sum;
            printf("Min CIT se ha actualizado, es %d y la mejor pos es %d para la fab %d\n dif_CIT_sum[%d]=%d\n", min_CIT_sum, pos_f[k],k,k,dif_CIT_sum[k]);
          }
        }
      }
      dif_CIT_sum[k]=CIT_sum_ini-min_CIT_sum;

  }

  for(k=0;k<f;k++)
  {
    if(best_f==-1 && pos_f[k]!=-1 )
    {
      best_f=k;
      best_pos=pos_f[k];
      max_dif=dif_CIT_sum[k];
      k=f; //Para salir del bucle cuando entre por primera vez en if
    }

  }

  for(k=best_f;k<f;k++)
  {
    if(dif_CIT_sum[k]>=max_dif && pos_f[k]!=-1)
    {
      max_dif=dif_CIT_sum[k];
      best_f=k;
      best_pos=pos_f[k];
      printf("Se ha actualizado max_dif=%d y best_f=%d, best_pos=%d\n", max_dif, best_f,best_pos);

    }
  }

  for (j=0;j<n;j++)
  {
    copia_fila_fab[j]=seqxfab[best_f][j];
  }
```

```
   insertion(copia_fila_fab,&(*fab_insertion),best_pos,job_to_insert,n);

   free_vector(copia_fila_fab);
   free_vector(dif_CIT_sum);
   free_vector(pos_f);
   free_matrix(copia_seqxfab,f);

   return best_f;
}

void copy_mat_int( MAT_INT original, MAT_INT destino, int filas, int cols)
{
   register int i,j;

   for (i=0;i<filas;i++)
   {
      for(j=0;j<cols;j++)
      {
         destino[i][j]=original[i][j];
      }
   }
   //printf("Sale función copy_may_int\n");
}

//HEURISTICAS CONSTRUCTIVAS TFM
void DLR_DNEH (VECTOR_INT *seq, VECTOR_INT *jobs_facts, MAT_INT *seq_fabs, MAT_INT pt, int
n, int m, int f, float x)
{
   printf("\n******Calculo de la secuencia DLR_DNEH******\n");
   float n_copia=n;
   float f_copia=f;
   float n_f=n_copia/f_copia;

   VECTOR_FLOAT IF0= DIM_VECTOR_FLOAT(n);
   VECTOR_FLOAT IT0= DIM_VECTOR_FLOAT(n);
   VECTOR_FLOAT C= DIM_VECTOR_FLOAT(n);
```

```
VECTOR_INT landa= DIM_VECTOR_INT(n);
MAT_INT pi= DIM_MAT_INT(f,n);
register int i,j,k;

for (k=0;k<f;k++)
{
   for(j=0;j<n;j++)
   {

      pi[k][j]=-1;
      IF0[j]=0;
      IT0[j]=0;
      C[j]=0;
      landa[j]=-1;
   }
}
int aux, dim=n; // para extraer los primeros f trabajos

//CÁLCULO DEL ÍNDICE IF0
for (j=0;j<n;j++)
{
   for (i=1;i<m;i++)
   {
      C[j]+=pt[j][i-1];
      IT0[j]+=(m*(C[j]))/(i);
   }
   C[j]+=pt[j][m-1]; // Makespan de todos los trabajos si se secuenciaran los primeros
   IF0[j]=((n_f)-2)*IT0[j]+ C[j];

}

//TRABAJOS ORDENADOS DE MANERA CRECIENTE SEÚN IF0
printf("Vector IF0\t");
print_vector(IF0,n);
sort_vector(IF0,landa,n,'A'); // Vector de trabajos landa ordenado de manera creciente
printf("Vector landa\t");
print_vector(landa,n);
```

```
//ASIGNAR LOS PRIMEROS f TRABAJOS A LAS f FÁBRICAS
for(j=0;j<f;j++)
{
   pi[j][0]=landa[0];
   aux=extract_vector(landa,dim,0); //Extraer los trabajos asignados de landa
   dim--;
}

setval_vector(*jobs_facts,f,1);
*seq=matriz_a_vector(pi,n,f);

int C_max, min_C_max, k_opt;
int max=0, lim=n*x, j_opt, pos;
int first_job, best_fo, k_inserted, fo, l;
float min_IF_nk;
VECTOR_INT pi_k_vector = DIM_VECTOR_INT(n);
VECTOR_INT jobs_facts_copia =DIM_VECTOR_INT(f);
MAT_INT S=DIM_MAT_INT(n,m);
MAT_INT CT =DIM_MAT_INT(f,n);
setval_matrix(CT,f,n,0);
setval_matrix(S,n,m,-1);
copy_vector(*jobs_facts,jobs_facts_copia,f);
VECTOR_FLOAT IF_nk=DIM_VECTOR_FLOAT(n);
setval_vector(IF_nk,n,0);
VECTOR_FLOAT IT_nk=DIM_VECTOR_FLOAT(n);
setval_vector(IT_nk,n,0);
VECTOR_INT best_pi_k_insertion = DIM_VECTOR_INT(n);
VECTOR_INT pi_k_insertion = DIM_VECTOR_INT(n);


while(dim>lim) // Trabajos a insertar por DLR
{
   printf("Entra en bucle DLR\n");
   //CÁLCULO DE COMPLETION TIME
   for(k=0;k<f;k++)
   {
```

```
for(j=0;j<n;j++) //Copia de la fila de la matriz pi en un vector
{
  pi_k_vector[j]=pi[k][j];

}
S[pi_k_vector[0]][0]=0; //En el instante cero el primer trabajo secuanciado en la fabrica

for (j=1;j<jobs_facts_copia[k];j++) //En la primera máquina no hay tiempos ociosos, un trabajo empieza cuando termina el anterior
{

  S[pi_k_vector[j]][0]+=S[pi_k_vector[j-1]][0]+pt[pi_k_vector[j-1]][0];
}
for (i=1;i<m;i++) //El primer trabajo secuenciado pasa a la siguiente máquina cuando termina el anterior
{
  S[pi_k_vector[0]][i]=S[pi_k_vector[0]][i-1] + pt[pi_k_vector[0]][i-1];
}
CT[k][pi_k_vector[0]]= S[pi_k_vector[0]][m-1]+pt[pi_k_vector[0]][m-1];  //Completion time del primer trabajo secuenciado en la fábrica k

for (j=1;j<jobs_facts_copia[k];j++) //A partir del segundo trabajo y la segunda máquina se generan los CIT
{
  for (i=1;i<m;i++)
  {
    if(S[pi_k_vector[j]][i-1]+pt[pi_k_vector[j]][i-1]<S[pi_k_vector[j-1]][i]+pt[pi_k_vector[j-1]][i])
    {
      S[pi_k_vector[j]][i]=S[pi_k_vector[j-1]][i]+pt[pi_k_vector[j-1]][i]; //El trabajo anterior en la máquina tarda más que el mismo trabajo en la máquina anterior
    }
    else
    {
      S[pi_k_vector[j]][i]=S[pi_k_vector[j]][i-1]+pt[pi_k_vector[j]][i-1]; //El trabajo en la máquina anterior tarda más que el anterior en la máquina
    }
  }
  CT[k][pi_k_vector[j]]= S[pi_k_vector[j]][m-1]+pt[pi_k_vector[j]][m-1];
}
}
```

```
printf("Matriz CT, f filas y n columnas\n");
print_matrix(CT,f,n);
printf("Matriz S, n filas y m maquinas\n");
print_matrix(S,n,m);
//MENOR MAKESPAN
min_C_max=CT[0][pi[0][jobs_facts_copia[0]-1]]; //CT del ultimo trabajo secuenciado en la fábrica 0
printf("min_makespan_calculado=%d\n",min_C_max);
k_opt=0;
for (k=1;k<f;k++)
{
  C_max=CT[k][pi[k][jobs_facts_copia[k]-1]];
  if (C_max<min_C_max)
  {
    min_C_max=C_max;
    k_opt=k;

  }
}

//BUSCAR j ÓPTIMO PARA LA FÁBRICA k_opt
for (j=0;j<dim;j++)
{
  for(i=1;i<m;i++)
  {

if (S[jobs_facts_copia[k_opt]][i]+pt[jobs_facts_copia[k_opt]][i]<S[landa[j]][i-1]+pt[landa[j]][i-1])
        {

                                                                                max=
S[landa[j]][i-1]+pt[landa[j]][i-1]-S[jobs_facts_copia[k_opt]][i]-pt[jobs_facts_copia[k_opt]][i];
        }
        else
        {
            max=0;
        }

IT_nk[landa[j]]+=(m*max)/(i+(jobs_facts_copia[k_opt])*(m-i)*((n_f)-2));
    }
```

```
        IF_nk[landa[j]]=((n_f)-jobs_facts_copia[k_opt]-2)*IT_nk[landa[j]]+min_C_max;

    }
    printf("Vector IF_nk\n");
    print_vector(IF_nk,n);
    min_IF_nk=IF_nk[landa[0]];
    j_opt=landa[0];
    pos=0;
    for (j=1;j<n;j++) // Comparar los IF_nk para obtener el trabajo j
      {
        if (IF_nk[landa[j]]<min_IF_nk && IF_nk[landa[j]]!=0 )
        {
          min_IF_nk=IF_nk[landa[j]];
          j_opt=landa[j];
        }
      }
    setval_vector(IF_nk,n,0);

                                                                                    //INS
ERTAR AL FINAL DE LA SECUENCIA DE k_opt EL TRBAJO j_opt
    pi[k_opt][jobs_facts_copia[k_opt]]=j_opt;
    jobs_facts_copia[k_opt]++;

    //EXTRAER EL TRABAJO j_opt DE LA SECUENCIA landa
    pos=search_vector(landa,dim,j_opt);
    aux=extract_vector(landa,dim,pos);
    dim--;

  }

  while (dim>0)   //El resto de trabajos se secuencian mediante DNEH
  {
    printf("Entra en bucle DNEH\n");
    first_job=landa[0];

    //Suponemos que la mejor fo es el trabajo insertado en la primera posicion de la primera fábrica para ir
comparando
    for(j=0;j<n;j++) // Copia de la fila de la matriz pi en un vector
```

```
{
   pi_k_vector[j]=pi[0][j];
}
insertion(pi_k_vector,&best_pi_k_insertion,0,first_job,n);
for(l=0;l<n;l++) // Copia del vector insertado en la matriz pi
{
   pi[0][l]=best_pi_k_insertion[l];
}
jobs_facts_copia[0]++;
best_fo = sumCIT(pi,jobs_facts_copia,pt,n,m,f);
k_inserted=0;
jobs_facts_copia[0]--;

for(j=0;j<n;j++) // Copia de la fila de la matriz pi en un vector
   {
      pi[0][j]=pi_k_vector[j];

   }
printf("Matriz pi tras conocer best_fo\n");
print_matrix(pi,f,j);
for (k=0;k<f;k++) //Para todas las fábricas
{

   for(j=0;j<n;j++) // Copia de la fila de la matriz pi en un vector
   {
      pi_k_vector[j]=pi[k][j];

   }

   for(j=0;j<=jobs_facts_copia[k];j++) //Insertar el primer trabajo de landa en todas las posiciones de la fabrica a evaluar
   {

      insertion(pi_k_vector,&pi_k_insertion,j,first_job,n);
      for(l=0;l<n;l++) // Copia del vector insertado en la matriz pi
      {
         pi[k][l]=pi_k_insertion[l];
```

```
            }
            printf("Matriz pi tras insertion\n");
            print_matrix(pi,f,n);
            jobs_facts_copia[k]++;
            fo = sumCIT(pi,jobs_facts_copia,pt,n,m,f);
            if (fo<best_fo) //Actualizacion de la mejor fo quedandonos siempre con la primera
            {
               best_fo=fo;
               k_inserted=k;
               jobs_facts_copia[k]--;
               copy_vector(pi_k_insertion,best_pi_k_insertion,n);
            }
            else
            {
               jobs_facts_copia[k]--;
            }

         }

         for(j=0;j<n;j++) // Copia de la fila de la matriz pi en un vector
         {
            pi[k][j]=pi_k_vector[j];

         }
      }
   for(j=0;j<n;j++)
   {
      pi[k_inserted][j]=best_pi_k_insertion[j];

   }
   extract_vector(landa,dim,0); // Extraer el trabajo insertado
   jobs_facts_copia[k_inserted]++; //Actualización del vector de trabajos asignados por fábrica
   dim--;
   }
```

```
   copy_vector(jobs_facts_copia,*jobs_facts,f);
   copy_mat_int(pi,*seq_fabs, f, n);
   *seq=matriz_a_vector(pi,n,f);

   free_vector(IF0);
   free_vector(IT0);
   free_vector(C);
   free_vector(landa);
   free_vector(pi_k_vector);
   free_vector(jobs_facts_copia);
   free_vector(IF_nk);
   free_vector(IT_nk);

   free_vector(best_pi_k_insertion);
   free_vector(pi_k_insertion);
   free_matrix(pi,f);
   free_matrix(S,n);
   free_matrix(CT,f);

}
void NEH2 (VECTOR_INT *seq, VECTOR_INT *jobs_facts, MAT_INT *seq_fabs, MAT_INT pt, int n, int
m, int f)
{
   VECTOR_INT sum_pt=DIM_VECTOR_INT(n);
   setval_vector(sum_pt,n,0);
   VECTOR_INT omega=DIM_VECTOR_INT(n);
   VECTOR_INT jobs_facts_copia=DIM_VECTOR_INT(f);
   setval_vector(jobs_facts_copia,f,0);
   MAT_INT seqxfab=DIM_MAT_INT(f,n);
   setval_matrix(seqxfab,f,n,-1);
   register int i,j, k;
   int dim=n;

   //SUMA DE TIEMPOS DE TRABAJO
   for(j=0;j<n;j++)
   {
      for(i=0;i<m;i++)
```

```
    {
        sum_pt[j]+=pt[j][i];
    }
}

//VECTOR pt ORDENADO DE MANERA DESCENDENTE
sort_vector(sum_pt,omega, n,'D');
printf("Vector ordenado suma pt\n");
print_vector(omega,n);

seqxfab[0][0]=omega[0];
copy_mat_int(seqxfab,*seq_fabs,f,n);
jobs_facts_copia[0]++;
dim=extract_vector(omega,n,0);

VECTOR_INT copia_fila_fab= DIM_VECTOR_INT(n);
VECTOR_INT fab_insertion= DIM_VECTOR_INT(n);
setval_vector(fab_insertion,n,-1);
VECTOR_INT CIT_f=DIM_VECTOR_INT(f);
setval_vector(CIT_f,f,-1);
VECTOR_INT posCIT_f=DIM_VECTOR_INT(f);
setval_vector(posCIT_f,f,-1);
int min_CIT, min_pos, best_f=-1;

while(dim>0)
{
    for(k=0;k<f;k++)
    {
        for(j=0;j<n;j++)
        {
            copia_fila_fab[j]=seqxfab[k][j];
        }
        if(jobs_facts_copia[k]<n)
        {
            insertion(copia_fila_fab,&fab_insertion,0,omega[0],n);
            jobs_facts_copia[k]++;
            min_CIT=CIT(fab_insertion,pt,jobs_facts_copia[k], m, n);
```

```
            posCIT_f[k]=0;

            for (j=1;j<jobs_facts_copia[k];j++)
            {
               insertion(copia_fila_fab,&fab_insertion,j,omega[0],n);
               CIT_f[k]=CIT(fab_insertion,pt,jobs_facts_copia[k], m, n);
               if(CIT_f[k]<min_CIT)
               {
                  min_CIT=CIT_f[k];
                  posCIT_f[k]=j;
               }

            }
            CIT_f[k]=min_CIT;
            jobs_facts_copia[k]--;
         }

      }

      //ENCONTRAR FAB CON MENOR CIT
      for(k=0;k<f;k++)
      {
         if(best_f==-1 && posCIT_f[k]!=-1 )
         {
            best_f=k;
            min_pos=posCIT_f[k];
            min_CIT=CIT_f[k];
            k=f; //Para salir del bucle cuando entre por primera vez en if
         }

      }

      for(k=best_f;k<f;k++)
      {
         if(CIT_f[k]<min_CIT && posCIT_f[k]!=-1) // Con la expresion menor estamos saturando las primeras
fábricas
         {
```

```
            min_CIT=CIT_f[k];
            best_f=k;
            min_pos=posCIT_f[k];

        }
    }

    //INSERTAMOS EL TRABAJO EN LA FAB Y POS ENCONTRADA
    for(j=0;j<n;j++)
    {
       copia_fila_fab[j]=seqxfab[best_f][j];
    }
    insertion(copia_fila_fab,&fab_insertion,min_pos,omega[0],n);
    pasterowImatrix(seqxfab,fab_insertion,best_f,n);
    jobs_facts_copia[best_f]++;
    printf("Matriz de seq por fab tras insertion\n");
    print_matrix(seqxfab,f,n);
    dim=extract_vector(omega,dim,0);

    best_f=-1;
    setval_vector(posCIT_f,f,-1);
    setval_vector(CIT_f,f,-1);
}

copy_vector(jobs_facts_copia,*jobs_facts,f);
copy_mat_int(seqxfab,*seq_fabs, f, n);
*seq=matriz_a_vector(seqxfab,n,f);

free_vector(sum_pt);
free_vector(omega);
free_vector(jobs_facts_copia);
free_vector(copia_fila_fab);
free_vector(fab_insertion);
free_vector(CIT_f);
free_vector(posCIT_f);
free_matrix(seqxfab,f);
```

```
}

void NEH2_en (VECTOR_INT *seq, VECTOR_INT *jobs_facts, MAT_INT *seq_fabs, MAT_INT pt, int n,
int m, int f)
{
   VECTOR_INT sum_pt=DIM_VECTOR_INT(n);
   setval_vector(sum_pt,n,0);
   VECTOR_INT omega=DIM_VECTOR_INT(n);
   VECTOR_INT jobs_facts_copia=DIM_VECTOR_INT(f);
   setval_vector(jobs_facts_copia,f,0);
   MAT_INT seqxfab=DIM_MAT_INT(f,n);
   setval_matrix(seqxfab,f,n,-1);
   register int i,j, k;
   int dim=n;

   //SUMA DE TIEMPOS DE TRABAJO
   for(j=0;j<n;j++)
   {
      for(i=0;i<m;i++)
      {
         sum_pt[j]+=pt[j][i];
      }
   }

   //VECTOR pt ORDENADO DE MANERA DESCENDENTE
   sort_vector(sum_pt,omega, n,'D');
   printf("Vector ordenado suma pt\n");
   print_vector(omega,n);

   seqxfab[0][0]=omega[0];
   copy_mat_int(seqxfab,*seq_fabs,f,n);
   jobs_facts_copia[0]++;
   dim=extract_vector(omega,n,0);

   VECTOR_INT copia_fila_fab= DIM_VECTOR_INT(n);
   VECTOR_INT fab_insertion= DIM_VECTOR_INT(n);
   setval_vector(fab_insertion,n,-1);
```

```
VECTOR_INT CIT_f=DIM_VECTOR_INT(f);
setval_vector(CIT_f,f,-1);
VECTOR_INT posCIT_f=DIM_VECTOR_INT(f);
setval_vector(posCIT_f,f,-1);
int min_CIT, min_pos, best_f=-1;
int n_al, h=-1,aux, job_h;

while(dim>0)
{
   for(k=0;k<f;k++)
   {
      for(j=0;j<n;j++)
      {
         copia_fila_fab[j]=seqxfab[k][j];
      }

      if(jobs_facts_copia[k]<n)
      {
         insertion(copia_fila_fab,&fab_insertion,0,omega[0],n);
         jobs_facts_copia[k]++;
         min_CIT=CIT(fab_insertion,pt,jobs_facts_copia[k], m, n);
         posCIT_f[k]=0;

         for (j=1;j<jobs_facts_copia[k];j++)
         {
            insertion(copia_fila_fab,&fab_insertion,j,omega[0],n);
            CIT_f[k]=CIT(fab_insertion,pt,jobs_facts_copia[k], m, n);
            if(CIT_f[k]<min_CIT)
            {
               min_CIT=CIT_f[k];
               posCIT_f[k]=j;
            }

         }
         CIT_f[k]=min_CIT;
         printf("Se guarda el minCIT de la fabrica en el vector CIT_f\n");
         print_vector(CIT_f,f);
```

```
            print_vector(posCIT_f,f);
            jobs_facts_copia[k]--;
        }

    }

    //ENCONTRAR FAB CON MENOR CIT
    for(k=0;k<f;k++)
    {
        if(best_f==-1 && posCIT_f[k]!=-1 )
        {
            best_f=k;
            min_pos=posCIT_f[k];
            min_CIT=CIT_f[k];
            k=f; //Para salir del bucle cuando entre por primera vez en if
        }

    }

    for(k=best_f;k<f;k++)
    {
        if(CIT_f[k]<min_CIT && posCIT_f[k]!=-1) // Con la expresion menor estamos saturando las primeras
fábricas
        {
            min_CIT=CIT_f[k];
            best_f=k;
            min_pos=posCIT_f[k];
            //printf("Se ha actualizado min_CIT=%d, best_f=%d y min_pos=%d\n", min_CIT, best_f,min_pos);

        }
    }

    //INSERTAMOS EL TRABAJO EN LA FAB Y POS ENCONTRADA
    for(j=0;j<n;j++)
    {
        copia_fila_fab[j]=seqxfab[best_f][j];
    }
```

```
insertion(copia_fila_fab,&fab_insertion,min_pos,omega[0],n);
pasterowImatrix(seqxfab,fab_insertion,best_f,n);
jobs_facts_copia[best_f]++;
printf("Matriz de seq por fab tras insertion\n");
print_matrix(seqxfab,f,n);
printf("El CIT de la seq %d es %d\n", best_f, min_CIT);
dim=extract_vector(omega,dim,0);

//CASUISTICAS POSIBLES DE h
n_al=rand()%2;
if (min_pos==0 && jobs_facts_copia[best_f]>1)
{
   h=1; //Hay más de dos trabajos en la secuencia
}
else if (min_pos==(jobs_facts_copia[best_f]-1))
{
   h=jobs_facts_copia[best_f]-2;
}
else if( 0<min_pos<(jobs_facts_copia[best_f]-1) && n_al==0)
{
   h=min_pos+1;
}
else if (0<min_pos<(jobs_facts_copia[best_f]-1) && n_al==1)
{
   h=min_pos-1;
}
printf("h=%d\t min_pos=%d\n",h, min_pos);

if(h!=-1)
{
   for(j=0;j<n;j++)
   {
      copia_fila_fab[j]=seqxfab[best_f][j];
   }
   printf("Copia secuencia fabrica %d\n",best_f);
   print_vector(copia_fila_fab,n);
   min_CIT=CIT(copia_fila_fab,pt,jobs_facts_copia[best_f], m, n);
```

```
printf("Se supone minCIT la seq anterior sin improvement\tminCIT=%d\n",min_CIT);
posCIT_f[best_f]=h;
job_h=copia_fila_fab[h];
aux=extract_vector(copia_fila_fab,n,h);

for (j=0;j<jobs_facts_copia[best_f];j++)
{

   insert_vector(copia_fila_fab, n, job_h, j);
   printf("Secuencia copia_fila_fab tras insertion en la pos %d\n",j);
   print_vector(copia_fila_fab,n);
   CIT_f[best_f]=CIT(copia_fila_fab,pt,jobs_facts_copia[best_f], m, n);
   printf("CIT de la de seq anterior es %d\n", CIT_f[best_f]);
   if(CIT_f[best_f]<min_CIT)
   {
      min_CIT=CIT_f[best_f];
      posCIT_f[best_f]=j;
   }
   aux=extract_vector(copia_fila_fab,n,j);
}

//TRAS CONOCER EL MEJOR CIT DE LA FABRICA SE GUARDA
CIT_f[best_f]=min_CIT;
printf("EL CIT tras improvement %d\n",CIT_f[best_f]);

//INSERTAMOS EL TRABAJO job_h EN LA POS QUE MEJORA EL CIT DE LA FAB
for(j=0;j<n;j++)
{
   copia_fila_fab[j]=seqxfab[best_f][j];
}
aux=extract_vector(copia_fila_fab,n,h);
insert_vector(copia_fila_fab, n, job_h, posCIT_f[best_f]);
pasterowImatrix(seqxfab,copia_fila_fab,best_f,n);

printf("La secuencia de best_f=%d es la siguiente\n", best_f);
print_vector(copia_fila_fab,n);
```

```
        }

        best_f=-1;
        setval_vector(posCIT_f,f,-1);
        setval_vector(CIT_f,f,-1);

    }

    copy_vector(jobs_facts_copia,*jobs_facts,f);
    copy_mat_int(seqxfab,*seq_fabs, f, n);
    *seq=matriz_a_vector(seqxfab,n,f);

    free_vector(sum_pt);
    free_vector(omega);
    free_vector(jobs_facts_copia);
    free_vector(copia_fila_fab);
    free_vector(fab_insertion);
    free_vector(CIT_f);
    free_vector(posCIT_f);
    free_matrix(seqxfab,f);

}

void NEH_R1_A4_en (VECTOR_INT *seq, VECTOR_INT *jobs_facts, MAT_INT *seq_fabs, MAT_INT pt,
int n, int m, int f)
{
    printf("\n******Calculo de la secuencia NEH(R1, A4)_en******\n");
    VECTOR_INT sum_pt= DIM_VECTOR_INT(n);
    setval_vector(sum_pt,n,0);
    VECTOR_INT omega=DIM_VECTOR_INT(n);
    setval_vector(omega,n,-1);
    VECTOR_INT pi_wkl=DIM_VECTOR_INT(n);
    setval_vector(pi_wkl,n,-1);
    VECTOR_INT jobs_facts_copia=DIM_VECTOR_INT(f);
    setval_vector(jobs_facts_copia,f,0);
    MAT_INT seqxfab=DIM_MAT_INT(f,n);
    setval_matrix(seqxfab,f,n,-1);
    int new_dim, dim_pi;
```

```
register int i,j,l;

//TRABAJOS ORDENADOS POR SUMA DE pt
for (j=0;j<n;j++)
{
   for (i=0;i<m;i++)
   {
      sum_pt[j]+=pt[j][i];
   }
}

sort_vector(sum_pt,omega,n,'D');

//ASIGNAR TRABAJO CON MAYOR pt A pi
*seq[0]=omega[0];
dim_pi=1;
seqxfab[0][0]=omega[0]; //Primer trabajo en la primera posicion de la primera fábrica
copy_mat_int(seqxfab,*seq_fabs,f,n);
new_dim=extract_vector(omega,n,0);//Extrae el trabajo de omega secuenciado en pi
int aux=new_dim;

int CIT_seq, min_CIT, f_asignada;
VECTOR_INT best_pi_wkl=DIM_VECTOR_INT(n);
setval_vector(best_pi_wkl,n,-1);
VECTOR_INT seq_best_l=DIM_VECTOR_INT(n);
setval_vector(seq_best_l,n,-1);
VECTOR_INT copia_best_pi_wkl=DIM_VECTOR_INT(n);
setval_vector(copia_best_pi_wkl,n,-1);
int job_h, min_pos, n_al, h;

while (new_dim>0)
{
   insertion(*seq,&best_pi_wkl,0,omega[0],n);
   dim_pi++;
   seqxfab[0][0]=best_pi_wkl[0];
   jobs_facts_copia[0]=1;
   for (j=1;j<dim_pi;j++)
```

```
{
   f_asignada=A_4(&seq_best_l,seqxfab,jobs_facts_copia,best_pi_wkl[j],f,n,m,pt);
   pasterowImatrix(seqxfab,seq_best_l,f_asignada,n);
   jobs_facts_copia[f_asignada]++;
}
copy_mat_int(seqxfab,*seq_fabs,f,n);
copy_vector(jobs_facts_copia,*jobs_facts,f);
setval_matrix(seqxfab,f,n,-1);
setval_vector(jobs_facts_copia,f,0);
min_CIT=sumCIT(*seq_fabs,*jobs_facts,pt,n,m,f);

for(l=1;l<dim_pi;l++)
{
   insertion(*seq,&pi_wkl,l,omega[0],n);
   seqxfab[0][0]=pi_wkl[0];
   jobs_facts_copia[0]=1;
   for (j=1;j<dim_pi;j++)
   {
      f_asignada=A_4(&seq_best_l,seqxfab,jobs_facts_copia,pi_wkl[j],f,n,m,pt);
      pasterowImatrix(seqxfab,seq_best_l,f_asignada,n);
      jobs_facts_copia[f_asignada]++;

   }
   printf("Matriz seqxfab\n");
   print_matrix(seqxfab,f,n);
   CIT_seq=sumCIT(seqxfab,jobs_facts_copia,pt,n,m,f);
   if (CIT_seq<min_CIT)
   {
      min_CIT=CIT_seq;
      copy_vector(pi_wkl,best_pi_wkl,n);
      copy_vector(jobs_facts_copia,*jobs_facts,f);
      copy_mat_int(seqxfab,*seq_fabs,f,n);
      printf("El programa ha mejorado\n");
      print_matrix(*seq_fabs,f,n);
      printf("minCIT mejorado para el programa anterior es %d\n",min_CIT);

   }
```

```
    setval_matrix(seqxfab,f,n,-1);
    setval_vector(jobs_facts_copia,f,0);
}

//CASUISTICAS POSIBLES DE h
n_al=rand()%2;
min_pos=search_vector(best_pi_wkl,n,omega[0]);
if (min_pos==0)
{
    h=1; //Hay más de dos trabajos en la secuencia
}
else if (min_pos==(dim_pi-1))
{
    h=dim_pi-2;
}
else if( 0<min_pos<(dim_pi-1) && n_al==0)
{
    h=min_pos+1;
}
else if (0<min_pos<(dim_pi-1) && n_al==1)
{
    h=min_pos-1;
}
//printf("h=%d\t min_pos=%d\n",h, min_pos);

if(h!=-1)
{
    job_h=best_pi_wkl[h];
    copy_vector(best_pi_wkl,copia_best_pi_wkl,n);
    extract_vector(copia_best_pi_wkl,n,h);
    //SE SUPONE MIN_CIT LA SECUENCIA SIN EL IMPROVEMENT

    for(l=0;l<dim_pi;l++)
    {
        insertion(copia_best_pi_wkl,&pi_wkl,l,job_h,n);
        seqxfab[0][0]=pi_wkl[0];
        jobs_facts_copia[0]=1;
```

```
        for (j=1;j<dim_pi;j++)
        {
           f_asignada=A_4(&seq_best_l,seqxfab,jobs_facts_copia,pi_wkl[j],f,n,m,pt);
           pasterowImatrix(seqxfab,seq_best_l,f_asignada,n);
           jobs_facts_copia[f_asignada]++;

        }
        CIT_seq=sumCIT(seqxfab,jobs_facts_copia,pt,n,m,f);
        if (CIT_seq<min_CIT)
        {
           min_CIT=CIT_seq;
           copy_vector(pi_wkl,best_pi_wkl,n);
           copy_vector(jobs_facts_copia,*jobs_facts,f);
           copy_mat_int(seqxfab,*seq_fabs,f,n);
           printf("El programa ha mejorado en el improvement\n");
           print_matrix(*seq_fabs,f,n);
           printf("minCIT mejorado para el programa anterior es %d\n",min_CIT);

        }
        setval_matrix(seqxfab,f,n,-1);
        setval_vector(jobs_facts_copia,f,0);
      }


   }
```

//Tras insertar el trabajo en todas las posiciones de posibles de pi_wkl y saber cual es la fabrica que se asigna, se conoce la que genera menor CIT

```
   new_dim=extract_vector(omega,aux,0);
   aux=new_dim;

   printf("Secuencia best_pi_wkl\n");
   print_vector(best_pi_wkl,n);
   copy_vector(best_pi_wkl,*seq,n);

}
```

```
    free_vector(sum_pt);
    free_vector(omega);
    free_vector(jobs_facts_copia);
    free_vector(pi_wkl);
    free_vector(best_pi_wkl);
    free_vector(seq_best_l);
    free_vector(copia_best_pi_wkl);
    free_matrix(seqxfab,f);
}

// METAHEURISTICAS
void ILS (VECTOR_INT *seq, VECTOR_INT *jobs_facts, MAT_INT *seq_fabs, MAT_INT pt, int n, int m,
int f )
{
    VECTOR_INT pi = DIM_VECTOR_INT(n);
    VECTOR_INT jobs_facts_copia = DIM_VECTOR_INT(f);
    VECTOR_INT jobs_facts_copia_1 = DIM_VECTOR_INT(f);
    VECTOR_INT jobs_facts_copia_2 = DIM_VECTOR_INT(f);
    MAT_INT seqxfab = DIM_MAT_INT(f,n);
    MAT_INT seqxfab_1 = DIM_MAT_INT(f,n);
    MAT_INT seqxfab_2 = DIM_MAT_INT(f,n);

    int t, sum_pt=0, best_CIT, ini_CIT;
    register int i,j,perturbation, performs ;

    //PARAMETROS DEL PROBLEMA
    char alfa='DLR_DNEH';
    float x=0.2;
    float beta=0.7;
    int tau=3;
    int omega=20;
    char xi='insertion_operation';

    //MAT INICIAL
    DLR_DNEH(&pi,&jobs_facts_copia,&seqxfab,pt,n,m,f,x);
    //NEH_R1_A4_en(&pi,&jobs_facts_copia,&seqxfab,pt,n,m,f);
    printf("Matriz ini\n");
```

```
print_matrix(seqxfab, f, n);
print_vector(jobs_facts_copia,f);

//Perform RLS to pi
RLS(&seqxfab,pt,&jobs_facts_copia,n,m,f);
copy_mat_int(seqxfab,*seq_fabs,f,n); //Por si nunca llega a actualizarse
copy_vector(jobs_facts_copia,*jobs_facts,f);
ini_CIT=sumCIT(seqxfab,jobs_facts_copia,pt,n,m,f);
best_CIT=ini_CIT;

for (j=0;j<n;j++)
{
   for (i=0;i<m;i++)
   {
      sum_pt+=pt[j][i];
   }
}

t=beta*sum_pt/(10*n*m);

//BUCLE WHILE
VECTOR_INT sumCIT_insertion = DIM_VECTOR_INT(omega);
int best_sumCIT_insertion, random;
copy_mat_int(seqxfab,seqxfab_1,f,n);
copy_vector(jobs_facts_copia,jobs_facts_copia_1,f);
clock_t clock (void);
int time=((double)clock()/CLK_TCK);
double                                                    parada=(0.01)*n*m;
                                                                  //10*n
*m milisegundos

while(time<=parada)
{
   printf("time=%d\n",time);
   for(performs=0;performs<tau;performs++)
   {
      insertion_operator(&seqxfab_1,&jobs_facts_copia_1,pt,m,n,f); //Suponemos que el mejor valor es el
primero generado
```

```
}
printf("Se suppone el mejor programa el primero generado\n");
print_matrix(seqxfab_1,f,n);
copy_mat_int(seqxfab_1,seqxfab_2,f,n);
copy_vector(jobs_facts_copia_1,jobs_facts_copia_2,f);
sumCIT_insertion[0]=sumCIT(seqxfab_1,jobs_facts_copia_1,pt,n,m,f);
printf("El sumCIT del programa anterior es %d\n",sumCIT_insertion[0]);
best_sumCIT_insertion=sumCIT_insertion[0];
for(perturbation=0;perturbation<omega;perturbation++)
{
   for(performs=0;performs<tau;performs++)
   {
      insertion_operator(&seqxfab_1,&jobs_facts_copia_1,pt,m,n,f);

   }
   sumCIT_insertion[perturbation]=sumCIT(seqxfab_1,jobs_facts_copia_1,pt,n,m,f);
   if(sumCIT_insertion[perturbation]<best_sumCIT_insertion)
   {
      copy_mat_int(seqxfab_1,seqxfab_2,f,n);
      copy_vector(jobs_facts_copia_1,jobs_facts_copia_2,f);
      best_sumCIT_insertion=sumCIT_insertion[perturbation];
   }

}
printf("El mejor CIT pertenece al siguiente programa con un objetivo de %d\n",best_sumCIT_insertion);
print_matrix(seqxfab_2,f,n);
print_vector(jobs_facts_copia_2,f);

//PERFORM RLS
RLS(&seqxfab_2,pt,&jobs_facts_copia_2,n,m,f);
printf("Seq_fabs tras RLS\n");
print_matrix(seqxfab_2, f, n);

random=(rand()%100)/100;
if(best_sumCIT_insertion<ini_CIT)
{
   printf("como best_sumCIT_insertion=%d es menor que ini_CIT=%d, se actualiza el mejor
```

```
programa\n",best_sumCIT_insertion, ini_CIT);
            ini_CIT=best_sumCIT_insertion;
            copy_mat_int(seqxfab_2,seqxfab,f,n);
            copy_vector(jobs_facts_copia_2,jobs_facts_copia,f);
        }
        else if (random<exp((double)(ini_CIT-best_sumCIT_insertion)/t))
        {
            printf("se actualiza seqxfab\n");
            ini_CIT=best_sumCIT_insertion;
            copy_mat_int(seqxfab_2,seqxfab,f,n);
            copy_vector(jobs_facts_copia_2,jobs_facts_copia,f);
            //print_matrix(seqxfab,f,n);
        }
        if(best_sumCIT_insertion<best_CIT)
        {
            copy_mat_int(seqxfab_2,*seq_fabs,f,n);
            copy_vector(jobs_facts_copia_2,*jobs_facts,f);
            best_CIT=best_sumCIT_insertion;
        }

    time=((double)clock()/CLK_TCK);
    }

    *seq=matriz_a_vector(*seq_fabs,n,f);

    free_vector(pi);
    free_vector(jobs_facts_copia);
    free_vector(jobs_facts_copia_1);
    free_vector(jobs_facts_copia_2);
    free_vector(sumCIT_insertion);
    free_matrix(seqxfab,f);
    free_matrix(seqxfab_1,f);
    free_matrix(seqxfab_2,f);
}


void RLS (MAT_INT *seqxfab, MAT_INT pt, VECTOR_INT *jobs_facts, int n, int m, int f)
```

```
{
  printf("\n******Calculo de la secuencia RLS******\n");
  VECTOR_INT copia_fab = DIM_VECTOR_INT(n);
  VECTOR_INT fab_reinsertion = DIM_VECTOR_INT(n);
  VECTOR_INT pi = DIM_VECTOR_INT(n);
  VECTOR_INT copia_jobs_facts = DIM_VECTOR_INT(f);
  setval_vector(copia_jobs_facts,f,0);
  MAT_INT copia_seqxfab = DIM_MAT_INT(f,n);
  setval_matrix(copia_seqxfab,f,n,-1);
  int job, pos=0, fab, tam, CIT_ini, encontrado, CIT_reinsertion;
  register int k,j,counter=0, actualizado=0;

  while(counter<n)
  {
    CIT_ini=sumCIT(*seqxfab,*jobs_facts,pt,n,m,f);
    pi=matriz_a_vector(*seqxfab,n,f);
    job=pi[pos]; //el trabajo buscado a reinsertar
    for(k=0; k<f; k++)
    {
      copyrowImatrix(*seqxfab,copia_fab,k,n);
      printf("Secuencia fabrica=%d\n",k);
      print_vector(copia_fab,n);
      encontrado=search_vector(copia_fab,n,job);
      if(encontrado!=-1)
      {
        fab=k;
        k=f; //bandera para salir del bucle
      }
    }

    //EXTRAE TRABAJO
    pos=search_vector(copia_fab,n,job);
    tam=extract_vector(copia_fab,n,pos);
    insert_vector(copia_fab,n,-1,n-1); //para que todas las fabricas tengan la misma dimensión
    copy_vector(*jobs_facts,copia_jobs_facts,f);
    copia_jobs_facts[fab]--;
```

```
//REINSERTA EL TRABAJO EN TODAS LAS POSICIONES PARA QUE MEJOR CIT
copy_mat_int(*seqxfab,copia_seqxfab,f,n);
pasterowImatrix(copia_seqxfab,copia_fab,fab,n);
for(k=0;k<f;k++)
{
  copyrowImatrix(copia_seqxfab,copia_fab,k,n);
  if(jobs_facts[k]<n)
  {
    copia_jobs_facts[k]++;
    for (j=0;j<copia_jobs_facts[k];j++)
    {
      insertion(copia_fab,&fab_reinsertion,j,job,n);
      pasterowImatrix(copia_seqxfab,fab_reinsertion,k,n);
      CIT_reinsertion=sumCIT(copia_seqxfab,copia_jobs_facts, pt,n, m, f);
      if(CIT_reinsertion<CIT_ini)
      {
        printf("Mejora el CIT con el siguiente programa CIT_reinsertion=%d y CIT_ini=%d\n",
CIT_reinsertion,CIT_ini);
        copy_mat_int(copia_seqxfab,*seqxfab,f,n);
        print_matrix(*seqxfab,f,n);
        copy_vector(copia_jobs_facts,*jobs_facts,f);
        print_vector(*jobs_facts,f);
        CIT_ini=CIT_reinsertion;
        counter=0;
        actualizado=1;
      }

    }
    copia_jobs_facts[k]--;
  }
}

if(actualizado==0)
{
  counter+=1;
  //printf("Incrementa el counter a %d\n",counter);
}
```

```
      j=(j+1)/n+1;
      actualizado=0;//Se resetea para la siguiente iteración
   }

   free_vector(copia_fab);
   free_vector(fab_reinsertion);
   free_vector(pi);
   free_vector(copia_jobs_facts);
   free_matrix(copia_seqxfab,f);
}

void insertion_operator(MAT_INT *seqxfab, VECTOR_INT *jobs_facts, MAT_INT pt, int m, int n, int f)
{
   printf("\n******Calculo insertion_operator******\n");
   VECTOR_INT seq = DIM_VECTOR_INT(n);
   VECTOR_INT copia_jobs_facts = DIM_VECTOR_INT(f);
   copy_vector(*jobs_facts,copia_jobs_facts,f);
   int aleatorio, encontrado=-1, pos_reinsertion, f_reinsertion, tam;
   register int k;

   aleatorio=rand()%n; //todos los trabajos están programados
   //printf("Trabajo aleatorio a reinsertar =%d\n",aleatorio);

   //EXTRAE EL TRABAJO PROGRAMADO
   for (k=0;k<f;k++)
   {
      copyrowImatrix(*seqxfab,seq,k,n);
      encontrado=search_vector(seq,n,aleatorio);
      if (encontrado!=-1)
      {
         tam=extract_vector(seq,n,encontrado);
         insert_vector(seq,n,-1,n-1);
         pasterowImatrix(*seqxfab,seq,k,n);
         copia_jobs_facts[k]--;
         k=f; //bandera para salir del bucle
```

```
    }
  }

  //INSERTION ALEATORIA, INCLUYENDO SU MISMA POS
  f_reinsertion=rand()%f; //elegimos fabrica a reinsertat
  int rango=copia_jobs_facts[f_reinsertion];
  if (rango==0)
  {
    pos_reinsertion=0;
  }
  else
  {
      pos_reinsertion=rand()%rango;
  }
  copyrowImatrix(*seqxfab,seq,f_reinsertion,n);
  tam=extract_vector(seq,n,n-1);
  insert_vector(seq,n,aleatorio,pos_reinsertion);
  copia_jobs_facts[f_reinsertion]++;
  pasterowImatrix(*seqxfab,seq,f_reinsertion,n);
  copy_vector(copia_jobs_facts,*jobs_facts,f);

  free_vector(copia_jobs_facts);
  free_vector(seq);
}

void IG2S (VECTOR_INT *seq, VECTOR_INT *jobs_facts, MAT_INT *seq_fabs, MAT_INT pt, int n, int m,
int f )
{
  MAT_INT pi_0 = DIM_MAT_INT(f,n);
  setval_matrix(pi_0,f,n,-1);
  VECTOR_INT jobs_facts_0 = DIM_VECTOR_INT(f);
  setval_vector(jobs_facts_0,f,0);
  VECTOR_INT seq_pi_0 = DIM_VECTOR_INT(n);
  setval_vector(seq_pi_0,f,-1);

  MAT_INT pi = DIM_MAT_INT(f,n);
  setval_matrix(pi,f,n,-1);
  VECTOR_INT jobs_facts_pi = DIM_VECTOR_INT(f);
```

```
setval_vector(jobs_facts_pi,f,0);

MAT_INT pi_destruction = DIM_MAT_INT(f,n);
setval_matrix(pi_destruction,f,n,-1);
VECTOR_INT jobs_facts_destruction = DIM_VECTOR_INT(f);
setval_vector(jobs_facts_destruction,f,0);

VECTOR_INT seq_pi_removed = DIM_VECTOR_INT(n);
setval_vector(seq_pi_removed,n,-1);

MAT_INT pi_reconstruction = DIM_MAT_INT(f,n);
setval_matrix(pi_reconstruction,f,n,-1);
VECTOR_INT jobs_facts_reconstruction = DIM_VECTOR_INT(f);
setval_vector(jobs_facts_reconstruction,f,0);

MAT_INT pi_localsearch = DIM_MAT_INT(f,n);
setval_matrix(pi_localsearch,f,n,-1);
VECTOR_INT jobs_facts_localsearch = DIM_VECTOR_INT(f);
setval_vector(jobs_facts_localsearch,f,0);

VECTOR_INT seq_2stage = DIM_VECTOR_INT(n);
setval_vector(seq_2stage,n,-1);

VECTOR_INT seq_insertion = DIM_VECTOR_INT(n);
setval_vector(seq_insertion,n,-1);

VECTOR_INT programados = DIM_VECTOR_INT(n);
setval_vector(programados,n,-1);

int  sum_pt=0, CPU, d, d_2, CIT_pi, CIT_localsearch, worst_CIT, worst_fab, CIT_fab, job, pos_job, tam, h,
pos_removed=0, best_CIT, aleatorio, count;
double random, temperature, T, rho;
register int i,j,k;

//PARÁMETROS DEL PROBLEMA
rho=0.95;
d=5;
```

```
d_2=6;
T=0.2;

//INICIALIZACIÓN
//NEH2_en(&seq_pi_0,&jobs_facts_0,&pi_0,pt,n,m,f);
NEH_R1_A4_en(&seq_pi_0,&jobs_facts_0,&pi_0,pt,n,m,f);
printf("Inicialización\n");
print_matrix(pi_0,f,n);
print_vector(jobs_facts_0,f);


//Local Search
LS3(&pi, pi_0, &jobs_facts_pi, jobs_facts_0, pt, n, m, f);
printf("Matriz pi tras LS3\n");
print_matrix(pi,f,n);
print_vector(jobs_facts_pi,f);
CIT_pi=sumCIT(pi, jobs_facts_pi, pt, n, m, f);
printf("El CIT_pi=%d\n",CIT_pi);
for(j=0;j<n;j++)
{
   for(i=0;i<m;i++)
   {
      sum_pt=sum_pt+pt[j][i];


   }
}
temperature=T*sum_pt/(10*m*n);

double parada=(0.01)*n*m;
clock_t clock (void);
CPU=((double)clock()/CLK_TCK);
while(CPU<=rho*parada) //
{
   destruction(pi, jobs_facts_pi, pt, &pi_destruction, &jobs_facts_destruction,&seq_pi_removed,d,n,m,f);
   printf("Matriz pi_destruction\n");
   print_matrix(pi_destruction,f,n);
   print_vector(jobs_facts_destruction,f);;
```

```
    printf("Secuencia pi_removed\n");
    print_vector(seq_pi_removed,n);
    reconstruction(pi_destruction, jobs_facts_destruction, &pi_reconstruction, &jobs_facts_reconstruction, seq_pi_removed, pt, n, m, f);
    printf("Matriz pi_reconstruction\n");
    print_matrix(pi_reconstruction,f,n);
    print_vector(jobs_facts_reconstruction,f);
    LS3(&pi_localsearch, pi_reconstruction, &jobs_facts_localsearch, jobs_facts_reconstruction, pt, n, m, f);
    printf("Matriz pi tras LS3 del bucle\n");
    print_matrix(pi_localsearch,f,n);
    print_vector(jobs_facts_localsearch,f);
    CIT_localsearch=sumCIT(pi_localsearch, jobs_facts_localsearch, pt, n, m, f);

    random=rand()%100/100;
    if(CIT_localsearch<CIT_pi)
    {
      CIT_pi=CIT_localsearch;
      copy_mat_int(pi_localsearch,pi,f,n);
      copy_vector(jobs_facts_localsearch,jobs_facts_pi,f);
      printf("Se actualiza matriz pi: CIT tras local search es menor que el inicial\n");
      print_matrix(pi,f,n);
      print_vector(jobs_facts_pi,f);
    }
    else if (random<exp((double)(CIT_pi-CIT_localsearch)/temperature))
    {
      CIT_pi=CIT_localsearch;
      copy_mat_int(pi_localsearch,pi,f,n);
      copy_vector(jobs_facts_localsearch,jobs_facts_pi,f);
      printf("Se actualiza matriz pi: recocido simulado\n");
      print_matrix(pi,f,n);
      print_vector(jobs_facts_pi,f);
    }

    CPU=((double)clock()/CLK_TCK);

  }
```

```
CPU=((double)clock()/CLK_TCK);

while(CPU<parada) //2Stage sigue hasta que ha llegado al tiempo de parada parada
{
   printf("entra en 2STAGE\n");
   copyrowImatrix(pi,seq_2stage,0,n);
   worst_fab=0;
   worst_CIT=CIT(seq_2stage,pt,jobs_facts_pi[worst_fab],m,n);
   printf("Suponemos que la peor fab es worst_fab=%d con worstCIT=%d\n",worst_fab,worst_CIT);
   for(k=0;k<f;k++)
   {
      copyrowImatrix(pi,seq_2stage,k,n);
      CIT_fab=CIT(seq_2stage,pt,jobs_facts_pi[k],m,n);
      if(CIT_fab>worst_CIT)
      {
         worst_fab=k;
         worst_CIT=CIT_fab;
      }
   }
   copyrowImatrix(pi,seq_2stage,worst_fab,n);
   printf("Por tanto voy a trabajar con la siguiente seq\n");
   print_vector(seq_2stage,n);

   //DESTRUCTION
   int extract=0;
   while(extract<d_2)
   {
      if(extract<jobs_facts_pi[worst_fab])
      {
      pos_job=rand()%jobs_facts_pi[worst_fab];
      job=seq_2stage[pos_job];
      tam=extract_vector(seq_pi_removed,n,n-1);
      insert_vector(seq_pi_removed,n,job,pos_removed);
      tam=extract_vector(seq_2stage,n,pos_job);
      insert_vector(seq_2stage,n,-1,n-1);
      pos_removed++;
      jobs_facts_pi[worst_fab]--;
```

```
        }
        extract++;
    }
    printf("vector pi_removed\n");
    print_vector(seq_pi_removed,n);
    printf("Secuencia seq_2stage\n");
    print_vector(seq_2stage,n);
    extract=0;
    pos_removed=0;

    //RECONSTRUCTION
    job=seq_pi_removed[0];
    tam=extract_vector(seq_pi_removed,n,0);
    insert_vector(seq_pi_removed,n,-1,n-1);
    printf("El trabajo a programar de pi_removed job=%d\n",job);
    print_vector(seq_pi_removed,n);

    while(job!=-1)
    {
        pos_job=0;
        insertion(seq_2stage,&seq_insertion,pos_job,job,n);
        jobs_facts_pi[worst_fab]++;
        best_CIT=CIT(seq_insertion,pt,jobs_facts_pi[worst_fab],m,n);
        printf("Se supone el mejor CIT best_CIT=%d al introducirlo en pos_job=%d
,jobs_facts_pi[%d]=%d)\n",best_CIT,pos_job,worst_fab,jobs_facts_pi[worst_fab]);
        print_vector(seq_insertion,n);
        for(j=1;j<jobs_facts_pi[worst_fab];j++)
        {
            insertion(seq_2stage,&seq_insertion,j,job,n);
            CIT_fab=CIT(seq_insertion,pt,jobs_facts_pi[worst_fab],m,n);
            if(CIT_fab<best_CIT)
            {
                pos_job=j;
                printf("Se mejora el CIT con pos_job=%d\n",pos_job);
            }

        }
```

```
insertion(seq_2stage,&seq_insertion,pos_job,job,n);
copy_vector(seq_insertion,seq_2stage,n);
printf("Al insertar el trabajo job=%d en pos_job=%d queda tal que\n",job,pos_job);
print_vector(seq_2stage,n);

aleatorio=rand()%2;
if (pos_job==0 && jobs_facts_pi[worst_fab]>1)
{
  h=1; //Hay más de dos trabajos en la secuencia
}
else if (pos_job==(jobs_facts_pi[worst_fab]-1))
{
  h=jobs_facts_pi[worst_fab]-2;
}
else if( 0<pos_job<(jobs_facts_pi[worst_fab]-1) && aleatorio==0)
{
  h=pos_job+1;
}
else if (0<pos_job<(jobs_facts_pi[worst_fab]-1) && aleatorio==1)
{
  h=pos_job-1;
}


printf("Ahora vamos a tomar el trabajo job=%d de la pos h=%d para reprogramarlo\n",seq_2stage[h],h);
if(h!=-1)
{
job=seq_2stage[h];
pos_job=h;
for(j=0;j<jobs_facts_pi[worst_fab];j++)
{
  tam=extract_vector(seq_insertion,n,h);
  insert_vector(seq_insertion,n,job,j);
  CIT_fab=CIT(seq_insertion,pt,jobs_facts_pi[worst_fab],m,n);
  if(CIT_fab<best_CIT)
  {
    pos_job=j;
    best_CIT=CIT_fab;
```

```
        }
        tam=extract_vector(seq_insertion,n,j);
        insert_vector(seq_insertion,n,job,h);
    }
    tam=extract_vector(seq_2stage,n,h);
    insert_vector(seq_2stage,n,job,pos_job);
    }
    job=seq_pi_removed[0];
    tam=extract_vector(seq_pi_removed,n,0);
    insert_vector(seq_pi_removed,n,-1,n-1);
}

//LOCAL SEARCH
best_CIT=CIT(seq_2stage,pt,jobs_facts_pi[worst_fab],m,n);
aleatorio=rand()%jobs_facts_pi[worst_fab];
job=seq_2stage[aleatorio];
h=-1;
while(count<jobs_facts_pi[worst_fab])
{
    h=search_vector(programados,n,job);
    if(h==-1)
    {
        programados[count]=job;
        copy_vector(seq_2stage,seq_insertion,n);
        tam=extract_vector(seq_insertion,n,aleatorio);
        for(j=0;j<jobs_facts_pi[worst_fab];j++)
        {
            insert_vector(seq_insertion,n,job,j);
            CIT_fab=CIT(seq_insertion,pt,jobs_facts_pi[worst_fab],m,n);
            if(CIT_fab<best_CIT)    //Acceptance criteria
            {
                best_CIT=CIT_fab;
                copy_vector(seq_insertion,seq_2stage,n);
            }
            extract_vector(seq_insertion,n,j);
        }
        count++;
```

```
        }
        aleatorio=rand()%jobs_facts_pi[worst_fab];
        job=seq_2stage[aleatorio];

      }
      CPU=((double)clock()/CLK_TCK);
      copyrowImatrix(pi,seq_2stage,worst_fab,n);
    }

    copy_mat_int(pi,*seq_fabs,f,n);
    copy_vector(jobs_facts_pi,*jobs_facts,f);
    *seq=matriz_a_vector(pi,n,f);

    free_matrix(pi,f);
    free_vector(jobs_facts_pi);
    free_matrix(pi_0,f);
    free_vector(jobs_facts_0);
    free_vector(seq_pi_0);
    free_matrix(pi_destruction,f);
    free_vector(jobs_facts_destruction);
    free_vector(seq_pi_removed);
    free_matrix(pi_reconstruction,f);
    free_vector(jobs_facts_reconstruction);
    free_matrix(pi_localsearch,f);
    free_vector(jobs_facts_localsearch);
    free_vector(seq_2stage);
    free_vector(seq_insertion);
    free_vector(programados);
}

void LS3(MAT_INT *pi, MAT_INT pi_0, VECTOR_INT *jobs_facts_pi, VECTOR_INT jobs_facts_0,
MAT_INT pt, int n, int m, int f)
{
    printf("\n******Local Search LS3******\n");

    VECTOR_INT programados = DIM_VECTOR_INT(n);
    setval_vector(programados,n,-1);
    VECTOR_INT copia_seq_fab = DIM_VECTOR_INT(n);
```

```
setval_vector(copia_seq_fab,n,-1);
VECTOR_INT seq_insertion = DIM_VECTOR_INT(n);
setval_vector(seq_insertion,n,-1);

int best_fab, best_pos, best_sumCIT, sumCIT_pi, worst_fab, worst_CIT, pos_job, job, repeated=-1, total_CIT, CIT_fab ;
register int k, j, count=0, new_job=0;

copy_mat_int(pi_0,*pi,f,n);
printf("Matriz pi\n");
print_matrix(*pi,f,n);
copy_vector(jobs_facts_0,*jobs_facts_pi,f);
print_vector(*jobs_facts_pi,f);
sumCIT_pi=sumCIT(pi_0,jobs_facts_0,pt,n,m,f);
printf("El sumCIT del programa anterior es =%d\n",sumCIT_pi);
copyrowImatrix(pi_0,copia_seq_fab,0,n);
worst_CIT=CIT(copia_seq_fab,pt,jobs_facts_0[0],m,n);
printf("El peor CIT es worst_CIT=%d\n",worst_CIT);
worst_fab=0; //inicializamos el peor valor con la primera fabrica

for(k=1;k<f;k++)
{
   copyrowImatrix(pi_0,copia_seq_fab,k,n);
   CIT_fab=CIT(copia_seq_fab,pt,jobs_facts_0[k],m,n);
   if(CIT_fab>worst_CIT)
   {
      worst_CIT=CIT_fab;
      worst_fab=k;
   }

}
printf("La fábrica con mayor CIT es worst_fab=%d\n",worst_fab);
while(count<jobs_facts_0[worst_fab])
{
   pos_job=rand()%jobs_facts_0[worst_fab];
   job=pi_0[worst_fab][pos_job];
   printf("El trabajo aleatorio extraido de worst_fab es job=%d=pi[%d][%d]\n",job,worst_fab,pos_job);
```

```
repeated=search_vector(programados,n,job); //hay que actualizar el valor repeated y el vector programados al cambiar de fábrica
if(repeated==-1) //no se ha programado hasta el momento
{
  copyrowImatrix(pi_0,copia_seq_fab,worst_fab,n);
  int tam=extract_vector(copia_seq_fab,n,pos_job);
  insert_vector(copia_seq_fab,n,-1,n-1);
  pasterowImatrix(pi_0,copia_seq_fab,worst_fab,n);
  jobs_facts_0[worst_fab]--;
  best_fab=0; //se supone la mejor fabrica
  best_pos=0; //se supone la mejor posición de la fabrica
  copyrowImatrix(pi_0, copia_seq_fab,best_fab,n);
  copy_vector(copia_seq_fab,seq_insertion,n);
  insertion(copia_seq_fab,&seq_insertion,best_pos,job,n);
  printf("Se supone que best_fab=%d y best_pos=%d\t",best_fab,best_pos);
  jobs_facts_0[best_fab]++;
  pasterowImatrix(pi_0,seq_insertion,best_fab,n);
  best_sumCIT=sumCIT(pi_0,jobs_facts_0,pt,n,m,f);
  print_matrix(pi_0,f,n);
  print_vector(jobs_facts_0,f);
  printf("...y por tanto best_sumCIT=%d\n",best_sumCIT);
  pasterowImatrix(pi_0,copia_seq_fab,best_fab,n);
  for(j=1;j<jobs_facts_0[best_fab];j++)
  {
    insertion(copia_seq_fab,&seq_insertion,j,job,n);
    pasterowImatrix(pi_0,seq_insertion,best_fab,n);
    total_CIT=sumCIT(pi_0,jobs_facts_0,pt,n,m,f);
    pasterowImatrix(pi_0,copia_seq_fab,best_fab,n);
    if(total_CIT<best_sumCIT)
    {
      best_sumCIT=total_CIT;
      best_pos=j;
      printf("Se actualiza a best_pos=%d\n",best_pos);
    }
  }
  jobs_facts_0[best_fab]--;
  for(k=1;k<f;k++)
  {
```

```
            jobs_facts_0[k]++;
            copyrowImatrix(pi_0,copia_seq_fab,k,n);
            copy_vector(copia_seq_fab,seq_insertion,n);
            for(j=0;j<jobs_facts_0[k];j++)
            {
               insertion(copia_seq_fab,&seq_insertion,j,job,n);
               pasterowImatrix(pi_0,seq_insertion,k,n);
               total_CIT=sumCIT(pi_0,jobs_facts_0,pt,n,m,f);
               pasterowImatrix(pi_0,copia_seq_fab,k,n);
               if(total_CIT<best_sumCIT)
               {
                  best_fab=k;
                  best_sumCIT=total_CIT;
                  best_pos=j;
               }

            }
            jobs_facts_0[k]--;
         }
         if(best_sumCIT<sumCIT_pi)
         {
            copyrowImatrix(pi_0,copia_seq_fab,best_fab,n);
            tam=extract_vector(copia_seq_fab,n,n-1);
            insert_vector(copia_seq_fab,n,job,best_pos);
            pasterowImatrix(pi_0,copia_seq_fab,best_fab,n);
            jobs_facts_0[best_fab]++;
            printf("Como el CIT ha mejorado, se ha introducido el trabjo job=%d en la fab=%d en la
pos=%d\n",job,best_fab,best_pos);
            print_matrix(pi_0,f,n);
            print_vector(jobs_facts_0,f);
            copy_mat_int(pi_0,*pi,f,n);
            copy_vector(jobs_facts_0,*jobs_facts_pi,f);
            sumCIT_pi=best_sumCIT;

            count=0;
            setval_vector(programados,n,-1);
            new_job=0;
```

```
            copyrowImatrix(pi_0,copia_seq_fab,0,n);
            worst_CIT=CIT(copia_seq_fab,pt,jobs_facts_0[0],m,n);
            worst_fab=0; //inicializamos el peor valor con la primera fabrica

            for(k=1;k<f;k++)
            {
               copyrowImatrix(pi_0,copia_seq_fab,k,n);
               CIT_fab=CIT(copia_seq_fab,pt,jobs_facts_0[k],m,n);
               if(CIT_fab>worst_CIT)
               {
                  worst_CIT=CIT_fab;
                  worst_fab=k;
               }

            }
            printf("Ahora la fabrica con peor CIT es worst_fab=%d",worst_fab);
         }
         else
         {
            printf("El CIT no ha mejorado, se introduce de nuevo en pos_job=%d de
worst_fab=%d\n",pos_job,worst_fab);
            copyrowImatrix(pi_0,copia_seq_fab,worst_fab,n);
            tam=extract_vector(copia_seq_fab,n,n-1);
            insert_vector(copia_seq_fab,n,job,pos_job);
            pasterowImatrix(pi_0,copia_seq_fab,worst_fab,n);
            jobs_facts_0[worst_fab]++;
            print_matrix(pi_0,f,n);
            print_vector(jobs_facts_0,f);
            copy_mat_int(pi_0,*pi,f,n);
            copy_vector(jobs_facts_0,*jobs_facts_pi,f);

            count++;
            insert_vector(programados,n,job,new_job);
            tam=extract_vector(programados,n,n-1),
            new_job++;
         }
      }
      repeated=-1;
```

```
    }

    free_vector(programados);
    free_vector(copia_seq_fab);
    free_vector(seq_insertion);
}

void destruction (MAT_INT pi, VECTOR_INT jobs_facts_pi, MAT_INT pt, MAT_INT *pi_destruction,
VECTOR_INT *jobs_facts_destruction, VECTOR_INT *pi_removed, int d, int n, int m, int f)
{
    printf("\n******Destruction******\n");
    VECTOR_INT copia_seq_fab = DIM_VECTOR_INT(n);
    setval_vector(copia_seq_fab,n,-1);
    MAT_INT pi_copia =DIM_MAT_INT(f,n);
    copy_mat_int(pi,pi_copia,f,n);
    VECTOR_INT jobs_facts_pi_copia = DIM_VECTOR_INT(n);
    copy_vector(jobs_facts_pi,jobs_facts_pi_copia,f);
    int worst_fab, worst_CIT, CIT_fab, pos_job, job, pos_removed=0,fab, extract=0, tam;
    register int k, j;

    copyrowImatrix(pi_copia,copia_seq_fab,0,n);
    worst_CIT=CIT(copia_seq_fab,pt,jobs_facts_pi[0],m,n);
    worst_fab=0;
    printf("Suponemos que el peor CIT lo genera la fabrica=%d con un CIT=%d\n",worst_fab,worst_CIT);
    for(k=1;k<f;k++)
    {
      copyrowImatrix(pi_copia,copia_seq_fab,k,n);
      CIT_fab=CIT(copia_seq_fab,pt,jobs_facts_pi_copia[k],m,n);
      if(CIT_fab>worst_CIT)
      {
        worst_CIT=CIT_fab;
        worst_fab=k;
        printf("Se actualiza la fábrica worst_fab=%d\n",worst_fab);
      }
    }

    while(extract<(d/2))
```

```
  {
    if(jobs_facts_pi[worst_fab]!=0)
    {
      copyrowImatrix(pi_copia,copia_seq_fab,worst_fab,n);
      pos_job=rand()%jobs_facts_pi_copia[worst_fab];
      job=copia_seq_fab[pos_job];
      insert_vector(*pi_removed,n,job,pos_removed);
      tam=extract_vector(*pi_removed,n,n-1);
      tam=extract_vector(copia_seq_fab,n,pos_job);
      insert_vector(copia_seq_fab,n,-1,n-1);
      pasterowImatrix(pi_copia,copia_seq_fab,worst_fab,n);
      jobs_facts_pi_copia[worst_fab]--;
      pos_removed++;
      extract++;
    }
    else
    {
      extract=d/2;
    }
  }

  while(extract<=d)
  {
    fab=rand()%f;
    if(fab!=worst_fab)
    {
      if(jobs_facts_pi_copia[fab]!=0)
      {

                                                                    copyr
owImatrix(pi_copia,copia_seq_fab,fab,n);
        pos_job=rand()%jobs_facts_pi_copia[fab];
        job=copia_seq_fab[pos_job];
        insert_vector(*pi_removed,n,job,pos_removed);
        tam=extract_vector(*pi_removed,n,n-1);
        tam=extract_vector(copia_seq_fab,n,pos_job);
        insert_vector(copia_seq_fab,n,-1,n-1);
        pasterowImatrix(pi_copia,copia_seq_fab,fab,n);
```

```
            jobs_facts_pi_copia[fab]--;
            pos_removed++;
          }

      }
      extract++;

   }

   copy_mat_int(pi_copia,*pi_destruction,f,n);
   copy_vector(jobs_facts_pi_copia,*jobs_facts_destruction,f);

   free_vector(copia_seq_fab);
}

void reconstruction (MAT_INT pi_destruction,VECTOR_INT jobs_facts_destruction, MAT_INT *pi_reconstruction, VECTOR_INT *jobs_facts_reconstruction, VECTOR_INT pi_removed, MAT_INT pt, int n, int m, int f)
{
   printf("\n******Reconstruction******\n");
   VECTOR_INT copia_seq_fab = DIM_VECTOR_INT(n);
   VECTOR_INT seq_insertion = DIM_VECTOR_INT(n);

   int job_to_insert, pos_job, CIT_fab, best_CIT, best_fab, best_pos, h, random, tam;
   register int j,k;

   job_to_insert=pi_removed[0];
   tam=extract_vector(pi_removed,n,0);
   insert_vector(pi_removed,n,-1,n-1);

   while(job_to_insert!=-1)
   {
      best_fab=0;
      best_pos=0;
      copyrowImatrix(pi_destruction,copia_seq_fab,best_fab,n);
      copy_vector(copia_seq_fab,seq_insertion,n);
      insertion(copia_seq_fab,&seq_insertion,best_pos,job_to_insert,n);
```

```
jobs_facts_destruction[best_fab]++;
best_CIT=CIT(seq_insertion,pt,jobs_facts_destruction[best_fab],m,n);
for(j=1;j<jobs_facts_destruction[best_fab];j++)
{
  insertion(copia_seq_fab,&seq_insertion,j,job_to_insert,n);
  CIT_fab=CIT(seq_insertion,pt,jobs_facts_destruction[best_fab],m,n);
  if(CIT_fab<best_CIT)
  {
    best_CIT=CIT_fab;
    best_pos=j;
    printf("Se ha actualizado best_CIT=%d y best_pos=%d\n",best_CIT,best_pos);
  }

}
jobs_facts_destruction[best_fab]--;

for(k=1;k<f;k++)
{
  copyrowImatrix(pi_destruction,copia_seq_fab,k,n);
  copy_vector(copia_seq_fab,seq_insertion,n);
  jobs_facts_destruction[k]++;
  for(j=0;j<jobs_facts_destruction[k];j++)
  {
    insertion(copia_seq_fab,&seq_insertion,j,job_to_insert,n);
    CIT_fab=CIT(seq_insertion,pt,jobs_facts_destruction[k],m,n);
    if(CIT_fab<best_CIT)
    {
      best_CIT=CIT_fab;
      best_pos=j;
      best_fab=k;
    }
  }
  jobs_facts_destruction[k]--;
}
printf("Se va introducir el trabajo=%d\t en best_pos=%d de
best_fab=%d\n",job_to_insert,best_pos,best_fab);
copyrowImatrix(pi_destruction,copia_seq_fab,best_fab,n);
copy_vector(copia_seq_fab,seq_insertion,n);
```

```
insertion(copia_seq_fab,&seq_insertion,best_pos,job_to_insert,n);
pasterowImatrix(pi_destruction,seq_insertion,best_fab,n);
jobs_facts_destruction[best_fab]++;
printf("Matriz con el trabajo job_to_insert=%d\n",job_to_insert);
print_matrix(pi_destruction,f,n);
print_vector(jobs_facts_destruction,f);

random=rand()%2;
if (best_pos==0 && jobs_facts_destruction[best_fab]>1)
{
   h=1; //Hay más de dos trabajos en la secuencia
}
else if (best_pos==(jobs_facts_destruction[best_fab]-1))
{
   h=jobs_facts_destruction[best_fab]-2;
}
else if( 0<best_pos<(jobs_facts_destruction[best_fab]-1) && random==0)
{
   h=best_pos+1;
}
else if (0<best_pos<(jobs_facts_destruction[best_fab]-1) && random==1)
{
   h=best_pos-1;
}
if(h!=-1)
{
job_to_insert=seq_insertion[h];
pos_job=h;
for(j=0;j<jobs_facts_destruction[best_fab];j++)
{
   tam=extract_vector(seq_insertion,n,h);
   insert_vector(seq_insertion,n,job_to_insert,j);
   CIT_fab=CIT(seq_insertion,pt,jobs_facts_destruction[best_fab],m,n);
   if(CIT_fab<best_CIT)
   {
      pos_job=j;
   }
```

```
            tam=extract_vector(seq_insertion,n,j);
            insert_vector(seq_insertion,n,job_to_insert,h);

        }
        tam=extract_vector(seq_insertion,n,h);
        insert_vector(seq_insertion,n,job_to_insert,pos_job);
        }
        job_to_insert=pi_removed[0];
        tam=extract_vector(pi_removed,n,0);
        insert_vector(pi_removed,n,-1,n-1);
    }
    copy_mat_int(pi_destruction,*pi_reconstruction,f,n);
    copy_vector(jobs_facts_destruction,*jobs_facts_reconstruction,f);

    free_vector(copia_seq_fab);
    free_vector(seq_insertion);
}
```