Trabajo Fin de Grado
Ingeniería de Organización Industrial

Optimización de la Producción en Entornos Multifábrica: Reducción de Costes Energéticos y Aplicación de la Restricción No Idle

Autor: María del Carmen Conejo Hidalgo

Tutor: Paz Pérez González

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

Sevilla, 2024

Trabajo Fin de Grado
Ingeniería de Organización Industrial

Optimización de la Producción en Entornos Multifábrica: Reducción de Costes Energéticos y Aplicación de la Restricción No Idle

Autor:
María del Carmen Conejo Hidalgo

Tutor:
Paz Pérez González

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

Sevilla, 2024

Trabajo Fin de Grado: Optimización de la Producción en Entornos Multifábrica: Reducción de Costes Energéticos y Aplicación de la Restricción No Idle

Autor: María del Carmen Conejo Hidalgo
Tutor: Paz Pérez González

El tribunal nombrado para juzgar el trabajo arriba indicado, compuesto por los siguientes profesores:

Presidente:

Vocal/es:

Secretario:

acuerdan otorgarle la calificación de:

El Secretario del Tribunal

Fecha:

Agradecimientos

A mi familia, por confiar siempre en mí

y acompañarme en cada paso de mi vida,

María

Resumen

Las cadenas de suministro han evolucionado hacia estructuras más complejas en las que las empresas distribuyen la producción en múltiples ubicaciones. Esto puede conllevar grandes ventajas como optar por una mayor diversidad de recursos, lo que permite aprovechar distintas capacidades y conocimientos para obtener un producto de mayor calidad. A su vez, trabajar en un entorno multifábrica puede reducir costes, al compartir recursos, instalaciones, equipos y mano de obra, es posible reducir los costes de producción, pues no cuesta lo mismo producir en diferentes ubicaciones. Compartir riesgos de producción es otro privilegio de estos entornos. Al distribuir la producción se reduce el riesgo de depender de una sola planta, un único conjunto de máquinas o de determinados proveedores. Esto proporciona una mayor capacidad para enfrentar contratiempos, como interrupciones en la cadena de suministro por avería de alguna máquina, por ejemplo. Este tipo de producción ofrece un mayor alcance geográfico al llegar a diversos mercados, existe la posibilidad de producir en proximidad a los clientes. Esto se traduce en una reducción de los costes de transporte y en la optimización de los tiempos de entrega.

Por otro lado, el sector Industrial se ha convertido en un principal contribuyente a la contaminación ambiental y la utilización de recursos. La sostenibilidad de la cadena de suministro es uno de los mayores desafíos de la actualidad. Para abordar este problema, se ha optado por reducir el consumo energético tanto en logística como en producción y consumo, así como por incentivar el uso de energías renovables. En la búsqueda de esta eficiencia energética u optimización de los recursos, se propone la restricción del tiempo de inactividad entre trabajos que se procesan en una máquina para minimizar el tiempo que estas están en funcionamiento. Además, en sectores industriales modernos como la fabricación de acero, el procesamiento de fibra de vidrio u otros procesos químicos, las características técnicas de estos procesos requieren operaciones sin interrupciones. De igual forma ocurre cuando se trabaja con maquinaria muy costosa o cuando ciertas máquinas no pueden ser encendidas y apagadas debido a restricciones tecnológicas. Por ello, la programación de la producción sin contemplar tiempos de inactividad ha comenzado a ser un tema bastante interesante.

En este proyecto se analizará el problema de programación de la producción en un entorno multifábrica, respetando la limitación de tiempos de inactividad o tiempos ociosos entre trabajos y atendiendo al objetivo de minimizar el coste energético de tener en funcionamiento las máquinas de cada planta.

Abstract

Supply chains have evolved towards more complex structures where companies distribute production across multiple locations. This can bring significant advantages such as opting for a greater diversity of resources, allowing the leveraging of different capabilities and knowledge to achieve a higher quality product. Furthermore, working in a multi-factory environment can reduce costs by sharing resources, facilities, equipment, and labor, thereby reducing production costs, as producing in different locations does not incur the same costs. Sharing production risks is another privilege of these environments. By distributing production, the risk of depending on a single plant, a single set of machines, or specific suppliers is reduced. This provides greater capacity to deal with setbacks, such as supply chain interruptions due to machine breakdowns, for instance. This type of production offers broader geographical reach by accessing diverse markets, with the possibility of producing close to customers. This translates into reduced transportation costs and optimized delivery times.

On the other hand, the industrial sector has become a major contributor to environmental pollution and resource utilization. Today, sustainability of the supply chain is one of the greatest challenges. To address this issue, efforts have been made to reduce energy consumption in logistics, production, and consumption, as well as to promote the use of renewable energies. In pursuit of this energy efficiency and resource optimization, it is proposed to restrict idle times between jobs processed on a machine. Additionally, in modern industrial sectors such as steel manufacturing, fiberglass processing, or other chemical processes, the technical characteristics of these processes require uninterrupted operations. Similarly, this occurs when working with expensive machinery or when certain machines cannot be easily started and stopped due to technological constraints. Therefore, production scheduling without considering idle times has become a highly interesting topic.

This project will analyze the production scheduling problem in a multi-factory environment, respecting idle times limitations, specifically, zero idle times, and aiming to minimize the energy cost of operating machines in each plant.

Índice

Notación

m	Número de máquinas
n	Número de trabajos
f	Número de fábricas
i	Índice para máquinas $1 < i < m$
j	Índice para trabajos $1 < j < n$
k	Índice para posición de los trabajos $2 < k < n$
CIT	Core Idle Time
SPT	Shortest Processing Time
LPT	Longest Processing Time
p_{ij}	Tiempo de proceso del trabajo j en la máquina i
DF	Distributed Flowshop
$w_f Cmáx^f$	Makespan por fábrica multiplicado por w_f
w_f	Coste energético por fábrica
$ARPD$	Average Relative Percentage Deviation

1 Introducción

Las cadenas de suministro han evolucionado hacia estructuras más complejas en las que las empresas distribuyen la producción en múltiples ubicaciones. Esto puede conllevar grandes ventajas como optar por una mayor diversidad de recursos, lo que permite aprovechar distintas capacidades y conocimientos para obtener un producto de mayor calidad. A su vez, trabajar en un entorno multifábrica puede reducir costes, al compartir recursos, instalaciones, equipos y mano de obra, es posible reducir los costes de producción, pues no cuesta lo mismo producir en diferentes ubicaciones. Compartir riesgos de producción es otro privilegio de estos entornos. Al distribuir la producción se reduce el riesgo de depender de una sola planta, un único conjunto de máquinas o de determinados proveedores. Esto proporciona una mayor capacidad para enfrentar contratiempos, como interrupciones en la cadena de suministro por avería de alguna máquina, por ejemplo. Este tipo de producción ofrece un mayor alcance geográfico al llegar a diversos mercados, existe la posibilidad de producir en proximidad a los clientes. Esto se traduce en una reducción de los costes de transporte y en la optimización de los tiempos de entrega.

Por otro lado, el sector Industrial se ha convertido en un principal contribuyente a la contaminación ambiental y la utilización de recursos. La sostenibilidad de la cadena de suministro es uno de los mayores desafíos de la actualidad (Wu & Lin, 2021). Una cadena de suministro sostenible se refiere a la gestión de los procesos de producción y distribución de bienes y servicios de manera que se minimicen los impactos negativos en el medio ambiente, se promueva la responsabilidad social y se asegure la viabilidad financiera a largo plazo. Esto implica considerar no solo los beneficios económicos, sino también los impactos sociales y ambientales a lo largo de toda la cadena de suministro, desde la adquisición de materias primas, la fabricación y hasta la entrega del producto final al consumidor y el manejo de residuos.Para abordar este problema, se ha optado por reducir el consumo energético tanto en logística como en producción y consumo, así como por incentivar el uso de energías renovables. En la búsqueda de esta eficiencia energética u optimización de los recursos, se propone la restricción del tiempo de inactividad entre trabajos que se procesan en una máquina para minimizar el tiempo que estas están en funcionamiento. Además, en sectores industriales modernos como la fabricación de acero, el procesamiento de fibra de vidrio u otros procesos químicos, las características técnicas de estos procesos requieren operaciones sin interrupciones. De igual forma ocurre cuando se trabaja con maquinaria muy costosa o cuando ciertas máquinas no pueden ser encendidas y apagadas debido a restricciones tecnológicas, por ejemplo (Ruiz et al., 2009). Por ello, la programación de la producción sin contemplar tiempos de inactividad ha comenzado a ser un tema bastante crítico en la industria.

En este proyecto se han desarrollado una serie de métodos y análisis para abordar el problema de programación de la producción en un entorno multifábrica, respetando la limitación de tiempos de inactividad y atendiendo al objetivo de minimizar el coste energético de tener en funcionamiento las máquinas de cada planta. Este documento ha sido elaborado utilizando el lenguaje de tipografía LaTeX, que permite crear documentos de alta calidad tipográfica de forma eficiente y estructurada. Para la implementación de los métodos de programación y parte del análisis de los resultados se ha utilizado Python, lenguaje de programación versátil que ha permitido la programación eficiente de los algoritmos propuestos, gracias a la utilización de varias librerías como "pandas" o "matplotlib", por ejemplo, para la obtención de gráficas. Además, el tratamiento de los datos se ha llevado a cabo en Excel, herramienta que ha permitido organizar y analizar todos los resultados obtenidos. Por otro lado, todas las figuras incluidas en este documento han sido elaboradas por el autor, salvo que se indique expresamente la fuente de origen.

1.1 Objetivos del Trabajo Fin de Grado

El objetivo principal de este proyecto es el estudio y análisis de un problema de programación de la producción en un entorno multifábrica DPFSP (Distributed Permutation Flowshop). Se trata de una extensión del entorno monofábrica PFSP (Permutation Flowshop), en el cual se consideran varios centros de producción distribuidos en distintas ubicaciones. Este problema nace de la necesidad de lograr operaciones de abastecimiento, producción, y distribución más eficientes junto con la introducción de la Industria 4.0 en la cadena de suministro.

En la literatura, se encuentran varios estudios donde la fabricación distribuida permite a las empresas lograr una mayor calidad del producto, menores costes de producción y riesgos de gestión. Sin embargo, dichos estudios se centran más en los aspectos económicos en lugar de en la programación de capacidad finita distribuida. Comparado con los problemas tradicionales de programación de la producción en una sola fábrica, la programación en entornos multifábrica es más compleja. En los problemas de una sola fábrica, el desafío es programar los trabajos en un conjunto de máquinas, mientras que en el problema distribuido surge una decisión adicional importante: la asignación de trabajos a fábricas adecuadas. Por lo tanto, uno de los objetivos de este estudio se basa en tomar dos decisiones: la asignación de trabajos a las fábricas y la programación de trabajos en cada fábrica. Obviamente, ambos problemas están estrechamente relacionados y no pueden resolverse secuencialmente si se desea un alto rendimiento (Naderi & Ruiz, 2016).

Otro de los grandes problemas de la industria actual es, como se ha comentado anteriormente, la eficiencia energética o el uso eficiente de sus recursos. Cuando se trabaja con maquinaria muy costosa o que interviene en procesos industriales críticos como equipos en plantas de procesamiento químico, refinerías y fábricas de producción continua, como plantas de acero o papel, se opta por operar sin interrupciones para maximizar la eficiencia y evitar los gastos asociados con los apagados y reinicios y posibles averías derivadas de ello. Por ello, se estudia la programación de la producción sin interrupciones entre trabajos con la finalidad de encontrar un programa que minimice el gasto energético de producir en un entorno multifábrica.

Adicionalmente, el objetivo de este trabajo es lograr lo siguiente:

- Estudio del arte: investigación bibliográfica del entorno, restricción y objetivo del problema, así como de los métodos para su resolución de la forma más eficiente.
- Resolución del problema mediante la implementación métodos heurísticos y metaheurísticas a partir de los existentes en la literatura.
- Análisis de los resultados obtenidos mediante Excel y Python con el fin de comparar los métodos y elegir aquel que proporcione mejores datos.
- Desarrollar y documentar pseudocódigos para los algoritmos clave del problema, con el fin de proporcionar una representación clara de sus estructuras y lógicas de funcionamiento.
- Profundizar en el lenguaje de programación Python para el desarrollo y análisis de los métodos y en el lenguaje de texto LaTeXpara la creación de este documento.

Tras exponer los objetivos de este proyecto, se realiza una breve explicación de la estructura del documento.

1.2 Estructura del documento

El presente Trabajo Fin de Grado se compone de seis Capítulos y un Anexo. En el primer capítulo, se presenta una breve introducción al problema que será analizado, así como una exposición de los objetivos correspondientes. El capítulo dos contiene una introducción a la programación de la producción en la cual se exponen conceptos básicos necesarios para entender el problema. En el tercer apartado, se define el problema de estudio, diferenciando cada uno de los parámetros en los que se clasifican los problemas de Programación de la Producción. Por otro lado, en el cuarto capítulo, se describen todos los métodos heurísticos y metaheurísticos que serán programados, así como ejemplos ilustrativos de estos. En el quinto apartado, se presenta el análisis de todos los métodos estudiados previamente en el capítulo anterior. Por último, el capítulo seis comprende las conclusiones extraídas del proyecto y en el Anexo se refleja el código del modelo y todos los métodos programados para su optimización.

2 Introducción a la Programación y Control de la Producción

La programación de la producción se refiere a la toma de decisiones a nivel operativo (o corto plazo). El objetivo de esta es la asignación de ciertos trabajos a los recursos (máquinas) disponibles de una empresa. El resultado es un programa en el que se detalla qué trabajo debe procesarse en cada recurso y cuándo, es decir, qué maquina procesa cada trabajo y los instantes de inicio y fin de cada uno de ellos (Framinan et al., 2014). Para obtener un programa se debe proporcionar una secuencia que indique el orden en el que cada trabajo empieza a procesarse en cada máquina. A su vez, el programa debe ser admisible (feasible schedule), es decir, tiene que cumplir con todas las restricciones y características del proceso productivo y semiactivo (semi-active schedule), no es posible adelantar ninguna operación sin cambiar el orden en que alguna máquina procesa los trabajos. Por otro lado, el Control de la Producción engloba, en primer lugar, el conjunto de mecanismos para monitorizar las desviaciones o cambios que aparecen en el programa de producción y, en segundo lugar, todas aquellas acciones correctoras que deben ejecutarse (Perez Gonzalez et al., 2021).

En este caso, nos centraremos en la Programación de la Producción y para ello es necesario conocer un conjunto de conceptos básicos sobre esta disciplina:

- Fábrica: instalación industrial donde se llevan a cabo procesos de producción en serie. Al conjunto de fábricas se le denota como $F = \{1,\ldots,f\}$.
  - Índice para fábricas: $f \in F$.
- Máquina: recurso productivo con capacidad para realizar operaciones de transformación y/o transporte de material. Al conjunto de máquinas se le denota como $M = \{1,\ldots,m\}$.
  - Índice para máquinas: $i \in M$.
- Trabajo: producto que es objeto de una operación en algunas de las máquinas disponibles de la fábrica. Al conjunto de trabajos se le denota como $N = \{1,\ldots,n\}$.
  - Índice para los trabajos: $j,k \in N$.
- Tiempo de proceso, p_{ij}, es el tiempo que la máquina $i \in M$ tarda en procesar el trabajo $j \in N$. La notación p_j se utiliza cuando es independiente de la máquina.
- Fecha de llegada, r_j, instante de disponibilidad a partir del cual el trabajo $j \in N$ puede empezar a ser procesado.
- Fecha de entrega, d_j, instante de tiempo en el que el trabajo $j \in N$ debe de estar terminado, es decir, la última operación de cada trabajo debe completarse antes de d_j.

2.1 Clasificación de los modelos de programación de la producción

Los problemas de programación de la producción se clasifican atendiendo a tres parámetros, $\alpha|\beta|\gamma$.

Donde α hace referencia a las características de las máquinas (entorno), β a las características de los trabajos (restricciones) y γ al objetivo a minimizar o maximizar.

Entorno del proceso α

Describe el entorno, es decir, el tipo de disposición de la fábrica y el número de máquinas que presenta. Algunos de los entornos más conocidos son:

- Single Machine
- Parallel Machines
- Flowshop
- Jobshop
- Openshop

El entorno Single machine ($\alpha = 1$), es el más simple. Cada trabajo tiene una única operación que se realiza en una única máquina. En este caso, no existe ruta de los trabajos y solo se necesita una secuencia para obtener un programa factible.

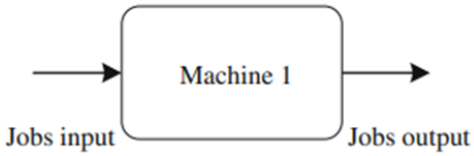

Figura 2.1 Esquema entorno Single machine (Framinan et al., 2014).

El entorno Parallel machines es una réplica del entorno Single machine para el incremento de la producción, cuando se necesita más de un recurso para satisfacer la demanda. Cada trabajo tiene una única operación y solo se procesa en una de las máquinas disponibles. Normalmente, todas las máquinas pueden realizar el trabajo. Para obtener un programa factible solo se necesita una secuencia, sin embargo, el problema se simplifica mediante una secuencia más regla de asignación. Dentro de este entorno, podemos distinguir distintos tipos de máquinas paralelas:

- Identical Parallel machines ($\alpha = Pm$): las máquinas son iguales por lo que el tiempo de proceso p_j es independiente de la máquina.
- Uniform Parallel machines ($\alpha = Qm$): las máquinas poseen diferentes velocidades v_i. Los tiempos de proceso de cada trabajo en cada máquina se calculan de la siguiente forma: $p_{ij} = \frac{p_j}{v_i}$.
- Unrelated Parallel machines ($\alpha = Rm$): las máquinas son diferentes por lo que el tiempo de proceso depende de cada máquina p_{ij}.

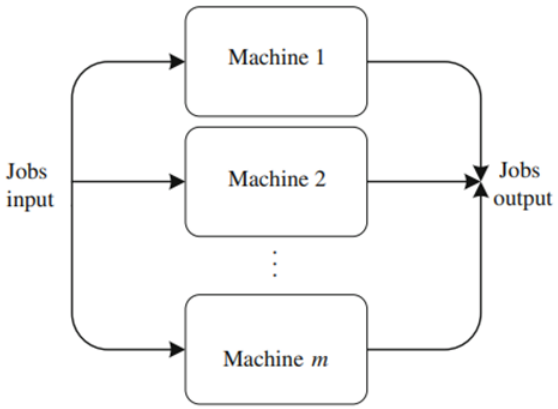

Figura 2.2 Esquema entorno Parallel machines (Framinan et al., 2014).

Por último, encontramos los entornos tipo taller. En estos entornos, las máquinas están dispuestas en serie y cada una de ellas tiene una función única por lo que cada trabajo visita todas las máquinas para procesarse por completo. Existen cuatro tipos de entornos tipo taller:

- Taller de flujo regular o Flowshop ($\alpha = Fm$): todos los trabajos tienen la misma ruta R_j = (1,2,...m), $\forall$ j ε N y se necesita una secuencia para cada máquina. Este tipo, a su vez, tiene un caso particular, conocido como Permutation flowshop ($\alpha = Fm \mid \beta = prmu \mid$). Consiste en una restricción según la cual solo existe una única secuencia para todas las máquinas.

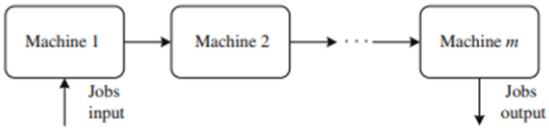

Figura 2.3 Esquema entorno Flowshop (Framinan et al., 2014).

- Taller Job Shop ($\alpha = Jm$): cada trabajo tiene una ruta diferente y estas son un dato necesario. Se necesita una secuencia extendida, formada por $n \times m$ elementos, apareciendo cada trabajo m veces en la secuencia.

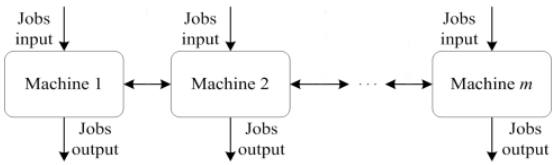

Figura 2.4 Esquema entorno Job Shop (Perez Gonzalez et al., 2021).

- Taller abierto u Openshop ($\alpha = Om$): es el más general y complejo de los talleres. No hay ruta predeterminada, cuando se proporciona el programa se construye la ruta. La secuencia es una secuencia de operaciones formada por $n \times m$ elementos.

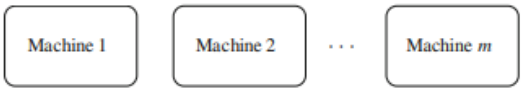

Figura 2.5 Esquema entorno Openshop (Framinan et al., 2014).

- Distributed Permutation Flowshop ($\alpha = DFm$): este entorno se caracteriza por poseer F fábricas idénticas que comparten un conjunto de N trabajos a ser procesados en cada una de las m máquinas o recursos disponibles dispuestos en serie. Puesto que las fábricas son iguales, el tiempo de proceso es independiente de estas, p_{ij}. Un trabajo que se asigna a una fábrica no puede ser asignado al resto de fábricas disponibles. Cada factoría es una entidad independiente por lo que se necesita una secuencia para cada fábrica. En este caso, es necesario asignar trabajos a cada una de las fábricas para asegurar que todos los recursos se utilicen.

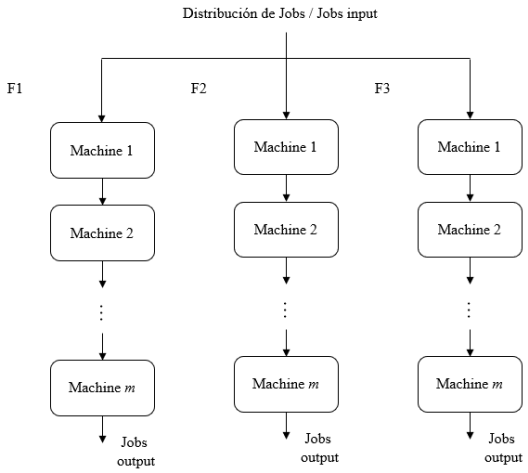

Figura 2.6 Esquema entorno Distributed Permutation Flowshop.

Restricciones β

Las restricciones son las limitaciones del modelo. Aunque no existan restricciones ($\beta = \varnothing$), se hallan unas suposiciones generales:

- Trabajos disponibles al principio del horizonte de programación.
- Los trabajos no se pueden interrumpir.
- Las máquinas siempre están disponibles.
- Cada máquina puede hacer un trabajo y un trabajo puede ser realizado sólo en una máquina.
- El buffer entre máquinas se supone infinito.
- El tiempo de transporte es despreciable.

Existen diversos tipos de restricciones como la restricción de interrupción ($\beta = pmtn$) en la cual, los trabajos u operaciones pueden interrumpirse una vez que han empezado a procesarse o la limitación de tiempos de setup, son tiempos de preparación que necesita la máquina para la siguiente actividad que va a realizar. Algunas de las restricciones más comunes son:

- Interrupción ($\beta = pmtn$): Los trabajos u operaciones pueden interrumpirse una vez que han empezado a procesarse. Esta parada puede ser debida a:
  - Indisponibilidad de las máquinas.
  - Conveniente para mejorar la eficiencia del sistema.
- Fecha de llegada ($\beta = r_j$): la fecha de llegada o disponibilidad de los trabajos son distintas de cero, es decir, no están disponibles al inicio del horizonte de programación.
- Tiempos de setup: son tiempos de preparación que necesita la máquina para la siguiente actividad que va a realizar. Pueden ser independientes de la secuencia ($\beta = s_{ij}$), dependientes de la secuencia ($\beta = s_{ijk}$) o independientes de la máquina, en cuyo caso, se elimina el índice i. A su vez, pueden ser anticipatorios o no anticipatorios.
- Lotes ($\beta = batch$): las máquinas pueden procesar a la vez, como mucho, b trabajos. Hay dos tipos, lotes en paralelo ($\beta = p\text{-}batch(b)$) y lotes en serie ($\beta = s\text{-}batch(b)$)

Por otro lado, encontramos las siguientes restricciones en los entornos de tipo taller:

- Restricción de permutación ($\beta = prmu$): los trabajos se procesan en el mismo orden en cada una de las máquinas.

- Restricción de no inactividad (β = *no-idle*): no se permiten tiempos ociosos de las máquinas entre trabajos. Una vez que la máquina empieza a procesar el primer trabajo de su programa no puede detenerse.
- Restricción no-wait (β = *no-wait*): los trabajos no pueden esperar entre máquinas o etapas.
- Restricción de buffer (β = *buffer* = b_i): el buffer es el espacio donde esperan los trabajos para ser procesados en cada máquina. Esta restricción limita el número de trabajos que se pueden alojar en el buffer.

Objetivo γ

El campo γ representa la función a optimizar. En programación de la producción las funciones objetivo se clasifican según el coste, tiempo, calidad y flexibilidad. El objetivo central es encontrar un programa óptimo, es decir, un programa mejor que cualquier otro programa que también sea admisible.

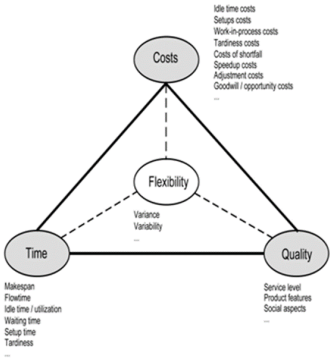

Figura 2.7 Clasificación de los objetivos (Framinan et al., 2014).

Los objetivos, en general, se miden según:

- C_j: tiempo de terminación o completion times de los trabajos.
- F_j: tiempo de flujo o tiempo que el trabajo está en el entorno (flowtime).

$$F_j = C_j - r_j$$

- L_j: retraso del trabajo j (lateness).

$$L_j = C_j - d_j$$

- T_j: tardanza del trabajo j (tardiness). Indica cuánto de tarde se procesa el trabajo respecto a su fecha de entrega.

$$T_j = \text{máx}\{0, L_j\} = \text{máx}\{0, C_j - d_j\}$$

- E_j: adelanto del trabajo j (earliness). Se refiere a la medida de cuánto antes se completa un trabajo en comparación con su fecha de entrega.

$$E_j = \text{máx}\{0, -L_j\} = \text{máx}\{0, d_j - C_j\}$$

- U_j: tardy job o trabajo tarde.

$$U_j = \begin{cases} 1 & \text{si el trabajo } j \text{ está tardío} \\ 0 & \text{si el trabajo } j \text{ no está tardío} \end{cases}$$

En cuanto a los objetivos, se clasifican en aquellos que están vinculados a las fechas de entrega, como por ejemplo, máximo lateness ($máxL_j$) o total lateness ($\sum L_j$), a la máxima tardanza del trabajo ($máxT_j$) o total tardiness ($\sum T_j$), al máximo adelanto del trabajo ($máxE_j$) o total earliness ($\sum E_j$) o al número total de trabajos que van tarde o total tardy jobs ($\sum U_j$). Por otro lado, se encuentran aquellos objetivos que no dependen de las fechas de entrega como el makespan o maximum completion time ($Cmáx$) o la suma de los tiempos en los que termina cada trabajo ($\sum C_j$); el máximo tiempo de flujo o flowtime ($máxF_j$), que es el tiempo que el trabajo está en el entorno o la suma total de este ($\sum F_j$).

3 Descripción del Problema

El problema de estudio, siguiendo la clasificación de programación de la producción $\alpha|\beta|\gamma$, se define como:

$$DFm|prmu,no-idle|\sum_{f=1}^{F} w_f Cmáx^f$$

3.1 Entorno α

El entorno se trata de un Distributed FlowShop *DFm*, una generalización del entorno Flowshop *Fm*, explicado anteriormente. Este entorno se caracteriza por disponer de F fábricas idénticas que comparten un conjunto de n trabajos a ser procesados. Cada uno de los centros de fabricación posee m máquinas dispuestas en serie. El tiempo de proceso, p_{ij}, del trabajo j en la máquina i es idéntico para todas las fábricas, por lo tanto, es independiente de estas. Un trabajo que se asigna a una fábrica no puede ser asignado a ninguna otra fábrica, es decir, los trabajos se distribuyen en las distintas plantas, sin repetición (Naderi & Ruiz, 2016).

3.2 Restricciones β

En cuanto a las limitaciones del problema, encontramos, en primer lugar, la restricción de permutación, es decir, la secuenciación de trabajos es la misma en cada máquina. Se trata de una restricción característica del entorno Flowshop, la cual, simplifica la programación pues los trabajos se procesan en el mismo orden en cada máquina y, un trabajo no puede empezar a producirse en la siguiente máquina hasta que no acaba en la anterior. Por otro lado, tenemos la restricción no-idle, es decir, no se permiten tiempos de inactividad entre trabajos una vez empiezan a procesarse en una máquina. Existen los siguientes componentes del tiempo de inactividad (Maassen et al., 2020):

- Front Idle Time de la máquina i, FIT_i, $2 \leq i \leq m$: tiempo antes de que la máquina i procese el trabajo en la primera posición.
- Core Idle Time de la máquina i, $CIT_{i,[k]}$, $2 \leq i \leq m$, $2 \leq k \leq n$: tiempo entre el instante de finalización del trabajo en la posición anterior *k-1* y el tiempo de inicio del trabajo en la posición k en la máquina i. Para una máquina i, $CIT_i = \sum_{k=1}^{n} CIT_{i,[k]}$.
- Back Idle Time de la máquina i, BIT_i, $1 \leq i \leq m-1$: tiempo después de que la máquina i termina el último trabajo antes de la finalización de la programación general.

  A continuación, se representa, mediante un Diagrama de Gantt, un ejemplo de tiempos ociosos en un entorno Flowshop con permutación, m=3 máquinas y n=4 trabajos:

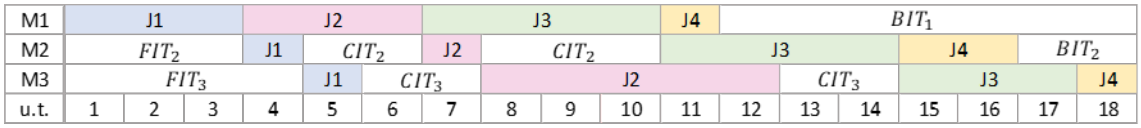

Figura 3.1 Representación de los tiempos ociosos.

En este caso, nos centramos en los Core Idle Times (CIT) de cada máquina.

Como se observa, existen Core Idle Times entre los trabajos en la segunda y tercera máquina, pues, en la primera, nunca aparecen Core Idle Times ni Front Idle Times, todos los trabajos entran en el horizonte de programación al inicio de este y se procesan consecutivamente. A partir de la segunda máquina, empiezan a presentarse estos tiempos ociosos pues para que el trabajo pueda procesarse en la siguiente máquina, debe de haber finalizado en la anterior.

Para solucionar este problema, se retrasa la entrada de los trabajos con el fin de que, una vez que entren en el recurso, se procesen sin paradas. Sin embargo, esto aumenta los Front Idle Times, así como los Back Idle Times. En la siguiente ilustración se representa mediante un Diagrama de Gantt la restricción "no-idle":

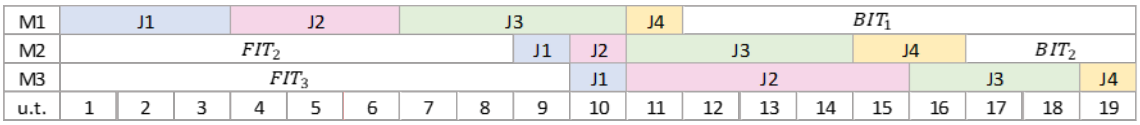

Figura 3.2 Diagrama de Gantt. Restricción no-idle.

En dicho Diagrama se observa cómo los Core Idle Times desaparecen y que, tanto los Front Idle Times como los Back Idle Times, aumentan.

3.3 Objetivo γ

El objetivo por minimizar es la suma del makespan de cada fábrica ponderado por un peso, que, en este caso, es el coste energético de tener en funcionamiento las máquinas de cada manufactura:

$$\sum_{f=1}^{F} w_f Cmáx^f$$

Supongamos que se tienen dos centros de producción, f_1 y f_2, ambos con tres máquinas $m = 3$, cinco trabajos $n = 5$ y los tiempos de proceso son los siguientes:

Tabla 3.1 Tiempos de proceso para un entorno compuesto por n = 5 trabajos, m = 3 máquinas en f = 2 fábricas análogas.

	J1	J2	J3	J4	J5
M1	5	1	4	3	3
M2	2	4	5	2	3
M3	1	4	3	2	4

Por otro lado, se conoce el coste energético de producir en cada fábrica:

Tabla 3.2 Coste energético de producir en cada fábrica.

Fábrica	C. Energético (w_f)
1	3
2	2

Aleatoriamente, se asignan los trabajos J1 y J4 a la fábrica f_1 y J2, J3 y J5 se procesan en la fábrica f_2. A continuación, se representan, mediante Diagramas de Gantt, los resultados o completion times de los trabajos obtenidos tras aplicar el modelo de estudio:

M1	J1					J4											
M2							J1		J4								
M3										J1	J4						
u.t.	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17

Figura 3.3 Diagrama de Gantt representativo de los completion times de los trabajos asignados a la Fábrica nº 1.

M1	J2	J3				J5											
M2		J2				J3					J5						
M3							J2				J3		J5				
u.t.	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17

Figura 3.4 Diagrama de Gantt representativo de los completion times de los trabajos asignados a la Fábrica nº 2.

El makespan se refiere al instante final en que se completa el último trabajo en la última máquina de cada fábrica o máximo completion time ($Cmáx^1 = 12$ y $Cmáx^2 = 17$) y el coste energético w_f se encuentra en la tabla 3.2. Entonces, el coste total de producción se calcula de la siguiente forma:

$$\sum_{f=1}^{2} w_f Cmáx^f = w_1 Cmáx^1 + w_2 Cmáx^2 = 3*12+2*17 = 70$$

Por lo tanto, el objetivo es minimizar este coste total mediante la implementación de métodos heurísticos y metaheurísticas que se estudiarán en el siguiente capítulo.

4 Metodología

Para la resolución del problema propuesto es necesario desarrollar métodos heurísticos y metaheurísticas, pues no es posible obtener una solución óptima en tiempos razonables. Estos métodos ayudan a encontrar una solución aproximada en el espacio de soluciones del problema.

4.1 Heurísticas

Los métodos heurísticos o heurísticas no garantizan encontrar la solución exacta del problema pero sí soluciones muy cercanas a la solución óptima. Suelen ser métodos complejos y diseñados especialmente para cada tipo de problema. En este caso, se estudian dos métodos con los que se obtienen resultados bastante positivos.

En primer lugar, se aplicarán dos reglas de despacho bastante eficaces para el problema $F|prmu|Cmáx$:

- SPT: Shortest Processing Time First
- LPT: Longest Processing Time First

Las reglas de despacho son reglas que ayudan a clasificar y ordenar los trabajos que van a ser programados en base a una prioridad establecida. Estas reglas, en concreto, ordenan el conjunto de trabajos dados en un vector, ya sea en orden creciente de la suma de los tiempos de proceso (SPT) o en orden decreciente (LPT), con lo que se obtiene una secuencia de procesamiento inicial.

Para poder decidir en qué fábrica se procesará cada trabajo, necesitamos una regla de asignación.

En este caso, se van a analizar dos reglas, propuestas en el artículo (Naderi & Ruiz, 2016), para determinar cómo asignar la carga de trabajo entre las fábricas una vez que se ha establecido una secuencia de procesamiento inicial de los trabajos:

- R1: Asignar el trabajo j a la fábrica que, tras la asignación de los trabajos previos y sin considerar este trabajo, tenga menor valor de la FO.
- R2: Asignar el trabajo j a la fábrica que, tras asignarle dicho trabajo, tenga el menor valor de la FO.

Estas dos reglas son conceptual y computacionalmente simples. De hecho: La primera regla no implica ningún coste adicional desde el punto de vista computacional en comparación con el necesario para establecer una secuencia base de procesamiento de los trabajos (Naderi & Ruiz, 2016).

La segunda regla, por otro lado, implica la necesidad de secuenciar las m tareas que componen cada trabajo, además de la secuencia base, para todas las fábricas. En otras palabras, introduce una complejidad adicional del orden mF ($O(mF)$).

Seguidamente, se realiza un ejemplo que ilustra el funcionamiento de ambas reglas:

> Se desea ejecutar la programación de la producción en un entorno compuesto por n = 5 trabajos, m = 2 máquinas, f = 2 fábricas y los siguientes tiempos de proceso:

Tabla 4.1 Tiempos de proceso para un entorno compuesto por n = 5 trabajos, m = 2 máquinas en f = 2 fábricas análogas.

	J1	J2	J3	J4	J5
M1	5	10	4	3	6
M2	8	4	5	2	7

Adicionalmente, se conoce el coste energético de producir en cada fábrica:

Tabla 4.2 Coste energético de producir en cada fábrica.

Fábrica	C. Energético (w_f)
1	3
2	2

Suponemos que aplicando la regla de despacho SPT obtenemos la siguiente secuencia inicial:

$$S = [J4, J3, J1, J5, J2]$$

Empezamos el procedimiento implementando la primera regla:

R1

En primer lugar, denotamos $w_f Cmáx^f$ como el valor de la Función Objetivo en cada fábrica que, en este caso, resulta de multiplicar el makespan de cada planta por su correspondiente coste energético w_f.

- Se empieza cogiendo el primer elemento de la secuencia inicial, J4. Como al principio ambas fábricas están vacías, se asigna J4 a la primera fábrica.
- Se escoge el siguiente trabajo, J3 y antes de asignarlo, se calcula el Objetivo tras añadir el primer trabajo:

$$w_1 Cmáx^1 = 15$$
$$w_2 Cmáx^2 = 0$$

J3 se asigna a la segunda fábrica pues alcanza un menor valor de la FO que la primera.

- Se escoge el siguiente trabajo de la secuencia inicial, J1 y se realiza el mismo cálculo:

$$w_1 Cmáx^1 = 15$$
$$w_2 Cmáx^2 = 18$$

Como la fábrica número uno tiene un valor de la Función Objetivo $w_f Cmáx^f$ menor que la segunda, J1 se asigna a la primera planta. Se procede de la misma forma hasta que todos los trabajos del vector inicial son asignados.

La solución final es $S = [[J4,J1],[J3,J5,J2]]$, es decir, J4 y J1 se procesan en la primera fábrica, por otro lado, los trabajos J3, J5 y J2 en la segunda.

La Figura 4.1 muestra el Diagrama de Gantt de la Fábrica nº 1 con el resultado obtenido tras aplicar la primera regla de asignación a la secuencia inicial S.

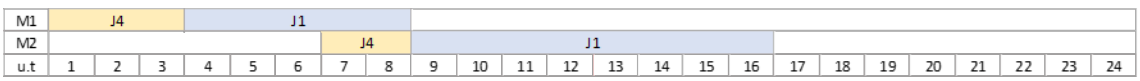

Figura 4.1 Diagrama de Gantt representativo de los trabajos asignados a la Fábrica nº 1.

La Figura 4.2 muestra el Diagrama de Gantt de la Fábrica nº 2 con el resultado obtenido tras aplicar la primera regla de asignación a la secuencia inicial S.

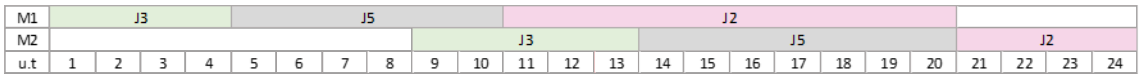

Figura 4.2 Diagrama de Gantt representativo de los trabajos asignados a la Fábrica nº 2.

R2

De nuevo partimos de la secuencia inicial *S= [J4, J3, J1, J5, J2]*. En este caso, primero probamos el trabajo a asignar en todas las fábricas y luego decidimos en qué planta lo asignamos. Esta decisión se toma atendiendo de nuevo al valor de la Función Objetivo.

- Elegimos el primer trabajo de S, J4. Probamos J4 en ambas fábricas, es decir, realizamos el siguiente cálculo:

$$w_1 Cmáx^1 = 15$$
$$w_2 Cmáx^2 = 10$$

Como el valor es menor en la segunda fábrica, asignamos J4 a dicha planta.

- Ahora, elegimos J3 y probamos a introducirlo en la primera y segunda fábrica:

$$w_1 Cmáx^1 = 27$$
$$w_2 Cmáx^2 = 24$$

Asignamos J3 a la segunda fábrica pues $w_2 Cmáx^2 < w_1 Cmáx^1$ tras introducir el trabajo. Por ahora, la asignación es, J4 y J3 a la 2º fábrica:

$$w_1 Cmáx^1 = 0$$
$$w_2 Cmáx^2 = 24$$

- Con el siguiente trabajo procedemos de la misma forma. Insertamos J1 en ambas fábricas:

$$w_1 Cmáx^1 = 39$$
$$w_2 Cmáx^2 = 40$$

Por lo tanto, J1 se asigna a la 1º fábrica.

Una vez se completa el procedimiento con todos los trabajos de la secuencia inicial, se obtiene que la fábrica 1 procesa únicamente el trabajo 1 y la fábrica 2 se encarga de producir J4,J3,J5 y J2 en este orden, es decir, *S = [[J1],[J4, J3, J5, J2]]*

La Figura 4.3 muestra el Diagrama de Gantt de la Fábrica nº 1 resultante de aplicar la regla R2 a la secuencia inicial S.

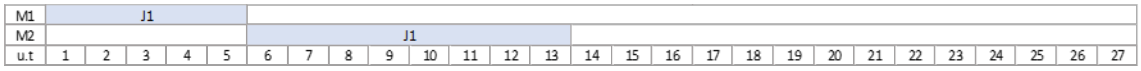

Figura 4.3 Diagrama de Gantt representativo de los trabajos asignados a la Fábrica nº 1.

La Figura 4.4 muestra el Diagrama de Gantt de la Fábrica nº 2 resultante de aplicar la regla R2 a la secuencia inicial S.

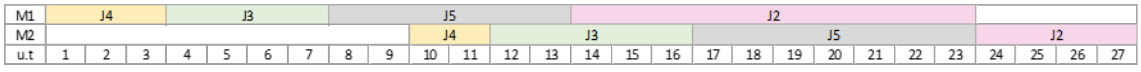

Figura 4.4 Diagrama de Gantt representativo de los trabajos asignados a la Fábrica nº 2.

Por otro lado, se contempla aplicar la heurística NEH (Naderi & Ruiz, 2016) adaptada al entorno Distributed Permutation Flowshop. Esta opción se estudia ya que para el problema $F|prmu|Cmáx$ da resultados muy positivos.

Esta nueva heurística utiliza las dos reglas explicadas anteriormente y el procedimiento es el siguiente:

1. Para cada trabajo se calcula, $p_i = \Sigma p_{ij}$, donde p_{ij} es el tiempo de proceso del trabajo j en la máquina i.
2. Ordenamos en orden decreciente el vector anterior, es decir, se aplica la regla de despacho LPT, mencionada previamente, idónea para el problema pues se usa para minimizar el makespan. Se obtiene la secuencia inicial S. En este caso, se aplica, además, la regla de despacho SPT con el fin de comparar resultados.
3. Se procede a la asignación de los trabajos a las distintas fábricas, para ello, se utilizan las reglas R1 y R2:
   a) NEH1. Se asigna el trabajo j a la fábrica con menor valor de la Función Objetivo, sin incluir dicho trabajo. Una vez asignado, el trabajo j se inserta en todas las posiciones de la secuencia de la fábrica seleccionada. La secuencia resultante será aquella que minimice al máximo el Objetivo.
   b) NEH2. Se asigna el trabajo j a la fábrica que, tras incluir dicho trabajo, alcance el menor valor de la Función Objetivo. Una vez asignado, el trabajo j se inserta en todas las posibles posiciones de todas las fábricas. La secuencia final será aquella que minimice al máximo el Objetivo.

A continuación, se realiza un ejemplo para explicar el procedimiento de ambas heurísticas:

NEH1

Se ordena de mayor a menor la suma de los tiempos de proceso, es decir, se aplica la regla de despacho LPT y se obtiene la secuencia inicial *S = [J2, J1, J5, J3, J4].*

- Se aplica la regla de asignación R1. Asignamos J2 a la primera fábrica y J1 a la segunda pues al principio del horizonte de programación el valor de la Función Objetivo (FO) en ambas fábricas es 0.

$$w_1 Cmáx^1 = 42$$
$$w_2 Cmáx^2 = 26$$

- Cogemos el siguiente trabajo J5 y lo asignamos a la fábrica dos pues $w_2 Cmáx^2 < w_1 Cmáx^1$. Ahora se prueba el trabajo en las posiciones posibles de la fábrica dos:
  - F1 = [J2] con FO=42 y F2 = [J1, J5] con FO = 40
  - F1 = [J2] con FO=42 y F2 = [J5, J1] con FO= 42

Por lo tanto, la secuencia elegida es la primera.

- El siguiente trabajo, se asigna a la segunda fábrica también. Se inserta J3 en todas las posiciones de F2:
  - F1= [J2] con FO=42 y F2 = [J1, J5, J3] con FO=50
  - F1= [J2] con FO=42 y F2= [J1, J3, J5] con FO= 50
  - F1= [J2] con FO=42 y F2= [J3, J1, J5] con FO= 48

Se elige la última secuencia.

Tras realizar el procedimiento con el resto de los trabajos se obtiene la siguiente solución: *F1=[J2,J4]* y *F2=[J3,J1,J5]* y el valor total del objetivo es 96 unidades.

NEH2

Se ordena de mayor a menor la suma de los tiempos de proceso, es decir, se aplica la regla de despacho LPT y se obtiene la secuencia inicial S = [J2, J1, J5, J3, J4].

- Se aplica la regla de asignación R2. Probamos el trabajo J2 en ambas fábricas:

$$w_1 Cmáx^1 = 42$$
$$w_2 Cmáx^2 = 28$$

Se asigna por tanto a la fábrica dos.

- Se selecciona J1 y se inserta en todas las posibles posiciones de todas las fábricas disponibles:

  - F1= [J1] con FO=39 y F2 = [J2] con FO=28
  - F1= [] con FO=0 y F2 = [J2, J1] con FO=46
  - F1= [] con FO=0 y F2 = [J1, J2] con FO=38

En este caso, J1 se asigna a la primera fábrica pues se ha impuesto la restricción de que ninguna fábrica se quede vacía para aprovechar todos los recursos disponibles.

- Se procede de igual forma con J5:
  - F1= [J1] con FO=39 y F2 = [J2 J5] con FO=46
  - F1= [J1] con FO=39 y F2 = [J5, J2] con FO=34
  - F1= [J1,J5] con FO=60 y F2 = [J2] con FO=28
  - F1= [J5,J1] con FO=63 y F2 = [J2] con FO=28

Se selecciona la segunda secuencia.

Una vez se finaliza el procedimiento, se obtiene la siguiente secuencia *S = [[J1, J4],[J3, J5, J2]]* y el valor total de la Función Objetivo es de 93 unidades.

4.2 Metaheurísticas

Las metaheurísticas al igual que los métodos heurísticos no proporcionan soluciones exactas pero sí soluciones bastante cercanas al óptimo. La principal diferencia con las heurísticas reside en que las metaheurísticas son de carácter general, no se diseñan específicamente para cada tipo de problema, por lo que se pueden aplicar para la resolución de una amplia variedad de problemas. Suelen estar inspiradas en procesos o fenómenos naturales como la evolución biológica o la conducta de las colonicas de insectos como el Algoritmo de la Colonia de Hormigas, entre otros. En este caso, se estudian dos metaheurísticas, el algoritmo Iterated Greedy y un algoritmo evolutivo (EA).

Iterated Greedy

El algoritmo Iterated Greedy es uno de los mejores métodos para optimizar problemas de entornos tipo Flowshop y por ello se prueba su eficacia en entornos multifábrica. Se trata de una metaheurística iterativa simple (Ruiz et al., 2018) de alto rendimiento que consta de cuatro operadores:

- Deconstrucción: consiste en destruir la solución inicial eliminando *d* trabajos aleatoriamente. En esta adaptación, solo se extraen trabajos de las fábricas que procesan más de un trabajo denominadas *"allowed_factories"*. Se elige una de dichas fábricas al azar para extraer los trabajos también aleatoriamente. Estos trabajos se almacenan en un vector parcial llamado "*pi_remove*".
- Reconstrucción: para reconstruir la solución se prueba cada trabajo del vector *pi_remove* en todas las posiciones de todas las fábricas y se inserta en la mejor posición de la mejor fábrica.
- Criterio de aceptación: se comprueba si la nueva solución es mejor que la solución inicial. Si la solución es mejor se convierte en la nueva solución inicial y se repite el proceso.
- Criterio de terminación: se establece un criterio de terminación para decidir cuándo acaba de ejecutarse la metaheurística. En este caso, si tras diez iteraciones *(stop=10)* no se consigue ninguna mejora se detiene el algoritmo.

En cuanto al procedimiento de inicialización, utilizamos las secuencias obtenidas de ejecutar los algoritmos NEH1 y NEH2.

Algorithm 1 Iterated Greedy

```
1:  procedure ITERATED_GREEDY(instance, sequences, d, n, stop)  ▷ It is initialized with NEH1 and NEH2
2:      bestseq ← sequences
3:      bestobj ← Cmax_total(bestseq)
4:      flag ← 0                                                ▷ Deconstruction
5:      for i ← 0 to n − 1 do
6:          seq ← sequences
7:          pi_remove ← []
8:          allowed_factories ← factories with more than one job
9:          if allowed_factories == [] then
10:             return bestseq, None
11:         end if
12:         while len(pi_remove) < d do
13:             f1 ← random factory from the allowed_factories vector
14:             f1 ← allowed_factories[f1]
15:             if len(seq[f1]) > 1 then
16:                 j1 ← random job from f1
17:                 pi_remove.append(seq[f1][j1])
18:                 remove j1 from seq[f1]
19:             end if
20:         end while
21:         random.shuffle(pi_remove)                           ▷ Reconstruction
22:         for j in pi_remove do
23:             best_fact, best_pos ← best_position_sequences(instance, seq, j)
24:             seq ← sequence formed by inserting job j in the best position of the best factory
25:         end for
26:         obj ← Cmax_total(seq)                               ▷ Acceptance and termination criteria
27:         if obj < bestobj then
28:             bestseq ← seq
29:             bestobj ← obj
30:             flag ← 0
31:         else
32:             flag ← flag + 1
33:         end if
34:         if flag == stop then
35:             break
36:         end if
37:     end for
38:     return bestseq, bestobj
39: end procedure
```

Evolutionary Algorithm EA

En esta sección proponemos un algoritmo evolutivo simple, denominado EA (Evolutionary Algorithm), que busca iterativamente el óptimo local en el vecindario de los individuos mutados en la población (Fernandez-Viagas et al., 2018).

Los Algoritmos genéticos son técnicas de optimización y búsqueda inspirados en la evolución biológica, en el principio de selección natural de Darwin. Estos algoritmos utilizan mecanismos como la selección, el cruce y la mutación para explorar una variedad más amplia de posibles soluciones. La combinación de mecanismos de cruce y mutación promueve la diversidad en la población de soluciones. Esto conduce a una variedad de nuevas soluciones que pueden ser evaluadas y comparadas, aumentando la probabilidad de encontrar soluciones innovadoras o de muy alta calidad. Debido a su gran capacidad de adaptación se trata de una técnica de Inteligencia Artificial muy usada para resolver problemas complejos, como los que se presentan en Redes Neuronales.

En este caso, se utiliza un Algoritmo Evolutivo para optimizar las soluciones obtenidas con las heurísticas NEH1 y NEH2 y buscar una secuencia más eficiente.

Los operadores utilizados son los siguientes:

- Población: la población es un conjunto de posibles soluciones. Esta población se construye a partir de las heurísticas NEH1 y NEH2. El primer individuo de la población proviene de aplicar estas heurísticas con el vector inicial obtenido de aplicar las reglas de despacho SPT y LPT. El resto de individuos se obtienen cambiando el orden, aleatoriamente, de este vector inicial y volviendo a implementar las heurísticas NEH1 y NEH2 hasta completar la población de tamaño *psize*. La población se ordena de menor a mayor coste tras evaluar los individuos de esta.
- Evaluate: esta función evalúa la solución, es decir, el objetivo, para comprobar cómo de buena es la secuencia. Si alguna fábrica se queda vacía penaliza el coste o función objetivo ($\sum_{f=1}^{F} w_f Cmáx^f$) para que se utilicen todos los recursos disponibles. Esta penalización consiste en sumarle un número muy grande, por ejemplo, 1000 unidades, a la función objetivo.
- Evolve: la función evolve se refiere al proceso de evolución o mejora de la población. En primer lugar, se establece una tasa de reemplazo o "*replace rate*" que indica la proporción de la población que se reemplazará por nuevos individuos (descendencia). Multiplicando esta tasa por el número de individuos de la población actual, se determina el número de nuevos individuos que se generan para la siguiente población (descendientes o hijos). El resto de individuos de la nueva población provienen de la población inicial, se eliminan en orden de esta y se añaden a la nueva generación. A continuación, se seleccionan aleatoriamente los padres de la población actual, y se generan los hijos mediante cruce y mutación, con una probabilidad de mutación establecida en un 50 %. Finalmente, la nueva población se completa con los individuos mantenidos de la población actual y los nuevos hijos generados y se ordena de menor a mayor valor de la FO.
- Cross: la función de cruce combina características de dos soluciones para crear una nueva. Se eligen dos padres, *Parent1* y *Parent2* de la población y se realiza el cruce en un punto. Este punto es una de las fábricas, es decir, se copian los trabajos hasta la fábrica aleatoria o punto de cruce *cross_point* de *Parent1*. Se eliminan de *Parent2* los trabajos que se han copiado de *Parent1*. Para completar el nuevo individuo, se recorren los trabajos restantes en las fábricas de *Parent2* y se colocan en el nuevo individuo en el mismo índice de fábrica en el que estaban en *Parent2*. Con esto conseguimos mayor variabilidad en las soluciones, explorando un número más amplio de resultados.
- Mutate: la función de mutación introduce pequeñas modificaciones aleatorias para explorar nuevas posibles soluciones. Se eligen dos fábricas y dos trabajos al azar de cada fábrica y se inserta el trabajo elegido, en primer lugar, en la posición del segundo, es decir, si se tienen las plantas F1 y F2 y los trabajos J1 y J2, se elimina J1 de F1 y se inserta en la posición de J2 en F2. Es posible que se elija la misma fábrica y el mismo trabajo y, por lo tanto, no se realizaría la mutación.

A continuación, se presentan los pseudocódigos utilizados para la programación de este Algoritmo Evolucionario:

Algorithm 2 Algoritmo Evolucionario

```
1: procedure EvolutionaryAlgorithm
2:     population ← InitializePopulation(instance, initial, psize)
3:     for gen ← 1 to generations do
4:         population ← EvolvePopulation(population, instance, replace_rate, mutate_prob)
5:     end for
6:     best_individual ← GetBestIndividual(population)
7:     return best_individual
8: end procedure
```

Algorithm 3 Inicialización de la Población

```
1: procedure InitializePopulation(instance, initial, psize)
2:     population ← [ ]                          ▷ It is initialized with NEH1 and NEH2
3:     initial_seq, initial_obj ← neh1(instance, initial)
4:     initial_individual ← Individual(initial_seq, instance)
5:     append initial_individual to population   ▷ The population is completed with other sequences
   obtained by swapping jobs from the initial vector
6:     for i ← 1 to psize − 1 do
7:         initial_copy ← initial
8:         pos1, pos2 ← 0, 0
9:         while pos1 == pos2 do
10:            pos1 ← random index in initial_copy
11:            pos2 ← random index in initial_copy
12:        end while
13:        swap initial[pos1] and initial[pos2]
14:        aux_seq, aux_obj ← neh1(instance, initial_copy)
15:        append Individual(aux_seq, instance) to population
16:    end for
17:    SortPopulation(population)
18:    return population
19: end procedure
```

Algorithm 4 Operador `evaluate`

```
1: procedure evaluate
2:     cost ← Cmax_f(sequences)
3:     cost_total ← Cmax_total(sequences)
4:     for each f in sequences do
5:         if f is empty then
6:             cost_total ← cost_total + 1000
7:         end if
8:     end for
9:     return (cost, cost_total)
10: end procedure
```

Algorithm 5 Ordenar Población

```
1: procedure SortPopulation(population)
2:     sort population by EvaluateIndividual(i)n ascending order
3: end procedure
```

Algorithm 6 Selección de padres

```
1: procedure GetParents(population)
2:     parent1 ← random individual from population
3:     parent2 ← random individual from population
4:     return parent1, parent2
5: end procedure
```

Algorithm 7 Evolucionar Población

```
1: procedure EvolvePopulation(population, instance, replace_rate, mutate_prob)
2:     SortPopulation(population)
3:     num_children ← integer result of "len(population) * replace_rate"
4:     new_population ← [ ]
5:     for i ← 1 to num_children do
6:         parent1, parent2 ← GetParents(population)
7:         if random number ≥ mutate_prob then
8:             Mutate(child)
9:         end if
10:        append child to new_population
11:    end for
12:    psize ← length of population
13:    while length of new_population < psize do
14:        append first element of population to new_population
15:        remove first element from population
16:    end while
17:    population ← new_population
18:    SortPopulation(population)
19:    return population
20: end procedure
```

Algorithm 8 Operador `cross`

```
1: procedure cross(individual)                          ▷ Select a random cross point
2:     cross_point ← random factory index               ▷ Select Parent1 and Parent2 from the population
3:     parent1 ← sequences from population
4:     parent2 ← sequences from population
5:     child ← empty sequences array                    ▷ Copy jobs from parent1 up to the cross point
6:     for each f in range(cross_point) do
7:         for each j in parent1[f] do
8:             append j to child[f]
9:             delete_job(parent2, j)
10:        end for
11:    end for                                          ▷ Copy remaining jobs from parent2 in the same factories index
12:    for each f in parent2 do
13:        for each j in parent2[f] do
14:            append j to child[f]
15:        end for
16:    end for
17:    return child
18: end procedure
```

Algorithm 9 Operador `mutate`

```
1:  procedure MUTATE                                          ▷ Select two random factories
2:      f1 ← random factory index
3:      f2 ← random factory index                             ▷ Ensure that the factories are not empty
4:      while sequences [f1] is empty do
5:          f1 ← random factory index
6:      end while
7:      while sequences [f2] is empty do
8:          f2 ← random factory index
9:      end while                  ▷ Select a random job from factory f1 and move it to factory f2
10:     pos1 ← random position from factory f1
11:     job ← find the job in pos1 of f1
12:     remove the job from sequences [f1]
13:     pos2 ← random position in sequences [f2]
14:     insert job at pos2 in sequences [f2]
15: end procedure
```

Algorithm 10 Método `delete_job`

```
1:  procedure DELETE_JOB(sequences, job)                      ▷ Remove job from sequences
2:      for each f in sequences do
3:          for each j in sequences[f] do
4:              if sequences[f][j] == job then
5:                  remove job from sequences[f]
6:                  return
7:              end if
8:          end for
9:      end for
10: end procedure
```

Algorithm 11 Obtener el Mejor Individuo

```
1: procedure GetBestIndividual(population)
2:     SortPopulation(population)
3:     return first element of population
4: end procedure
```

5 Análisis del problema

En este Capítulo se analizan los resultados obtenidos de la ejecución de cada uno de los algoritmos explicados anteriormente. Para ello se han utilizado dos conjuntos de instancias. El primero, presenta instancias de $f \in \{2\}$, $m \in \{5,10,20\}$ y $n \in \{20,50,100,200\}$. Las instancias del segundo conjunto presentan datos de $f \in \{2,3,4\}$, $m \in \{2,3,4,5\}$ y $n \in \{4,6,8,10,12,14,16\}$. La codificación de todos los métodos se ha realizado en Spyder (Python 3.9) debido a que se ha convertido en uno de los lenguajes de programación más populares y versátiles, especialmente en el ámbito de la programación de máquinas y la resolución de problemas de optimización. Para la programación en Python se ha utilizado la librería "scheptk" (Torres, 2022). Esta librería contiene una colección de clases y funciones para modelar y resolver problemas de programación de máquinas. Facilita la lectura de instancias, la generación de secuencias aleatorias y el cálculo de funciones objetivo, entre otras capacidades. Además, para la recopilación de los resultados se ha usado la librería "pandas", la cual permite almacenar datos en formato de tabla utilizando la estructura "DataFrame". Por otro lado, se ha utilizado la librería "openpyxl" para leer y escribir archivos Excel (.xlsx), en los cuales se han recogido los datos obtenidos para su posterior análisis, y el módulo "matplotlib.pyplot" dentro de la librería "matplotlib" que ofrece una interfaz basada en funciones para crear gráficos con los resultados. El código de los algoritmos programados se encuentra en el Capítulo 7.

Para realizar el análisis de los resultados se utiliza la medida RPD o Relative Percentage Deviation, con la cual se evalúa la calidad de la solución obtenida en comparación con la mejor solución o mínimo valor de la función objetivo para todos los métodos de una instancia dada. Valores iguales a 0 indican que la solución obtenida es igual a la solución de referencia o mejor solución y valores cercanos a 0 indican que la solución obtenida se aproxima a la mejor solución. Un RPD bajo indica el algoritmo es bastante preciso mientras que un RPD alto indica que el algoritmo proporciona peores resultados. La fórmula es la siguiente:

$$RPD = \frac{OBJ - MIN}{MIN} \times 100$$

Tras calcular el promedio del indicador RPD para cada método, ilustrado en la tabla 5.1, se concluye que el método con menor ARPD (Average Relative Percentage Deviation) es el Algoritmo Evolutivo (EA) inicializado con la heurística NEH2 y regla de despacho LPT. Esto indica que los objetivos obtenidos con este algoritmo son relativamente más similares o cercanos al mejor valor.

Tabla 5.1 RPD promedio de cada método.

	R1	R2	NEH1	NEH2	IG NEH1	IG NEH2	EA NEH1	EA NEH2
SPT	42,079	32,187	27,645	21,587	22,149	19,864	5,906	6,123
LPT	43,755	37,503	26,374	22,616	22,312	19,791	5,851	5,579

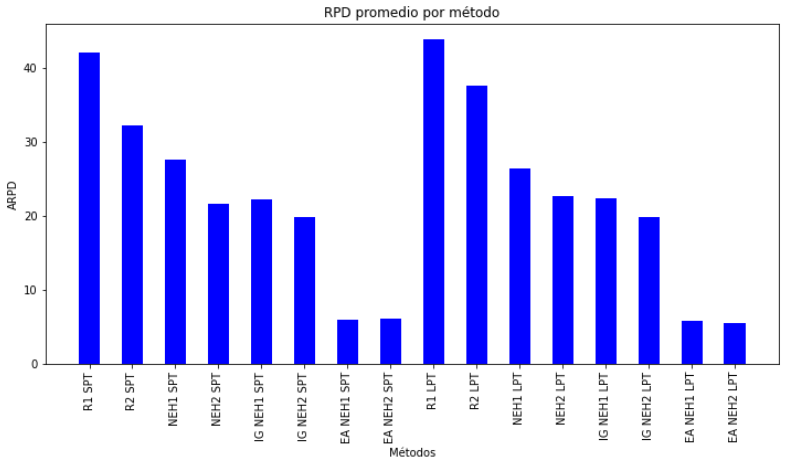

Figura 5.1 RPD promedio por método.

No obstante, estos valores promedios de RPD se han obtenido analizando todo el conjunto de resultados, sin distinguir entre el número de fábricas f, el número de máquinas m o el número de trabajos n de las instancias analizadas. Por ello, se realiza un análisis más exhaustivo calculando el valor promedio RPD según los parámetros mencionados anteriormente, es decir, se analizan aquellas instancias que coinciden en número de fábricas, máquinas y trabajos. Por un lado, se estudia el primer conjunto de instancias o instancias de gran tamaño, que se analizan según el número de trabajos y de máquinas para dos fábricas. Por otro lado, se comparan los valores de las instancias pequeñas para $f \in \{2,3,4\}$.

Tabla 5.2 RPD promedio en función de n y regla de despacho SPT.

n	R1	R2	NEH1	NEH2	IG NEH1	IG NEH2	EA NEH1	EA NEH2
20	48,978	44,483	33,184	27,000	29,315	25,983	4,921	2,596
50	37,958	35,316	20,349	17,884	18,873	17,234	1,238	3,082
100	29,427	28,321	16,546	12,461	15,599	12,111	2,124	2,572

Tabla 5.3 RPD promedio en función de n y regla de despacho LPT.

n	R1	R2	NEH1	NEH2	IG NEH1	IG NEH2	EA NEH1	EA NEH2
20	52,278	47,322	30,814	28,165	28,940	27,555	6,266	3,156
50	39,278	37,281	18,121	15,612	17,172	15,092	1,082	1,770
100	30,363	28,846	13,896	10,322	13,356	10,157	2,654	1,999

En el siguiente gráfico se visualiza la dispersión del RPD promedio obtenido para cada método en función del número de trabajos. Como se puede observar, en general, para una programación con n=100 trabajos el RPD promedio es más bajo, lo que indica que los métodos utilizados son más eficientes para instancias con dicho parámetro. Si se analiza el Algoritmo Evolutivo con NEH2 y LPT, mejor método programado, tanto para n=50 como para n=100 se consiguen muy buenos resultados.

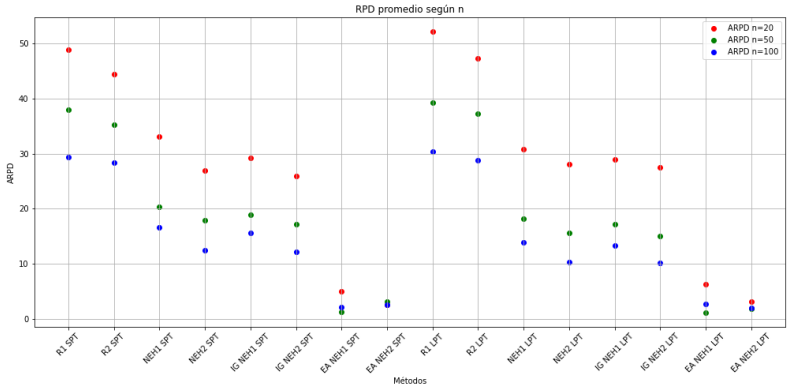

Figura 5.2 RPD promedio en función de n.

Analizando el RPD promedio en función del número de máquinas m del entorno, se observa que, en general, para *m=5* el ARPD es menor tanto para los métodos con secuencia inicial SPT como LPT. En este caso, los valores de RPD promedio son bastantes similares, sin embargo, se vuelve a verificar que para el método EA NEH2 se consiguen los mejores resultados, seguido del método EA NEH1, el cual también posee un ARPD bajo.

Tabla 5.4 RPD promedio en función de m y regla de despacho SPT.

m	R1	R2	NEH1	NEH2	IG NEH1	IG NEH2	EA NEH1	EA NEH2
5	36,402	32,374	20,893	15,720	18,572	15,017	2,183	3,898
10	38,618	35,987	23,202	19,504	20,704	18,824	2,423	1,810
20	40,888	39,387	25,223	21,380	24,127	20,729	3,661	2,639

Tabla 5.5 RPD promedio en función de m y regla de despacho LPT.

m	R1	R2	NEH1	NEH2	IG NEH1	IG NEH2	EA NEH1	EA NEH2
5	35,276	33,069	17,319	13,600	15,936	13,221	2,500	2,020
10	41,948	38,606	20,750	17,509	19,603	17,008	3,257	2,741
20	43,724	41,014	23,912	22,281	23,158	21,844	4,387	2,110

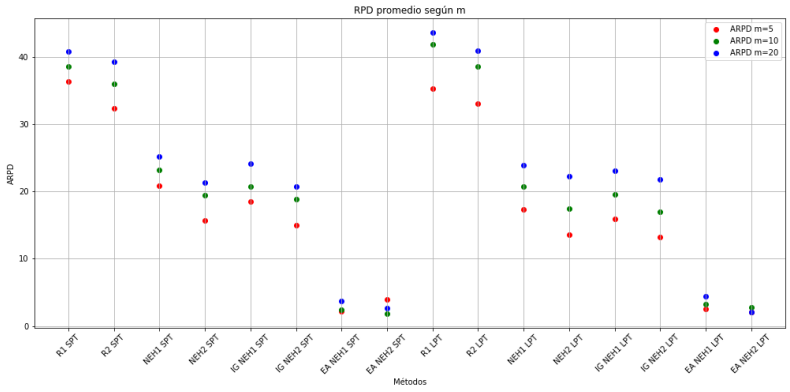

Figura 5.3 RPD promedio en función de m.

Por último, se estudia el ARPD para $f \in \{2,3,4\}$. Para este caso, los valores RPD promedio más bajos se consiguen para *f=3* para las dos metaheurísticas mejor programadas, EA NEH1 y EA NEH2.

Tabla 5.6 RPD promedio en función de f y regla de despacho SPT.

f	R1	R2	NEH1	NEH2	IG NEH1	IG NEH2	EA NEH1	EA NEH2
2	49,269	18,834	37,333	29,833	25,773	28,568	16,070	18,009
3	50,646	30,883	38,082	23,889	26,925	23,889	8,123	6,103
4	51,447	16,174	40,698	29,685	19,624	16,689	17,990	20,457

Tabla 5.7 RPD promedio en función de f y regla de despacho LPT.

f	R1	R2	NEH1	NEH2	IG NEH1	IG NEH2	EA NEH1	EA NEH2
2	58,087	40,178	45,671	40,867	44,423	40,844	15,167	14,783
3	52,506	39,107	40,153	33,128	38,266	32,618	8,790	9,085
4	43,636	29,633	34,836	28,573	24,408	23,494	13,055	18,475

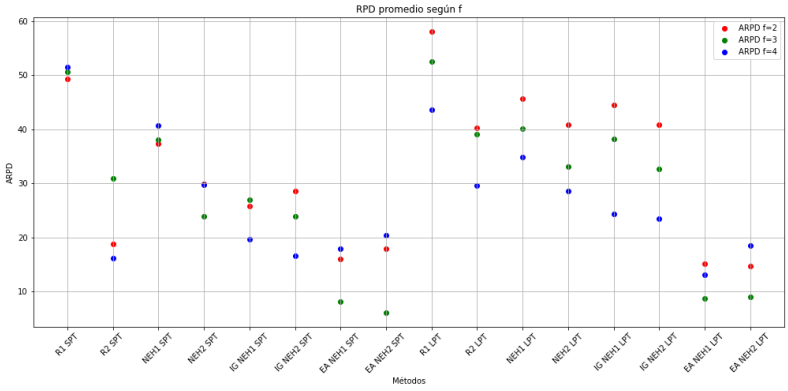

Figura 5.4 RPD promedio en función de f.

Tras el análisis de los resultados obtenidos de la ejecución de los distintos métodos programados se concluye que, en cuanto a heurísticas la NEH2 proporciona un RPD promedio más bajo y que no existen diferencias muy significativas entre la utilización de la regla de despacho SPT o LPT para obtener la secuencia inicial S de la heurística NEH. Sin embargo, los valores más bajos se consiguen con la segunda metaheurística propuesta. El Algoritmo Evolutivo inicializado con las heurística NEH1 y NEH2 y reglas de despacho SPT y LPT alcanza mejores valores para el objetivo en estudio pues explora un amplio número de soluciones.

6 Conclusiones

En la economía globalizada actual, las fábricas ya no funcionan de manera aislada. La fabricación distribuida y las cadenas de suministro complejas tienen repercusiones que se extienden a las decisiones de programación en las plantas manufactureras. La cantidad de escenarios posibles en estos casos es considerablemente amplia (Framinan et al., 2014). Esto supone una complejidad adicional para la distribución de los trabajos en las distintas máquinas de las diferentes fábricas disponibles, a lo cual se suma uno de los problemas a gran escala de la actualidad, la eficiencia energética y la sostenibilidad. La industria manufacturera es uno los principales sectores de uso final de la energía por lo que es de suma importancia realizar una gestión eficiente de dicho recurso.

En este proyecto, se ha realizado un estudio de un entorno Distributed Flowshop, con f fábricas idénticas que comparten un conjunto de n trabajos para ser ejecutados en m máquinas; con restricción de permutación, es decir, la secuenciación de los trabajos es la misma en cada máquina y restricción no-idle, la cual no permite tiempos ociosos entre trabajos. Como objetivo se ha estudiado la suma del makespan de cada fábrica ponderado por el coste energético de tener en funcionamiento las máquinas de estas, para así, minimizar el consumo de energía. Este objetivo se establece con el fin de contribuir con los objetivos de desarrollo sostenible (ODS), en concreto, con el objetivo número doce, el cuál aboga por una producción y consumo responsables. Además, se trata de equilibrar el triángulo ilustrado en el Capítulo 2 (Figura 2.7) o también conocido en gestión de proyectos como triángulo de hierro. Este equilibrio se consigue atacando dos de los tres pilares que lo forman, en este caso, al tiempo y al coste, intentando reducir los tiempos de inactividad o "idle times" y con ello el makespan de cada fábrica.

Para ello, se han programado cuatro heurísticas y dos metaheurísticas. En cuanto a las heurísticas, se han desarrollado dos reglas de asignación (R1 y R2) y dos métodos (NEH1 y NEH2) propuestos en el artículo (Naderi & Ruiz, 2016). Basándose en el RPD promedio, la regla de asignación R2 proporciona resultados más eficientes que R1, lo que resulta lógico pues esta regla primero prueba el trabajo en las fábricas disponibles y lo asigna en el que obtenga menor valor de la Función Objetivo. De igual forma, se consiguen ARPD más bajos aplicando el método NEH2, el cual se compone de la regla R2. En cuanto a las metaheurísticas, se ha adaptado el algoritmo Iterated Greedy del artículo (Ruiz et al., 2018) con el fin de buscar mejores soluciones a partir de los métodos NEH1 y NEH2. Por otro lado, se ha programado un Algorítmo Evolutivo basado en el ilustrado en el artículo (Fernandez-Viagas et al., 2018). Tras el análisis realizado, se ha comprobado que este método es el que alcanza mejores resultados, consiguiendo así, optimizar la Función Objetivo.

7 Anexo Código

En este capítulo se presenta la implementación en Python del modelo $DFm|prmu,no-idle|\sum W_f Cmax$ y todos los métodos programados para su optimización.

7.1 Código principal

A continuación se muestra el código implementado:

```
from scheptk.scheptk import Model
from scheptk.util import read_tag, write_tag, random_sequence
import copy, random

class DF_prmu_no_idle_Cmax_wCE(Model):

    def __init__(self, filename):
        self.jobs = read_tag(filename, 'JOBS')
        self.factories = read_tag(filename, 'FACTORIES')
        self.machines = read_tag(filename, 'MACHINES')
        self.pt = read_tag(filename, 'PT')
        self.Wi = read_tag(filename, 'WI') # peso coste energético por fábrica

    # implementation of the completion times of each job on each machine for FlowShop
    def ct(self, sequences):
        all_ct=[]
        for f in range(self.factories):
            # sequence
            sequence = sequences[f]
            if len(sequence)==0:
                all_ct.append([[0]])
                continue
            # initializing the completion times
            completion_time = [[0 for j in range(len(sequence))] for i in range(self.↩
                machines)]
            # first job in first machine
            completion_time[0][0] = self.pt[0][sequence[0]]
            # first job in all machines
            for i in range(1,self.machines):
                completion_time[i][0] = completion_time[i-1][0] + self.pt[i][sequence↩
                    [0]]
            # rest of jobs in first machine
            for j in range(1, len(sequence)):
                completion_time[0][j] =completion_time[0][j-1] + self.pt[0][sequence[j↩
                    ]]
            # rest of jobs in rest of machines
            for i in range(1, self.machines):
                for j in range(1, len(sequence)):
```

```
                    completion_time[i][j] = max(completion_time[i-1][j], completion_time[i][j-1]) + self.pt[i][sequence[j]]
            # Cálculo de los idle times
            idle_time=[0 for i in range(self.machines)]
            idle_time[0]=0
            for i in range(1,self.machines):
                for j in range(len(sequence)-1):
                    if(completion_time[i][j]<completion_time[i][j+1]-self.pt[i][sequence[j+1]]): # hay idle times
                        idle_time[i]+=(completion_time[i][j+1]-self.pt[i][sequence[j+1]])-completion_time[i][j]
            # initializing the start of each machine
            S=[0 for i in range(self.machines)]
            S[0]=0
            for i in range(1,self.machines):
                S[i]=max(completion_time[i-1][0]+ idle_time [i], S[i-1]+self.pt[i-1][sequence[0]])
            # Cálculo de los completion times
            # primer trabajo en cada máquina
            for i in range(self.machines):
                completion_time[i][0]=S[i]+self.pt[i][sequence[0]]
            # resto de trabajos
            for i in range(self.machines):
                for j in range(1, len(sequence)):
                    completion_time[i][j]=completion_time[i][j-1]+self.pt[i][sequence[j]]
            all_ct.append(completion_time)
        return all_ct, sequence

    def Cmax_total(self, sequences): # valor de la FO total
        obj = 0
        cmax_total = []
        ct, job_order = self.ct(sequences) # job_order=sequence
        for f in range(0,self.factories):
            cmax_total.append(max(ct[f]))
        cmax_total = [job_order[-1] for job_order in cmax_total]
        for f in range(0,self.factories):
            obj+=cmax_total[f]*self.Wi[f]
        return obj

    def Cmax_f(self, sequences): # valor de la FO por fábricas
        fobj = []
        cmax_f = []
        ct, job_order = self.ct(sequences) # job_order=sequence
        for f in range(0,self.factories):
            cmax_f.append(max(ct[f]))
        cmax_f = [job_order[-1] for job_order in cmax_f]
        for f in range(0,self.factories):
            fobj.append(cmax_f[f]*self.Wi[f])
        return fobj
```

7.2 Dispatching Rules y Heurísticas

```

from scheptk.util import sorted_index_asc, sorted_index_desc

import copy, random


```

```
#Dispatching Rules

def SPT(self):

    SPT=[]
    pj=[0 for j in range(self.jobs)]

    for j in range(self.jobs):
       for i in range(self.machines):
           pj[j]+=self.pt[i][j]

    SPT=sorted_index_asc(pj)

    return SPT

def LPT(self):

    LPT=[]
    pj=[0 for j in range(self.jobs)]

    for j in range(self.jobs):
       for i in range(self.machines):
           pj[j]+=self.pt[i][j]

    LPT=sorted_index_desc(pj)

    return LPT

#Heurísticas

def rule1(instance, initial):

    sequences= [[] for _ in range(instance.factories)]

    for j in range(len(initial)):

        obj = instance.Cmax_f(sequences) #calcula el valor de la FO en cada fábrica

        factory=obj.index(min(obj)) #busca la fabrica con menor valor de la FO sin ↩
            incluir el trabajo

        sequences[factory].append(initial[j])

    fobj = instance.Cmax_f(sequences)
    obj_total=instance.Cmax_total(sequences)

    return sequences, obj_total

def rule2(instance,initial):

    sequences= [[] for _ in range(instance.factories)]

    for j in initial:

        factory=get_best_factory(instance, initial, sequences, j)

        sequences[factory].append(j)

    obj = instance.Cmax_f(sequences)
    obj_total=instance.Cmax_total(sequences)

    return sequences, obj_total
```

```
68
69  def get_best_factory(instance, initial, sequences, j): #le paso un trabajo
70
71      seq = copy.deepcopy(sequences)
72
73      for f in range(instance.factories):
74
75          seq[f].append(j)
76
77      obj=instance.Cmax_f(seq)
78
79      factory = obj.index(min(obj)) #busca la fábrica con menor FO tras incluir el ↩
            trabajo
80
81      return factory
82
83  def neh1(instance, initial):
84
85      sequences= [[] for _ in range(instance.factories)]
86
87      for j in range(len(initial)):
88
89          obj = instance.Cmax_f(sequences) #calcula el valor de la FO en cada fábrica
90
91          factory=obj.index(min(obj)) #busca la fabrica con menor valor de la FO sin ↩
                incluir el trabajo
92
93          sequences[factory].append(initial[j])
94
95          best_position = best_position_factory(instance, sequences, factory, initial[j↩
                ])
96
97          sequences[factory].remove(initial[j])
98
99          sequences[factory].insert(best_position,initial[j])
100
101     obj=instance.Cmax_total(sequences)
102
103     return sequences, obj
104
105
106
107 def best_position_factory(instance, sequences, f, j): #se pasa un trabajo y la fá↩
        brica en la que está asignado para probar en todas las posiciones posibles de esa↩
         factoría y elegir la mejor
108
109     bestobj=instance.Cmax_f(sequences)[f]
110
111     bestobj_index=sequences[f].index(j) #best_k da la posicion de j en esa fabrica
112
113     for k in range(len(sequences[f])):
114
115         seq= copy.deepcopy(sequences)
116
117         seq[f].remove(j)
118
119         seq[f].insert(k,j)
120
121         obj=instance.Cmax_f(seq)[f]
122
123         if obj<bestobj:
124
```

```
            bestobj=obj

            bestobj_index=k #si en otra posicion da menor obj, cambia la posicion

    return bestobj_index

def neh2(instance, initial):

    sequences= [[] for _ in range(instance.factories)]

    jobs= copy.deepcopy(initial)

    for j in jobs:

        best_fact, best_pos = best_position_sequences(instance, sequences, j)

        sequences[best_fact].insert(best_pos,j)


        print(sequences)

    obj=instance.Cmax_total(sequences)


    return sequences, obj



def best_position_sequences(instance, sequences, j): #se pasa un trabajo y lo prueba en todas las posiciones de todas las fabricas

    bestobj = None

    bestobj_index = None #best_k da la posicion de j en esa fabrica

    empty_factories = [f for f in range(len(sequences)) if len(sequences[f])==0]

    available_factories = [f for f in range(len(sequences))]

    if len(empty_factories)>0:

        available_factories = copy.deepcopy(empty_factories)

    for f in available_factories: #siempre coge las fabricas disponibles para que no exista ninguna ociosa sin procesar trabajos

        seq = copy.deepcopy(sequences)

        seq[f].append(j)

        best_position=best_position_factory(instance, seq, f, j)

        seq[f].remove(j)

        seq[f].insert(best_position,j)

        obj = instance.Cmax_f(seq)[f]

        if bestobj is None or obj<bestobj:

            bestobj=obj
```

```
184             bestobj_index=(f, best_position) #si en otra posicion da menor obj, cambia la posicion, se obtiene la mejor posición en la mejor fábrica
185
186     return bestobj_index
187
188
189
190 def iterated_greedy(instance, sequences, d=1, n=100, stop=10):
191
192
193     bestseq=copy.deepcopy(sequences)
194
195     bestobj=instance.Cmax_total(bestseq)
196
197     flag=0
198
199     #deconstruccion
200
201     for i in range(0,n):
202
203         seq=copy.deepcopy(sequences)
204
205         pi_remove=[]
206
207
208         allowed_factories = [f for f in range(len(seq)) if len(seq[f])>1] #vector con las fábricas con más de un trabajo de las que se pueden extraer trabajos para el algoritmo
209
210         print(allowed_factories)
211
212         if allowed_factories == []:
213
214             print("No se pueden extraer trabajos, hay 1 solo por fábrica")
215
216             return bestseq, None
217
218         elif len(allowed_factories)>0:
219
220             while len(pi_remove)<d:
221
222                 f1=random.randint(0, len(allowed_factories)-1) #de las fabricas que se pueden extraer trabajos se elige una al azar
223
224                 f1=allowed_factories[f1]
225
226                 if len(seq[f1])>1:
227
228                     j1=random.randint(0,len(seq[f1])-1)
229
230                     pi_remove.append(seq[f1][j1])
231
232                     seq[f1].pop(j1)
233
234             random.shuffle(pi_remove)
235
236            #construccion
237
238             for j in pi_remove:
239
240                 best_fact, best_pos= best_position_sequences(instance, seq, j)
241
```

```
                seq[best_fact].insert(best_pos,j)

            obj=instance.Cmax_total(seq)

            #criterio de aceptacion y de terminacion

            if obj<bestobj:

                bestseq=copy.deepcopy(seq)
                bestobj=obj
                flag=0

            else:
                flag+=1 #si no mejora


            if flag==stop: #si no mejora en 10 iteraciones para

                flag=i

                break

    return bestseq, bestobj
```

7.3 Algoritmo Evolutivo

```


import random, copy

class Individual:

    def __init__(self,sequences,instance):

        self.sequences = copy.deepcopy(sequences)

        self.instance = instance


    def evaluate(self):

        cost=self.instance.Cmax_f(self.sequences)

        cost_total=self.instance.Cmax_total(self.sequences)

        for f in self.sequences:

            if len(f)==0: #evita que queden fábricas sin procesar trabajos

                cost_total+=1000

        return cost, cost_total

    def mutate(self):

        f1=random.randint(0,self.instance.factories-1)

        f2=random.randint(0,self.instance.factories-1)

```

```
        while len(self.sequences[f1])==0:

            f1=random.randint(0,self.instance.factories-1)

        while len(self.sequences[f2])==0:
            f2=random.randint(0,self.instance.factories-1)

        pos1=random.randint(0,len(self.sequences[f1])-1)

        job=self.sequences[f1][pos1]

        self.sequences[f1].pop(pos1)

        pos2 = random.randint(0,len(self.sequences[f2]))

        self.sequences[f2].insert(pos2,job)


    def cross(self, individual):

        cross_point = random.randint(0, self.instance.factories-1)

        parent1, parent2 = copy.deepcopy(self.sequences), copy.deepcopy(individual.↩
            sequences)

        child=[[] for _ in range(self.instance.factories)]

        for f in range(cross_point):

                for j in range(len(parent1[f])):

                    child[f].append(parent1[f][j])

                    self.__delete_job(parent2, parent1[f][j])

        for f in range(len(parent2)):

            for j in range(len(parent2[f])):

                child[f].append(parent2[f][j])

        return child


    def __delete_job(self,sequences, job):

        for f in range(len(sequences)):

            for j in range(len(sequences[f])):

                if sequences[f][j]==job:

                    sequences[f].pop(j)

                    return


    def __str__(self):

        return "Individual: "+str(self.sequences)+" obj: "+ str(self.evaluate()[1])

from Individual import Individual
```

```
from heuristicas import neh2,neh1
import random, copy

class Population:

    def __init__(self, instance, initial, psize):

        self.population=[]

        self.instance=instance

        initial_seq, initial_obj=neh1(instance, initial) #se utilizan NEH1 y NEH2 como secuencias iniciales

        initial_individual=Individual(initial_seq,instance)

        self.population.append(initial_individual) #la población se forma con la clase individual, la secuencia se elige de los métodos NEH1 y NEH2

        for _ in range(psize):

            initial_copy=copy.deepcopy(initial)

            pos1 = 0

            pos2 = 0

            while pos1==pos2:

                pos1=random.randint(0,len(initial_copy)-1)

                pos2=random.randint(0,len(initial_copy)-1)

            initial[pos1],initial[pos2]=initial[pos2],initial[pos1]


            aux_seq, aux_obj=neh1(instance, initial_copy)

            self.population.append(Individual(aux_seq, instance)) #se termina de formar con otras secuencias obtenidas de intercambiar trabajos del vector inicial

        self.sort_population()


    def sort_individual(self, individual):

         return individual.evaluate()[1] #devuelve el coste de cada individuo de la población

    def sort_population(self):

        self.population.sort(key=self.sort_individual, reverse=False) #ordena por el coste y reverse = False hace que el menor coste aparezca el primero

    def get_parents(self):

        parent1=self.population[random.randint(0, len(self.population)-1)]

        parent2=self.population[random.randint(0, len(self.population)-1)]

        return parent1, parent2
```

```
151
152     def evolve(self, replace_rate=0.2, mutate_prob=0.5):
153
154         self.sort_population()
155
156         num_children=int(len(self.population)*replace_rate) #número de hijos que se ↩
                generan para la siguiente población
157
158         new_population=[]
159
160         for _ in range(num_children):
161
162             parent1,parent2=self.get_parents()
163
164             child=Individual(parent1.cross(parent2), self.instance) #parent1.instance ↩
                    o parent2.instance
165
166             if random.random()>=mutate_prob:
167
168                 child.mutate()
169
170             new_population.append(child)
171
172         psize=len(self.population)
173
174         while len(new_population)<psize:
175
176             new_population.append(self.population.pop(0))
177
178         self.population=copy.deepcopy(new_population)
179
180         self.sort_population()
181
182     def get_best_individual(self):
183
184         self.sort_population()
185
186         return self.population[0]
```

Índice de Figuras

Índice de Tablas

Bibliografía

Fernandez-Viagas, V., Perez-Gonzalez, P., & Framinan, J. M. (2018). The distributed permutation flow shop to minimise the total flowtime. *Elsevier*.

Framinan, J., Leisten, R., & Ruiz, R. (2014). *Manufacturing Scheduling Systems: An Integrated View on Models, Methods and Tools*. https://doi.org/10.1007/978-1-4471-6272-8

Maassen, K., Perez-Gonzalez, P., & Günther, L. C. (2020). Relationship between common objective functions, idle time and waiting time in permutation flow shop scheduling. *Elsevier*.

Naderi, B., & Ruiz, R. (2016). The distributed permutation flowshop scheduling problem. *Elsevier*.

Perez Gonzalez, P., Framiñan Torres, J. M., Fernandez-Viagas Escudero, V., & Talens Fayos, C. (2021). *Programación de la Producción*.

Ruiz, R., Pan, Q.-K., & Naderi, B. (2018). Iterated Greedy methods for the distributed permutation flowshop scheduling problem. *Elsevier*.

Ruiz, R., Vallada, E., & Fernández-Martínez, C. (2009). Scheduling in Flowshops with No-Idle Machines. En U. K. Chakraborty (Ed.), *Computational Intelligence in Flow Shop and Job Shop Scheduling* (pp. 21-53). Springer.

Torres, J. M. F. (2022). *Start using scheptk*. https://github.com/framinan/scheptk/wiki/Start-using-scheptk

Wu, C.-C., & Lin, W.-C. (Eds.). (2021). *Theoretical and Computational Research in Various Scheduling Models*. MDPI Mathematics journal. https://www.mdpi.com/books/reprint/4919