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

Taller de flujo regular con capacidad de almacenamiento limitada: Heurísticas constructivas.

Autora: Belén Navarro García
Tutor: Víctor Fernández-Viagas Escudero

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

Sevilla, 2021

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

Taller de flujo regular con capacidad de almacenamiento limitada: Heurísticas constructivas.

Autora:
Belén Navarro García

Tutor:
Víctor Fernández-Viagas Escudero
Profesor Titular de Universidad

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

Sevilla, 2021

Trabajo Fin de Máster: Taller de flujo regular con capacidad de almacenamiento limitada: Heurísticas constructivas.

Autora: Belén Navarro García

Tutor: Víctor Fernández-Viagas Escudero

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

Presidente:

Vocal/es:

Secretario:

Acuerdan otorgarle la calificación de:

El Secretario del Tribunal

Sevilla, 2021

Agradecimientos

Me gustaría agradecer a mis profesores, y en especial a mi tutor Víctor, por descubrirme el mundo de la investigación, por darme su confianza, apoyo y motivación, y por acompañarme en el camino del aprendizaje.

Gracias a todos los que han formado parte de mi vida durante este periodo, por haberme aportado valores, experiencias, e invertir su tiempo en mí. A mis compañeros de clase, por todos nuestros viernes de emprendimiento, por aprender juntos en esta etapa de nuestras vidas.

Y por último, pero no menos importante, a mi familia, mis amigos, y mi compañero de vida Álvaro, por su comprensión y amor incondicional, por estar siempre a mi lado impulsándome a perseguir mis metas. En especial, a mi abuelo Bernabé, por enseñarme el valor del conocimiento y su pasión por la ciencia.

Belén Navarro García

Sevilla, 2021

Resumen

Este Trabajo Fin de Máster aborda un problema de programación de la producción en un taller de flujo regular, conocido en inglés como *flow shop*. Se presenta como restricción la permutación de trabajos, y adicionalmente, se consideran limitaciones de almacenamiento entre las máquinas que componen el taller (buffers limitados). Se tiene como objetivo minimizar el tiempo de finalización máximo de los trabajos (makespan), alcanzando una solución de calidad para el problema en un tiempo de cálculo razonable.

Los métodos aproximados propuestos para la resolución del problema de secuenciación son heurísticas constructivas y de mejora En concreto, se proponen algoritmos heurísticos basados en memoria, en donde se prioriza en caso de empate las secuencias con menor tiempo ocioso entre el procesado de los trabajos y menor tiempo de bloqueo de las máquinas. Algunos de los métodos incluyen una búsqueda local iterativa en cada paso de construcción de la secuencia, repitiendo las combinaciones de trabajos más prometedoras en futuras iteraciones (al contrario que la búsqueda tabú).

Se desarrollan 6 heurísticas constructivas distintas, a las cuales también se les añade una búsqueda local reducida de inserción para ampliar la calidad de la solución, dando lugar a 6 heurísticas de mejora. Estos algoritmos se evalúan para distintos parámetros de entrada, obteniendo un total de 60 variantes. Adicionalmente, han sido reimplementadas 24 heurísticas propuestas en la literatura para resolver el problema abordado o similares, como es el problema de secuenciación de flowshop con bloqueo (cuando la capacidad de los buffers entre las máquinas es cero).

La evaluación computacional llevada a cabo para las conocidas instancias de referencia de Taillard, demuestra la eficiencia de las 84 heurísticas comparadas, consiguiendo, por lo general, mejores resultados las heurísticas propuestas que las existentes en la literatura.

Abstract

This thesis addresses the permutation flow shop scheduling problem (PFSP) with intermediate limited buffers located between two consecutive machines. The goal of this study is to minimize the maximum completion time (i.e., makespan). To deal with it, several approximate algorithms are proposed to achieve a good solution for the problem in a reasonable computational time.

These proposed approximate methods are constructive heuristics that consider the influence of the idle-time and the blocking-time to break ties with the same makespan. Some of them are based on a memory mechanism, which works in a contrary way that the tabu search, so they include an iterative local search at each step of the sequence construction, repeating the most promising job combinations in future iterations.

As a result, six heuristics are developed, to which a reduced local search by insertion is also added to increase the quality of the solution, resulting in 6 enhancement heuristics. Their performance has been tested on the well-known Taillard's benchmark instances.

Computational simulations and comparisons are provided for a total of 84 heuristics. It is shown that the proposed algorithms are capable to generate better results than the state-of-the-art algorithms in terms of solution quality and efficiency. In addition, the algorithms are competitive with the algorithms proposed in the literature for solving the blocking flow shop scheduling problem (i.e., LBPFSP with zero-capacity buffers).

Índice

ÍNDICE DE TABLAS

ÍNDICE DE FIGURAS

1 INTRODUCCIÓN

Este primer capítulo introductorio describe el propósito del Trabajo Fin de Máster (TFM), la motivación que ha llevado a su realización, su ámbito y alcance, así como un breve resumen del contenido del presente documento.

1.1 Objeto del Proyecto

En una sociedad tan competitiva como la actual, las empresas deben poseer herramientas que les permitan alcanzar la eficiencia requerida por el mercado. Para poder satisfacer las necesidades de los consumidores, que cada vez son más exigentes, se requiere una mejora continua y una optimización de todos los procesos.

Dentro de la organización de la producción, la programación de operaciones es una especialidad en auge, pues aporta soluciones competitivas que optimizan el aprovechamiento de los recursos. El estudio de este ámbito resulta de gran importancia en la industria actual, puesto que multitud de empresas precisan resolver problemas de secuenciación para mejorar la eficiencia de sus plantas de producción, reduciendo costes y tiempos de entrega (entre otros) para superar así la competencia del sector.

Como indica Pinedo (2016), la secuenciación de tareas conlleva un alto volumen de decisiones centradas en la asignación de recursos a tareas en un determinado periodo temporal, con el fin de optimizar dicha colocación en base a uno o múltiples objetivos fijados. Establecer una planificación adecuada facilita la toma de decisiones en el largo plazo. Además, la secuenciación no solo aplica al ámbito industrial, sino que además juega un rol importante en cuanto a logística y distribución, o en cualquier servicio con una duración determinada.

En la actualidad, el objetivo principal en muchos escenarios de producción es el tiempo de finalización de los procesos. Mientras más se reduzcan los tiempos de fabricación, se estará haciendo un mejor aprovechamiento de los recursos de la empresa, reduciendo costes y siendo más eficientes. Mediante una adecuada secuenciación de las tareas a procesar en un sistema productivo se puede conseguir minimizar el tiempo (e implícitamente los costes), consiguiendo un mayor alcance y cumpliendo los objetivos marcados en la producción de manera eficiente.

Además, uno de los entornos de fabricación más comunes es el taller de flujo regular. Este tipo de configuración de taller es uno de los problemas más estudiados en la investigación operativa, debido a que es un problema muy presente en el sector industrial, y los algoritmos desarrollados para este problema se han demostrado muy eficientes en problemas similares a lo largo de los últimos años. Generalmente, se suele considerar que la capacidad de almacenamiento entre las máquinas del taller es ilimitada, sin embargo, esto no se asemeja a los sistemas productivos reales. En el sector industrial se presentan limitaciones de capacidad debido a la necesidad física de que los trabajos esperen a ser procesados en una máquina, si ésta se encuentra ocupada procesando el trabajo previo de la secuencia. Cuando mayor es el tamaño o volumen de los trabajos, esta limitación de capacidad se vuelve mucho más relevante.

Por ello, dado el interés, cada vez mayor, en abordar problemas de secuenciación con características realistas, en este Trabajo Fin de Máster se aborda la resolución de un problema de secuenciación en un taller de flujo regular, con restricciones de permutación y existencia de buffers entre máquinas con capacidad de almacenamiento limitada, para aprovechar la capacidad de los recursos y conseguir una producción óptima, reduciendo el tiempo de finalización máximo de las tareas.

1.2 Objetivo y alcance del proyecto

Este proyecto se enmarca en el campo de la Organización Industrial, dentro del ámbito de la programación de la producción. Como se ha mencionado, el objetivo de este proyecto será el estudio de un problema de programación de la producción: un taller de flujo regular con restricción de permutación de trabajos y con la existencia de buffers entre máquinas con capacidad de almacenamiento limitada (denotado por $F_m|prmu,b_i|C_{max}$ acorde a (Graham et al., 1979).

Como resultado, la solución que se pretende obtener es una secuencia de trabajos que indique su orden de procesado en el taller. Esta solución será de mayor calidad cuanto menor sea el tiempo de finalización del último trabajo en la última máquina (minimización de *makespan*).

Al tratarse de un problema complejo (*NP-hard*), no se tratará de evaluar todas las combinaciones de soluciones posibles ni de desarrollar métodos exactos, sino que se buscan métodos aproximados que simplifiquen la búsqueda y proporcionen soluciones cercanas al óptimo en un tiempo de cálculo razonable.

Por ello, se pretende desarrollar algoritmos aproximados que puedan ser aplicados en entornos reales de trabajo. En concreto, se combinan heurísticas constructivas con mecanismos propios de metaheurísticas de búsqueda local, con el objetivo de proporcionar mejores soluciones que la conocida heurística NEH (Nawaz et al., 1983), que por lo general, es la heurística constructiva con mejor desempeño para este problema en concreto.

El funcionamiento de la búsqueda local seleccionada será similar al de la conocida búsqueda tabú, a diferencia de que en lugar de conservar en una lista los movimientos prohibidos, se mantendrán las combinaciones más prometedoras que se hayan realizado durante el procedimiento de construcción de la secuencia. Estas combinaciones prometedoras se repetirán para conseguir soluciones de mayor calidad, y evitar que se descarten soluciones, que aunque a priori no sean las mejores, podrían llegar a serlo.

Como se verá más adelante, se presentan seis heurísticas constructivas para resolver el problema que proporcionan soluciones de buena calidad en un determinado tiempo. Generalmente cuando el tiempo que conlleve el cálculo sea mayor, se generarán mejores soluciones.

1.3 Sumario

En cuanto a la estructura del documento, el contenido de este Trabajo Fin de Máster se distribuye en 8 capítulos y un anexo, cuyo contenido se describe a continuación.

- Capítulo 1: Introducción.

  El primer capítulo es un apartado introductorio donde se describe el propósito del TFM y se comenta brevemente su interés de estudio, su aplicabilidad, objetivos, ámbito y alcance. Además, se menciona la metodología aplicada y un pequeño avance de las conclusiones obtenidas tras la investigación.

- Capítulo 2: Marco teórico.

  Esta sección surge con el propósito de contextualizar la naturaleza del problema de decisión dentro del marco de la programación de la producción. En especial, se comentan algunos conceptos teóricos que han servido de base para el análisis y resolución del problema objeto del proyecto, y se aclara la terminología que se utilizará en aspectos concretos.

- Capítulo 3: Estado del arte.

  En este capítulo se presenta una revisión de la literatura que expone los antecedentes relacionados con el problema, así como los métodos de resolución empleados por otros autores para resolver el mismo problema o similares.

- Capítulo 4: Descripción del problema.

  Este capítulo describe en profundidad el problema a resolver. Se exponen sus características principales en cuanto al entorno, restricciones y objetivos, además de comentar cualquier aspecto que ayude a comprender el problema en cuestión.

- Capítulo 5: Análisis de la metodología

  Una vez planteado el problema objeto de estudio, se realiza un análisis de las metodologías existentes que suelen emplearse en la actualidad para la resolución de problemas de programación de la producción similares al que se trata en este proyecto. Se describen con mayor exhaustividad aquellas metodologías que serán consideradas como base para el desarrollo de la metodología de resolución del problema.

- Capítulo 6: Aplicación de la metodología.

  Este apartado recoge una descripción detallada de los métodos de resolución propuestos para resolver el problema descrito en el capítulo 4. En concreto, se presentan 6 heurísticas constructivas y la adición de una búsqueda local a cada una de ellas.

- Capítulo 7: Evaluación computacional.

  Tras la implementación en lenguaje de programación C# de los algoritmos descritos en el capítulo anterior, en este capítulo se presenta un análisis de los resultados obtenidos al resolver el problema. Para ello, se especifica el conjunto de pruebas realizadas, y se muestra una comparativa de los resultados obtenidos en cuanto al nivel de calidad de la solución y al tiempo computacional del algoritmo.

- Capítulo 8: Conclusiones.

  Para finalizar, este último capítulo recoge las conclusiones extraídas tras el estudio, así como las posibles futuras líneas de investigación que se podrían plantear.

Al final del documento se encuentran las referencias bibliográficas, además de un Anexo que contiene el código en C# de los algoritmos desarrollados.

2 MARCO TEÓRICO

Con el objetivo de enmarcar el desarrollo del proyecto en el ámbito de la organización de la producción, en este capítulo se introducen algunas definiciones y conceptos teóricos relacionados con el área de estudio.

2.1 Definiciones

A lo largo del documento, se emplearán términos específicos que conviene definir para una correcta comprensión del presente trabajo.

- Trabajos o tareas (*jobs*): operaciones cuya realización es necesaria para la fabricación de un producto a partir de materia prima. El concepto de trabajo puede representar tanto a un único objeto, como a un grupo de objetos físicos.
- Máquinas (*machines*): recursos con capacidad productiva para el procesado de trabajos. Abarca recursos físicos (objetos mecánicos), como recursos humanos (operarios o un grupo de objetos mecánicos que realicen una operación en su conjunto).
- Tiempo de proceso (*processing time*): duración temporal de una operación en una máquina. Dicho tiempo dependerá tanto del trabajo a procesar como de la máquina en la que se realice la operación.
- Programa (*schedule*): resultado de la programación de la producción que recoge la asignación en la escala temporal concreta de las máquinas para el procesado de los trabajos. En el programa quedan reflejadas las fechas de comienzo de cada trabajo en cada máquina, y para que sea factible, debe cumplir con todas las restricciones y características del entorno productivo.
- Secuencia (*sequence*): serie de trabajos que marca su orden de procesado en las máquinas.

2.2 Introducción a la programación de la producción

Según Framinan et al. (2014), la organización de la producción (*Production Management*) "es un proceso industrial consistente en la toma de un número elevado de decisiones a lo largo del tiempo para asegurar la entrega al cliente de los productos con la máxima calidad, mínimo coste, y mínimo tiempo de entrega". Estas decisiones se toman dentro de un proceso productivo en el que se realizan transformaciones. Todas las acciones de planificación y gestión que incluye cualquier proceso productivo son abarcadas por la organización de la producción, que engloba un gran volumen de decisiones diversas que aplican a distintos ámbitos y plazos de tiempo.

Entre todos los aspectos de planificación que engloba la organización de la producción, en concreto, la programación de la producción (*Manufacturing Scheduling*) forma parte de la toma de decisiones a corto plazo relativas al proceso de fabricación. La programación de la producción se encarga de ordenar de forma consecutiva las tareas para que puedan ser procesadas por los recursos en un determinado periodo de tiempo, con la misión de cumplir con un objetivo concreto. Como cualquier proceso de toma de decisiones, la secuenciación juega un papel importante en los sistemas de fabricación, producción, transporte, distribución, e incluso algunos tipos de servicios (Pinedo, 2016).

Multitud de investigadores han contribuido significativamente en este área, desde problemas clásicos de la literatura hasta aplicaciones prácticas (Allahverdi, 2015; Allahverdi et al., 2008; Gupta & Stafford, 2006). En especial, la secuenciación de la producción en el taller de flujo con permutación ha sido uno de los problemas

de secuenciación que más estudiados de la investigación operativa, desde el artículo seminal de Johnson (1954). Este entorno es bastante común en los escenarios reales de fabricación, ya que presenta una serie de ventajas respecto a configuraciones más genéricas de talleres (ver por ejemplo Krajewski et al., 1987), así como, muchos talleres se pueden reducir a talleres de flujo en la mayoría de los trabajos (Storer et al., 1992).

Adicionalmente, la minimización del makespan es uno de los objetivos más frecuentes. Esto se debe a que una minimización del tiempo de finalización máximo provoca la minimización de la producción total, lo que maximiza la utilización de las máquinas y minimiza los costes unitarios fijos (Fernandez-Viagas & Framinan, 2014).

Sin embargo, en la mayoría de las investigaciones realizadas sobre este campo, se suele asumir que existe una capacidad de almacenamiento infinita entre las máquinas del taller. Esto no se asemeja a los sistemas productivos reales, puesto que siempre se presenta una limitación de capacidad debido a la necesidad física de que los trabajos esperen a ser procesados en una máquina, si ésta se encuentra ocupada procesando el trabajo previo de la secuencia. Cuando mayor es el tamaño o volumen de los trabajos, esta limitación de capacidad se vuelve mucho más relevante. Varios autores han estudiado el problema de Flow-shop con la restricción de bloqueo, pero, por el contrario, la existencia de buffers con capacidad limitada entre las máquinas ha sido objeto de estudio solo por algunos autores.

Cabe mencionar que la programación de la producción conlleva una serie de decisiones exhaustivas y precisas, que deben ser tomadas en el instante adecuado, puesto que pueden influir en gran medida en el resultado final. Por este motivo, el tiempo es un factor muy importante por considerar a la hora de desarrollar un método, que a veces debe proporcionar decisiones instantáneas, incluso en tiempo real. Dichas decisiones vienen condicionadas por diversos factores que dependen del tipo de entorno en el que nos encontremos, las restricciones impuestas, o los objetivos fijados. Graham et al. (1979) propone una de las notaciones que más se usan en la actualidad con el fin de definir las características de un problema de programación de la producción, basada en el uso de tres parámetros de la siguiente forma: α | β | γ. Cada uno de ellos describe las características de las máquinas (entorno, α), de los trabajos (restricciones, β), y de la función objetivo (objetivos, γ), respectivamente (véase Figura 2.1).

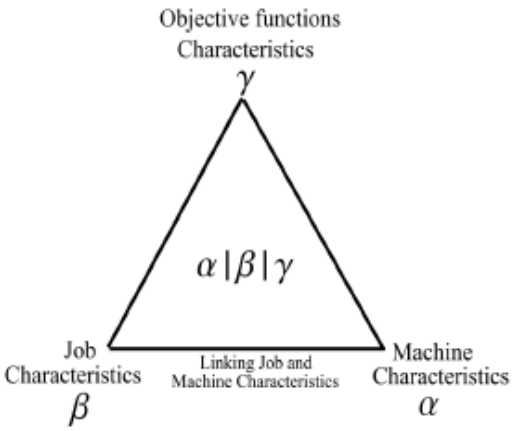

Figura 2.1. Representación de los entornos, restricciones y objetivos.
[Fuente: Temario Programación y Control de la Producción, Escuela Técnica Superior de Ingeniería]

2.2.1 Entorno (α)

Existen distintos tipos de entornos de fabricación (*layout*) con distintas características en función del número, distribución y características de las máquinas. En la Figura 2.2. se muestra una clasificación de algunos de los entornos más comunes. En este documento, se aborda un entorno de tipo taller, en concreto un taller de flujo regular.

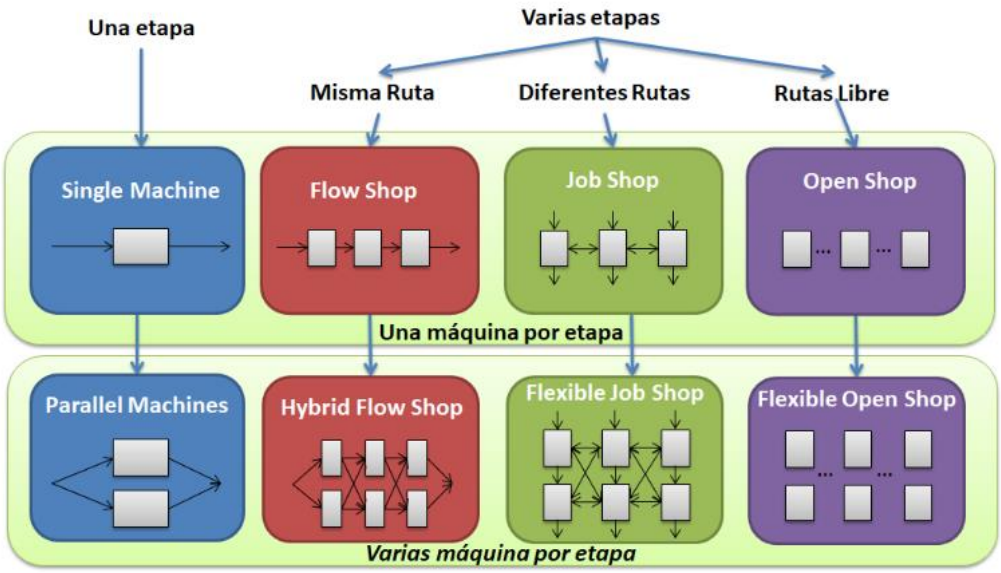

Figura 2.2. Entornos clásicos de fabricación. [Fuente: (Fernández-Viagas, V., s.f.)]

La configuración tipo taller se caracteriza por disponer de varias máquinas en serie, donde cada una realiza una operación distinta sobre el trabajo, el cuál debe pasar por las máquinas del taller según su ruta de fabricación. Según el tipo de ruta que siguen los trabajos a la hora de ser procesados, se diferencian 3 tipos de talleres:

- Taller de flujo (*Flow Shop,* $\alpha = F_m$): los trabajos siguen la misma ruta de fabricación predeterminada.
- Taller de trabajos (*Job Shop,* $\alpha = J_m$): cada trabajo sigue una ruta predeterminada distinta.
- Taller abierto (*Open Shop,* $\alpha = O_m$): la ruta de los trabajos es libre, no está predeterminada.

El taller de flujo regular conocido comúnmente por el término anglosajón *Flow Shop*, además de ser uno de los problemas más estudiados en la actualidad, es el tipo de problema abordado en este documento. Por ello, conviene conocer sus características y las principales particularidades de este tipo de entorno, que se detallan a continuación.

Como se ha mencionado, este taller se compone de varias etapas de producción de una sola máquina. El número de máquinas, y por tanto el número de etapas, viene definido mediante el subíndice *m*. Los trabajos procesados en este entorno se caracterizan por seguir una misma ruta de fabricación, es decir, todos pasan por las mismas máquinas en el mismo orden, como se muestra en el esquema de la Figura 2.3.

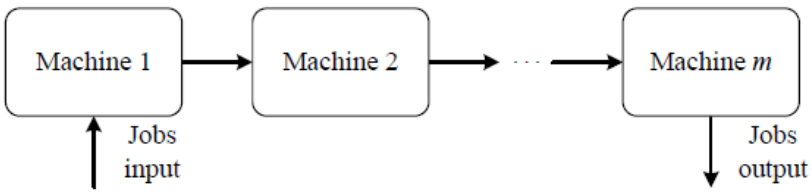

Figura 2.3. Esquema de flujo en un entorno *Flow-Shop*.
[Fuente: Temario Programación y Control de la Producción, Escuela Técnica Superior de Ingeniería]

Por último, los modelos de programación de la producción pueden clasificarse según su complejidad como: polinomiales (*P*) y no polinomiales (*NP-hard*). Los de clase *P* son de menor complejidad, y por tanto su resolución es más simple; pero en cambio los de clase *NP-hard* son más complejos, y es poco probable encontrar una solución óptima para una instancia realista en un tiempo polinomial. El problema de flowshop pertenece generalmente a los de tipo *NP-hard* debido a su nivel de complejidad, ya que resolverlo de forma

óptima implica evaluar $(n!)^m$ posibles programas para una instancia de *n* trabajos y *m* máquinas. Por tanto, no suele ser posible encontrar una solución óptima en un tiempo razonable, y se suele optar por emplear métodos aproximados para su resolución, con el objetivo de alcanzar una solución que se aproxime a la solución óptima en la mayor medida posible, en un tiempo de cómputo admisible y razonable.

2.2.2 Restricciones (β)

Según las características de los trabajos que son procesados en las máquinas, el sistema presentará ciertas restricciones. Las suposiciones generales por defecto presentes en los problemas de programación de la producción (cuando el campo del parámetro β se encuentra vacío, β = ∅) son las siguientes:

- Disponibilidad total de máquinas y trabajos en el instante inicial.
- No existe la posibilidad de interrumpir un trabajo que está siendo procesado. Una vez que inicia, se debe continuar hasta que transcurra por completo su tiempo de proceso correspondiente.
- Una máquina no puede procesar varios trabajos simultáneamente.
- Un trabajo no puede ser procesado en varias máquinas simultáneamente.
- El buffer entre máquinas es infinito, no hay un límite de unidades de trabajos en cola de espera para ser procesados en una máquina.
- Los tiempos de transporte entre máquinas es despreciable.

En cambio, cuando el campo del parámetro β no se encuentra vacío, significa que además de las restricciones anteriores, se presentan una o varias restricciones adicionales. En este documento se mencionarán algunas restricciones relacionadas con el procesado de trabajos o la capacidad de almacenamiento de los buffers entre las máquinas, que se detallan en la Tabla 2.1.

Tabla 2.1. Restricciones habituales en el campo β.

Restricción	β	Descripción
Permutación	*prmu*	En entornos compuestos por varias etapas, el orden de procesado de los trabajos en cada etapa es el mismo.
Tiempos ociosos no permitidos	*No-idle*	No se permiten tiempos ociosos en las máquinas. Es decir, no puede transcurrir tiempo entre el procesado de dos trabajos distintos en una misma máquina.
Almacenes de máquinas (*buffer*)	*Buffer* = b_i	La capacidad de espera de los trabajos para ser procesados en una máquina o etapa *i* está limitada a b_i trabajos.
Bloqueo	*block*	No se dispone de *buffers* entre las máquinas, es decir, la capacidad de almacenamiento intermedio se considera cero. Como consecuencia, cuando un trabajo termina de ser procesado en una máquina, la bloquea hasta que la siguiente máquina esté disponible.
Esperas no permitidas	*No-wait*	No se permite la espera de trabajos entre etapas (inexistencia de buffer entre máquinas).

2.2.3 Objetivos (γ)

El último de los aspectos mencionados para tener en cuenta a la hora de describir un problema de programación de la producción se corresponde con los objetivos. A la hora de buscar una solución se fija un objetivo a conseguir, que viene marcado por una función objetivo, cuyo valor se pretende que sea mínimo, máximo, fijo y predeterminado, u acotado en un intervalo.

Las funciones objetivo se pueden clasificar en 4 categorías en función de las 3 variables que componen el conocido "triángulo de hierro": tiempo, coste, y calidad (véase Figura 2.4). Estas tres variables están relacionadas entre sí, ya que el aumento de una de ellas suele implicar una variación del resto. Por ejemplo, si se requiere un producto con alta calidad, su coste y tiempo de fabricación serán más elevados. El equilibrio entre las tres se conoce como flexibilidad.

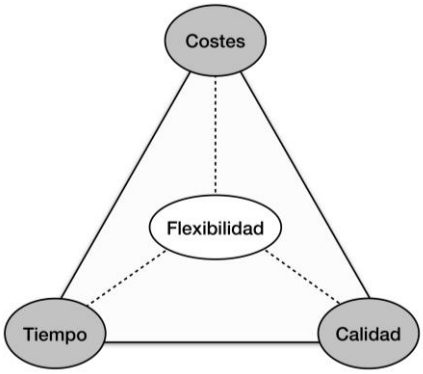

Figura 2.4. Representación del triángulo de hierro.

El objetivo que se aborda en este documento está relacionado con la variable tiempo. Cada trabajo procesado en una máquina tiene un tiempo de proceso determinado, y una vez transcurrido, el trabajo se considera finalizado. Al instante de finalización del proceso se le denomina tiempo de terminación (*Completion time*, C_j). Esta medida puede formar parte de la función objetivo tanto en formato sumatorio (*p.ej.* $\sum C_j$) como en formato de máximo (*p.ej.* C_{max}, que mide el instante de finalización del último trabajo, se conoce como *makespan*).

3 NOTACIÓN Y DESCRIPCIÓN DEL PROBLEMA

El problema objeto de estudio se caracteriza por ser un entorno *Flow-shop* con restricción de permutación entre trabajos, donde existen buffers entre cada máquina que permiten que una cantidad limitada de tareas puedan permanecer en espera para ser procesada en la siguiente máquina del taller. Se busca como objetivo un programa que proporcione el mínimo *makespan*. En este capítulo se describirán en profundidad todos los aspectos relativos al problema.

3.1 Notación

A continuación, se presenta la Tabla 3.1, que resume la notación que es usada a lo largo del documento.

Tabla 3.1. Notación

Notación	Descripción
n	Número de trabajos a procesar
m	Número de máquinas que componen el taller
p_{ij}	Tiempo de proceso del trabajo j en la máquina i
b_i	Capacidad del buffer tras la máquina i
Tiempo de bloqueo (blocking time)	Tiempo de bloqueo de la máquina
Tiempo ocioso (idle time)	Tiempo ocioso de las máquinas durante el procesado de trabajos
Holgura (slack)	Holgura de capacidad en el buffer
C_{ij}	Tiempo de terminación del trabajo j en la máquina i
C_{max}	Instante de terminación del último de los trabajos
F_m	*Flow-shop* de m máquinas
prmu	Restricción de permutación
FO	Función objetivo
Π	Secuencia de trabajos

3.2 Descripción del problema

En este documento se aborda un problema de programación de la producción. En concreto, se analiza la secuenciación de tareas en un taller de flujo regular compuesto por un número m de máquinas, donde se procesarán n trabajos. Este taller tiene como limitaciones la permutación de trabajos y capacidad de almacenamiento entre las máquinas (buffers limitados). Como objetivo, se pretende generar una secuencia de trabajos ordenados de forma que se completen todas las tareas en el mínimo tiempo posible. Por tanto, puede denotarse como: $F_m \mid prmu, b_i \mid C_{max}$, y es conocido por sus siglas en inglés como LBPFSP (*Limited-Buffer Permutation Flowshop Scheduling Problem*).

Por un lado, el entorno de fabricación ($\alpha = F_m$) se corresponde con la configuración de un taller de flujo regular, conocido como *Flow-shop*. Como se ha mencionado en el capítulo 2.2.1, este tipo de entorno se compone de m etapas, donde cada etapa se corresponde con una sola máquina, y todos los trabajos objeto de las operaciones realizadas en las máquinas siguen la misma ruta de fabricación predeterminada. Así, cada uno

de los *n* trabajos a procesar en un entorno F_m se procesarán una sola vez en cada una de las *m* máquinas que componen el taller.

En cuanto a las restricciones del problema correspondientes al parámetro β, se presentan simultáneamente dos restricciones. Por un lado, la restricción de permutación de trabajos (*prmu*) detallada en el apartado 2.2.2. Cuando un taller de flujo regular presenta esta restricción, la secuencia de procesado de los trabajos es la misma para todas las máquinas. Por ello, la secuencia correspondiente a un programa factible para este problema se representa como un único vector que contiene el orden de procesado del conjunto de trabajos $N = \{1,\ldots, n\}$, válido para todo el conjunto de máquinas $M = \{1,\ldots, m\}$. Por otro lado, la restricción de almacenes de máquinas (*buffers*), que proporciona cierta capacidad de espera a los trabajos para ser procesados en una máquina *i* (limitada a un número b_i de trabajos). Adicionalmente, se consideran todas las suposiciones generales para $\beta = 0$, pero no se considerarán interrupciones de los procesos u otras restricciones que no se especifiquen. (Véase sección 2.2.2)

Respecto al objetivo del problema, se busca minimizar el valor del *makespan* ($\gamma = C_{max}$), es decir, se busca que el instante de fin del último trabajo procesado sea lo más pronto posible. Por tanto, interesará que el conjunto de tareas se procese completamente en el mínimo tiempo posible. Conociendo que el *completion time* (C_{ij}) de un trabajo se corresponde con el instante de finalización del trabajo *i* en la máquina *j*, se deduce que, en este tipo de problema, el C_{max} se corresponde con el *completion time* de la última tarea de la secuencia procesada en la última máquina de la ruta de fabricación.

Por otro lado, en cuanto a los datos de entrada del problema, no será necesario tener en cuenta datos relativos a fechas de llegada al sistema de los trabajos (*release dates*, r_j) o fechas de entrega de estos (*due dates*, d_j). Hay que tener en cuenta que cada trabajo se inicia y se termina de procesar en una máquina en un determinado instante de tiempo, siendo ese intervalo transcurrido el correspondiente a su tiempo de proceso (p_{ij}), el cual sí será dato de entrada conocido. También serán conocidos los datos relativos al número de trabajos *n*, número de máquinas *m*, y capacidad del buffer entre cada máquina *i* (b_i).

Respecto a la dinámica seguida en este tipo de taller, cabe destacar que, cuando un trabajo *j* termina de ser procesado en una máquina *i* y la siguiente máquina ($i + 1$) en la que se debe procesar se encuentra ocupada procesando otro trabajo *k*, el trabajo *j* debe esperar a que el trabajo *k* termine de ser procesado para poder continuar su ruta por el taller. Si se dispone de capacidad de almacenamiento en el buffer *i*, este trabajo puede almacenarse temporalmente ahí, y de esta forma la máquina *i* puede continuar procesando otros trabajos mientras el trabajo *k* termina de ser procesado. Por el contrario, si no se dispone de buffer entre las máquinas o éste se encuentra completo, el trabajo deberá esperar en la máquina *i*, bloqueando dicha máquina. Al bloquear la máquina se impide que se continúe procesando otros trabajos, y se produce un tiempo de bloqueo (*blocking time*), correspondiente al tiempo transcurrido desde que el trabajo *j* termina de ser procesado hasta que el trabajo *k* que está siendo procesado en la siguiente máquina ($i + 1$) finalice y pueda iniciarse el procesado del trabajo *j* en la máquina ($i + 1$).

En la literatura se identifican varios tipos de bloqueo diferentes, en función de cuándo se bloquea una máquina: RS_b (*Release when starting blocking*) y RC_b (*Release when completing blocking*); véase (Miyata & Nagano, 2019). El más frecuente es el RS_b, también conocido en inglés como *blocking before processing*, donde la máquina se mantiene bloqueada hasta que comience a procesarse el trabajo en la siguiente máquina del taller. Es este el tipo de bloqueo que se considera en el problema abordado.

Además del tiempo de bloqueo, otro concepto relevante en este tipo de taller es el concepto de tiempo ocioso (*idle time*). Se denomina así al intervalo de tiempo que se da un una maquina desde que termina de procesar un trabajo hasta que comienza a procesar el siguiente. Si comienza a procesar el siguiente trabajo justo cuando el anterior finaliza, el intervalo de tiempo será nulo, no produciéndose tiempo ocioso. En cambio, si la máquina se mantiene en espera entre el procesado de dos trabajos consecutivos, se genera un tiempo ocioso que influirá en un incremento del *makespan*.

A continuación, se presenta un pequeño ejemplo para mostrar la dinámica que sigue este tipo de taller, con las características y restricciones mencionadas, así como los beneficios de una secuenciación adecuada sobre el objetivo del problema.

Ejemplo 1.2.1. *Hallar una secuencia de trabajos para la siguiente instancia del problema $F_2|prmu, b_i|C_{max}$.*

Número de trabajos: n = 4

Número de máquinas: m = 2

Capacidad del buffer: $b_1 = 1$

Tiempos de proceso (p_{ij}):

m \ n	1	2	3	4
1	2	2	1	5
2	5	2	1	2

En la Figura 3.1 se representa la secuencia de trabajos [1, 2, 3, 4] mediante un diagrama de Gantt. Como se puede observar, cuando un trabajo termina de ser procesado en una máquina, comienza a procesarse en la siguiente máquina (obsérvese trabajo 1). Si ésta se encuentra ocupada, el trabajo permanece en espera en el buffer hasta que se quede libre (trabajo 2). Si el buffer también se encuentra completo, sin capacidad para almacenar más trabajos, el trabajo se mantiene en espera en la máquina en la que se acaba de procesar, bloqueando dicha máquina temporalmente e impidiendo la entrada de nuevos trabajos a la máquina hasta que la siguiente máquina finalice de procesar el trabajo en curso y quede espacio libre en el buffer (trabajo 3). El trabajo 4 tiene un largo tiempo de procesado en la primera máquina, y provoca que la máquina 2 esté disponible para su procesado antes de que se haya finalizado de procesar en la máquina 1. Esto implica que la máquina 2 tenga un tiempo ocioso en el que no procesa ningún trabajo. Además, el buffer no dispone de ningún trabajo en espera para su procesado en la segunda máquina, quedándose vacío durante un periodo de tiempo, lo que se denominará en adelante como tiempo de holgura. En este caso, el tiempo de finalización máximo (C_{max}), que corresponde con el tiempo de finalización del último trabajo procesado en la última máquina, es de 14 minutos.

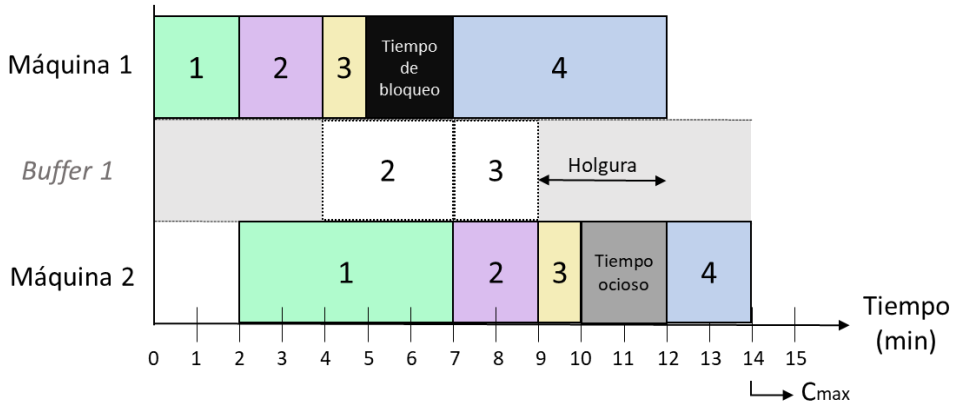

Figura 3.1. Diagrama de Gantt para la secuencia $\prod = [1, 2, 3, 4]$ en $F_2 \mid prmu, b_i \mid C_{max}$.

Para demostrar la influencia que tiene la secuenciación de trabajos sobre el tiempo de finalización máximo, en la Figura 3.2 se representa el diagrama de Gantt de una secuencia de trabajos distinta a la anterior, en este caso la secuencia [4, 1, 2, 3], secuenciando el trabajo 4 el primero en lugar del último. Como se observa, en este caso el C_{max} es de 15 minutos, por lo que se está empleando más tiempo para procesar los mismos trabajos, resultando más ineficiente.

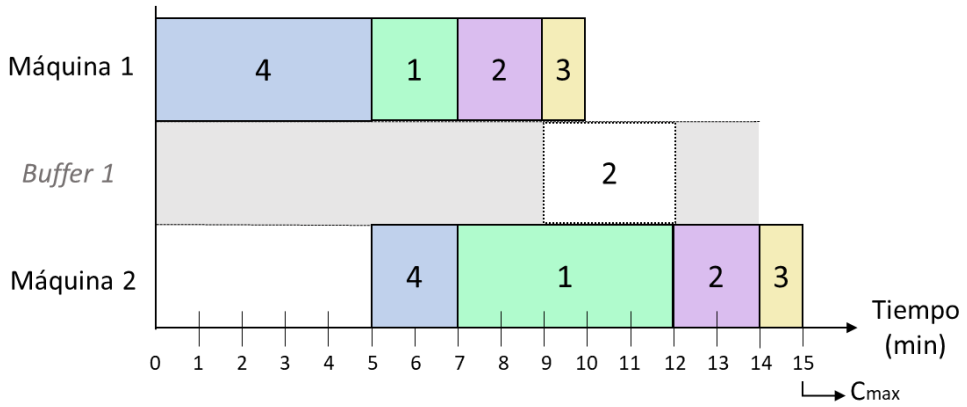

Figura 3.2. Diagrama de Gantt para la secuencia $\prod = [4, 2, 3, 1]$ en $F_2 \mid prmu, b_i \mid C_{max}$.

Se podría considerar que un aumento de la capacidad de almacenamiento del buffer eliminaría el tiempo de bloqueo, y por tanto el tiempo ocioso, que se generaba en la Figura 3.1. A continuación, en la Figura 3.3 se muestra la misma secuencia de trabajos pero ampliando la capacidad del buffer a dos trabajos en lugar de uno. Como se puede comprobar, se consigue reducir el C_{max} a 12 minutos.

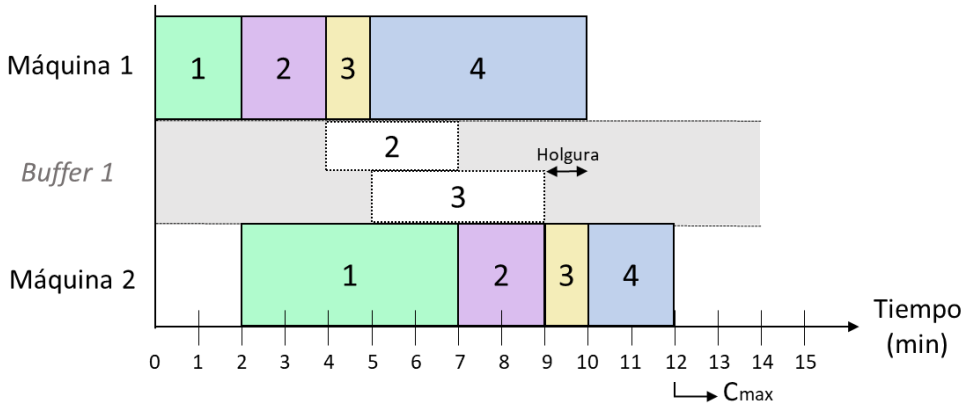

Figura 3.3. Diagrama de Gantt para la secuencia $\prod = [1, 2, 3, 4]$ en $F_2 \mid prmu, b_i \mid C_{max}$.

Sin embargo, una correcta secuenciación de los trabajos mejora el aprovechamiento de los recursos y ayuda a reducir el tiempo de finalización total. En la Figura 3.4 se representa la secuencia de trabajos [3, 2, 1, 4], donde se puede apreciar que se reduce el C_{max} a 12 minutos, sin siquiera hacer uso del buffer entre las máquinas.

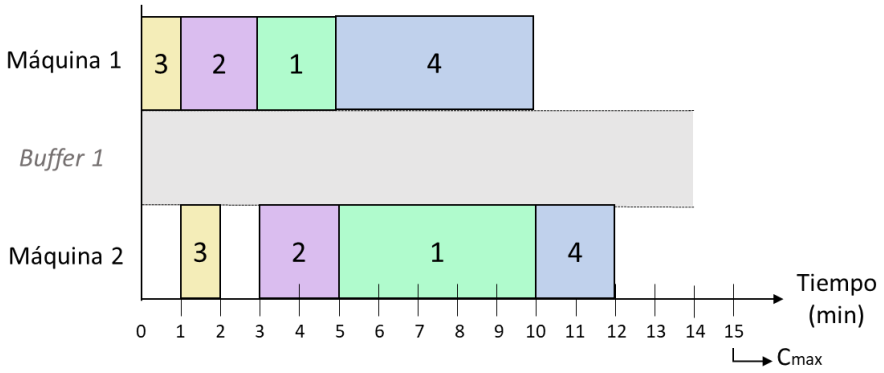

Figura 3.4. Diagrama de Gantt para la secuencia $\prod = [3, 2, 1, 4]$ en $F_2 \mid prmu, b_i \mid C_{max}$.

4 ESTADO DEL ARTE

Como se menciona en el capítulo anterior, este trabajo se centra en la resolución de un problema de secuenciación de tareas, concretamente, en un taller regular de flujo con restricciones de permutación y existencia de buffers entre máquinas con capacidad limitada. En esta sección se analizan los principales artículos de la literatura que tratan de resolver este problema y han servido como referencia para la realización de este proyecto.

4.1 Revisión de la literatura

Dutta y Cunningham (1975) elaboran el primer artículo que deja constancia en la literatura sobre la problemática de secuenciación con limitaciones de capacidad, donde se estudia una configuración taller de flujo regular compuesto por dos máquinas y se desarrolla un procedimiento de programación dinámica para resolverlo. Posteriormente, (Papadimitriou & Kanellakis, 1980) demostraron que el problema con buffer de capacidad limitada suponía una complejidad de tipo NP-hard, y plantearon la primera heurística específica para resolver este tipo de problemas. Estos autores, además, identificaron que la relación entre la fecha de fin del último trabajo (C_{max}) con un buffer de capacidad b entre las dos máquinas y el mismo problema sin buffer es de $(b+1)/(2b+1)$, siendo b la cantidad de trabajos que puede contener el buffer simultáneamente. En Knopf (1985) se aborda el problema de manera exacta en un Flow-shop con dos máquinas mediante un procedimiento de ramificación y acotación. Sin embargo, el primero en plantear un modelo general para incorporar las restricciones de capacidad a los problemas de secuenciación es Leisten (1990), quien propone varias heurísticas constructivas para cuatro variantes del problema de Flow-shop de 2 máquinas: sin restricción de capacidad (conocido en inglés como *ilimitate buffer*), con capacidad limitada a un valor mayor que cero (*limited/finite intermediate storage*), con capacidad nula (*blocking*) y sin esperas (*no-wait*). En este trabajo se formaliza por primera vez la relación existente entre los tiempos de inicio y fin de los trabajos en las máquinas del taller en función de la capacidad de los buffers intermedios, concluyendo que el método NEH propuesto por Nawaz, Enscore, & Ham (1983) es la que proporciona mejores resultados. Nowicki (1999), generalizando para el caso de m máquinas el método propuesto por Smutnicki (1998), representa el problema usando un modelo de grafos dirigidos y desarrolla un algoritmo de Búsqueda Tabú (*Tabu Search, TS*) que se basa en las propiedades de eliminación de bloqueos para determinar vecindarios eficientes. Posteriormente, Li & Tang (2005) presentan un método de aceleración en el cálculo del makespan de una secuencia determinada y nuevas propiedades del bloqueo que aplican en el algoritmo TS propuesto por Nowicki (1999) para reducir el tamaño del vecindario. Wang et al. (2006) abordan el mismo problema del taller de flujo regular con permutación mediante un algoritmo genético híbrido HGA (*Hybrid Genetic Algorithm*) y analizan los efectos del tamaño del buffer sobre el valor del makespan. Sin embargo, Liu et al. (2008) proponen un efectivo algoritmo híbrido basado en la optimización de enjambre de partículas HPSO (*Hybrid Particle Swarm Optimization*), el cuál mejora al HGA en la mayoría de las instancias. Qian et al. (2009) fueron los primeros en aplicar el algorimo DE (*Differential Evolution*) para resolver el LBPFSP (*Limited-buffer permutation flowshop scheduling problem*), mientras que Pan et al. (2011) fueron los primeros en proponer el algoritmo CHS (*Chaotic Harmony Search*). CHS demostró ser mejor que los algoritmos HGA o HPSO en términos de calidad de la solución y robusteza. Pan et al. (2011) obtienen mejores resultados en comparación con los algoritmos HGA y HPSO con un efectivo algoritmo HDDE (*Hybrid Discrete Differential Evolution*). Demostraron que el algoritmo propuesto alcanza excelentes resultados para problemas de gran tamaño. Moslehi & Khorasanian (2014) propone un algoritmo de búsqueda de vecindad variable híbrida HVNS (*Hybrid Variable Neighborhood Search*), hibridizada con el algoritmo SA (*Simulated Anneling*). Obtiene mejores resultados en comparación con los algoritmos HGA, HPSO y CHS. Adicionalmente, desarrollan un método para reducir la complejidad de los enfoques de búsqueda local, comparándolos también con otros algoritmos del problema de Flow-shop con bloqueo. Por último, Zhao et al. (2014) presenta una PSO (Particle Swarm Optimisation)

mejorada con un término de perturbación linealmente decreciente (LDPSO). Además, también analizan los efectos del tamaño del buffer y de la probabilidad de decisión sobre el desempeño de la optimización.

En la Tabla 4.1 se recopilan los artículos ya mencionados que resuelven el problema de secuenciación de un taller de flujo con permutación y buffers limitados (LBPFSP) para *m* máquinas (excepto Leisten, 1990, quién lo resuelve para 2 máquinas). En esta recopilación se pueden comparar los procedimientos de resolución aplicados por los distintos autores a lo largo de la literatura. Destaca que todos ellos son métodos aproximados, por lo que no aportan una solución exacta al problema. Además, la mayoría son metaheurísticas, por lo que requieren un criterio de parada concreto (al contrario que las heurísticas constructivas que finalizan cuando se obtiene la secuencia completa de trabajos). Por tanto, no se tiene constancia de que se haya estudiado la resolución de este problema en concreto mediante heurísticas constructivas para un taller de *m* máquinas.

Tabla 4.1. Recopilación de artículos que resuelven el problema $F_m \mid prmu, b_i \mid C_{max}$ en la literatura.

Autores	Año	Título	Tipo método resolución	Tipo genérico de algoritmo	Tipo particular de algoritmo	Acrónimo	Criterio de parada
Leisten	1990	Flowshop sequencing problems with limited buffer	Aproximado	Heurística constructiva	Heuristic Dynamic Programming (HDP)	BPFS, BPFSE	Cuando se genera la secuencia completa
Nowicki	1999	The permutation flow shop with buffers: A tabu search	Aproximado	Metaheurística	Tabu Search	TS	Cuando se agota la memoria a largo plazo
Li & Tang	2005	A tabu search algorithm based on new block properties and speed-up method for permutation flow-shop with finite intermediate storage	Aproximado	Metaheurística	Tabu Search	TSN, TSI	Cuando la memoria a largo plazo se agota o se alcanza el número de iteraciones máximo establecido (5000)
Wang et al.	2006	An effective hybrid genetic algorithm for flow shop scheduling with limited buffers	Aproximado	Metaheurística	Hybrid Genetic Algorithm	HGA	Cuando se alcanza el número de generaciones máximo (nxm)
Liu et al.	2008	An effective hybrid PSO-based algorithm for flow shop scheduling with limited buffers	Aproximado	Metaheurística	Hybrid Particle Swarm Optimization	HPSO	Cuando el valor de la mejor partícula encontrada se repita L veces consecutivas (L=5)
Qian et al.	2009	An effective hybrid DE-based algorithm for flow shop scheduling with limited buffers	Aproximado	Metaheurística	Hybrid Differential Evolution	HDE	Cuando se alcanza el número de generaciones máximo (nxm)
Pan et al.	2011	A chaotic harmony search algorithm for the flow shop scheduling problem with limited buffers	Aproximado	Metaheurística	Chaotic Harmony Search	CHS	Cuando se alcanza el tiempo computacional máximo ($T=10xnxm$) ms

Pan et al.	2011	An effective hybrid discrete differential evolution algorithm for the flow shop scheduling with intermediate buffers	Aproximado	Metaheurística	Hybrid Discrete Differential Evolution	HDDE	Cuando se alcanza el tiempo computacional máximo (T=30x*n*x*m*) ms
Moslehi & Khorasanian	2014	A hybrid variable neighbourhood search algorithm for solving the limited-buffer permutation flow shop scheduling problem with the makespan criterion	Aproximado	Metaheurística	Hybrid Variable Neighborhood Search	HVNS	Cuando se alcanza el número de iteraciones máximo (1.8x105)
Zhao et al.	2014	An improved particle swarm optimisation with a linearly decreasing disturbance term for flow shop scheduling with limited buffers	Aproximado	Metaheurística	Linearly Decreasing disturbance term Particle Swarm Optimization	LDPSO	Los parámetros decrecen linealmente hasta que converge, o en su defecto, cuando alcanza un número máximo de iteraciones

5 ANÁLISIS DE LA METODOLOGÍA

Para la resolución del problema objeto de estudio, se emplean métodos de optimización para acelerar la búsqueda de soluciones. En este capítulo se comentarán algunos de los métodos generales empleados en la literatura para la resolución de problemas de programación de la producción similares, razonando cuáles se tendrán como base a la hora de resolver el problema.

5.1 Procedimientos generales de resolución

5.1.1 Métodos exactos

Los métodos exactos proporcionan la solución óptima del problema, ya que evalúan todas las combinaciones posibles del modelo, explícita o implícitamente. Se pueden clasificar en dos tipos:

- Algoritmos constructivos exactos: resuelven problemas polinomiales, de clase *P*, evaluando todas las soluciones del modelo para obtener la óptima de ellas. Pertenecen a este grupo, entre otros, los conocidos algoritmos de Johnson, Lawler, Moore, y algunas reglas de despacho que especifican el orden de procesado de los trabajos, para problemas de programación muy sencillos. Algunos ejemplos de reglas de despacho son la conocida FIFO (*First In First Out*), donde el primer trabajo que llega es el primero en ser procesado (obtiene el óptimo para el problema $1 \mid r_j \mid C_{max}$), o la regla SPT (*Shortest Processing Time first*), donde se procesan los trabajos ordenados de menor a mayor tiempo de proceso (resuelve de forma óptima los problemas $1 \mid\mid \sum C_j$ y $P_m \mid\mid \sum C_j$).

- Algoritmos enumerativos: aunque no evalúan todas las soluciones explícitamente, garantizan la obtención de la solución óptima. Las soluciones que no son evaluadas explícitamente se evalúan implícitamente, ya que existen evidencias que indican que la evaluación de esta implicaría un valor de la función objetivo igual o superior al de otra solución ya evaluada. Al ser de tipo no polinomial, NP, sólo dan soluciones óptimas para problemas con instancias pequeñas. Pertenecen a este grupo la programación matemática (en concreto los modelos de programación lineal entera mixta), el método de ramificación y acotación conocido como *Branch and Bound*, y la programación dinámica.

5.1.2 Métodos aproximados

Debido a la inviabilidad de hallar una solución óptima mediante un método exacto en un tiempo razonable, existen los métodos aproximados. Con ellos no existe certeza de que la solución alcanzada sea la óptima, ya que no se evalúa la totalidad del conjunto de soluciones posibles, sino parte de él, consiguiendo una buena solución en un tiempo aceptable. Entre los métodos de resolución aproximados se encuentran:

- Heurísticas: "Las heurísticas son procedimientos simples, a menudo guiados por el sentido común, que están destinados a proporcionar soluciones buenas pero no necesariamente óptimas a problemas difíciles, de manera fácil y con rapidez." (Zanakis & Evans, 1981). Son de gran utilidad para problemas que no requieren una solución óptima o en los que no se pueda aplicar métodos exactos por otros motivos. También suelen emplearse como paso previo a la aplicación de otro algoritmo o cuando los datos del problema tienen una alta variabilidad o no son fiables.

  Una de las heurísticas más usadas en la actualidad por su efectividad en la resolución del problema de *flowshop* es la NEH propuesta por (Nawaz et al., 1983). Se trata de una heurística constructiva que consiste en construir la solución en un número finito de pasos, insertando uno a uno los trabajos aún sin secuenciar en la posición de la secuencia parcial que menor valor de la función objetivo proporcione.

- Metaheurísticas: se emplean cuando se abordan problemas complejos para los que las heurísticas resultan ineficientes e inefectivas. Son procedimientos adecuados para realizar una búsqueda de soluciones eficaz y efectiva, encontrando el equilibrio adecuado entre intensificación y diversificación.

  Estas dos características de la búsqueda son muy importantes. Por un lado, la intensificación está relacionada con la explotación del espacio en la búsqueda local, centrándose en las que aparentan ser más prometedoras, consiguiendo así soluciones de mayor calidad. Por otro lado, la diversificación, consigue una apropiada exploración en la búsqueda global, abarcando un espacio mayor de soluciones variadas mediante la búsqueda en regiones que contrastan fuertemente con las regiones exploradas hasta el momento, no aleatoriamente, sino teniendo en consideración el proceso de búsqueda anterior.

  Es fundamental buscar el equilibrio correcto entre ambas características en el proceso de diseño de una metaheurística efectiva, ya que una falta de intensificación provoca que la exploración sea demasiado larga, y una falta de diversificación produce una rápida convergencia a mínimos locales.

  Los métodos metaheurísticos pueden presentar los siguientes enfoques:

  - Enfoque constructivo, basado en generar la solución paso a paso, tomando una decisión cada vez que se secuencia un trabajo, disminuyendo así el tamaño del problema en cada decisión al descartar posibles soluciones.
  - Enfoque iterativo, que realiza una búsqueda local en cada iteración, estudiando las soluciones del entorno de la solución que realiza. Este entorno se denomina vecindario, y se corresponde con el conjunto de soluciones formadas a partir de una modificación de una solución. Uno de los métodos más conocidos de este tipo de enfoque es la conocida búsqueda Tabú (*Tabú Search*).
  - Enfoque evolutivo, inspirado en mecanismos biológicos como son la selección y evolución natural, genera nuevas soluciones seleccionando, cruzando, o mutando las soluciones existentes en un conjunto denominado población inicial, obteniendo así soluciones de mayor calidad "evolucionadas" a partir de las existentes.

5.2 Metodología seleccionada

El problema en cuestión es de tipo combinatorio, por lo que teóricamente evaluando todas las combinaciones posibles que componen el conjunto finito de soluciones sería suficiente para determinar cuál de ellas es la óptima para el objetivo fijado. Cuando se trata de problemas complejos (*NP-hard*) o de gran tamaño cuyo conjunto finito de soluciones tiene un tamaño elevado, no es posible determinar el programa óptimo para este problema en un tiempo admisible, no teniendo la certeza de que el programa dado como solución sea mejor que cualquier otro programa admisible. Por tanto, puesto que no es posible encontrar la solución óptima para el problema $F_m \mid prmu, b_i \mid C_{max}$ en un tiempo razonable, debido a su complejidad y número elevado de programas factibles, se considera que lo más adecuado es emplear métodos aproximados para su resolución. Estos métodos hallarán una secuencia que contenga la combinación de trabajos que proporcione el mínimo valor posible del *makespan* en un tiempo admisible. No será posible conocer el programa óptimo, ni por tanto, cuánto se aproxima el valor del *makespan* de la secuencia solución del algoritmo al valor óptimo.

Es muy común hacer uso de algoritmos en los que se codifica dicho procedimiento con un número finito de pasos y se ejecuta mediante un ordenador. Se diferencia entre algoritmos exactos, que alcanzan el óptimo del problema, y algoritmos aproximados, que proporcionan una solución aproximada. En concreto, en este documento se proponen 6 algoritmos aproximados basados en métodos existentes en la literatura, como la Búsqueda tabú, las heurísticas NEH, MCH, o PF. También se usará, como secuencia inicial de los algoritmos, una de las reglas de despacho mencionadas: *Longest Processing Time first* (LPT), que consiste en secuenciar los trabajos ordenados de mayor a menor tiempo de proceso total (suma de los tiempos de proceso de cada trabajo en todas las máquinas). A continuación, se describen los métodos principales en los que se basarán los algoritmos propuestos.

5.2.1 Heurística NEH

En cuanto a la heurística constructiva NEH, a pesar de que rara vez se consigue obtener el óptimo con este procedimiento, proporciona muy buenos resultados para el problema $F_m|prmu|C_{max}$, y aporta como ventajas su eficacia, sencillez y rapidez en la búsqueda de soluciones.

Su procedimiento consiste, en primer lugar, en construir una secuencia inicial según la regla LPT. Para ello se suman para cada uno de los trabajos, sus tiempos de proceso en todas las máquinas, y se ordenan de mayor a menor. A continuación, comienza la fase de inserción, donde progresivamente se va construyendo la secuencia, incorporando uno a uno los trabajos en el orden previamente marcado por la regla LPT. Para ello se crea inicialmente una secuencia parcial compuesta por la mejor combinación entre los primeros dos trabajos, y se van insertando sucesivamente los trabajos de la secuencia inicial en la posición de la secuencia parcial que proporcione un mejor valor de la función objetivo, hasta secuenciar todos los trabajos.

Este procedimiento se refleja en el pseudocódigo de la Figura 5.1.

Entrada: datos de la instancia
Salida: Π, C_{max}
Inicio
  $P_{\gamma_j} = \sum_{i=1}^{m} p_{i\gamma_j}$;
  $\Pi = \emptyset$, $J = \{\gamma_1, \ldots, \gamma_n\}$ verificándose que $P_{\gamma_1} \geq P_{\gamma_2} \geq \ldots P_{\gamma_n}$;
  $\Pi^{(1)} = \{\gamma_1\}$;
  Para $k = 2$ **hasta** n **hacer**
    Insertar el trabajo γ_k en la posición de $\Pi^{(k)}$ que conlleve el menor C_{max};
  Fin
  Devolver Π, C_{max}
Fin

Figura 5.1. Pseudocódigo algoritmo NEH.

Es cierto, como muestran Companys Pascual, R. & Ribas Vila (2012), que este procedimiento no siempre mejora la calidad de la secuencia inicial. Por tanto, es recomendable evaluar siempre la secuencia inicial antes para compararla con la solución obtenida, y así mantener la mejor de ellas, lo cual se tendrá en cuenta posteriormente en el desarrollo de los algoritmos.

5.2.2 Búsqueda Tabú

La búsqueda tabú (*tabu search*) es una técnica de búsqueda local basada en la memoria que proporciona muy buenos resultados en problemas de optimización debido a su versatilidad. Este método presenta un enfoque iterativo, puesto que consiste en prohibir los peores movimientos durante un número de iteraciones, consiguiendo así evitar caer en óptimos locales. Se considera muy eficiente en la resolución de problemas combinatorios complejos, como es el conocido problema del viajero, problemas de rutas y envíos de productos, y por supuesto, problemas de secuenciación.

La búsqueda tabú, como cualquier otra búsqueda local, parte de una solución inicial y genera otra nueva solución a partir de ella. Para ello, transforma la solución inicial mediante la realización de movimientos, como por ejemplo, retirar de la secuencia un trabajo ya secuenciado y colocarlo en otra posición distinta, o simplemente intercambiar dos de los trabajos ya secuenciados. De esta forma se evita caer en un óptimo local

y poder alcanzar el óptimo global del problema (Véase Figura 5.2).

Esta metaheurística se caracteriza por su lista tabú. Consiste en una lista que contiene un conjunto de movimientos tabú (prohibidos), que permiten evitar probar soluciones poco diversas. El tamaño de la lista define el tamaño de este conjunto de movimientos. En el caso en que el número de movimientos contenidos en la lista alcance el tamaño total de la lista tabú, antes de añadir un nuevo movimiento a la lista se debe eliminar uno de los que ya contiene. Para decidir qué movimiento se elimina, se establecen criterios de aspiración por defecto, donde se elimina el más antiguo (el movimiento que lleve más tiempo en la lista), o criterios de aspiración por objetivo, donde solo se admite el movimiento si produce una solución mejor a la obtenida hasta el momento. Este último suele ser el criterio mayoritariamente utilizado.

La Figura 5.3 muestra el diagrama de flujo de la búsqueda tabú, donde *MS* representa la mejor solución hasta el momento, y *AS* la actual solución durante el proceso de búsqueda.

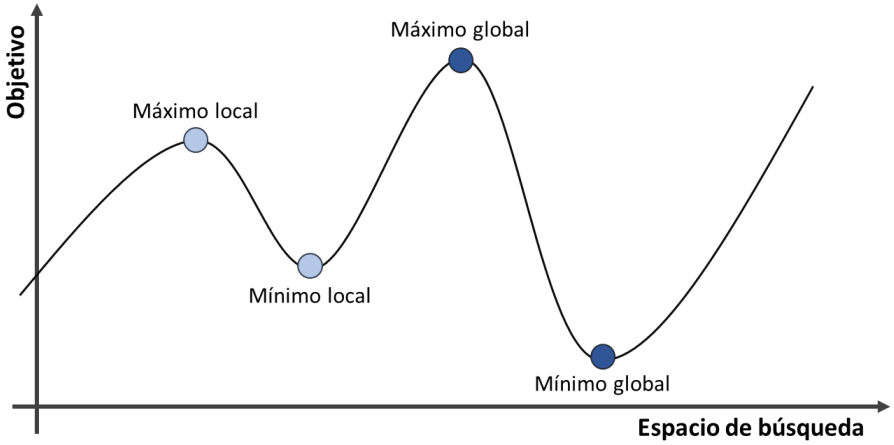

Figura 5.2. Representación mínimos y máximos (locales y globales).

Es importante encontrar el equilibrio correcto entre intensificación y diversificación, como se menciona en el apartado 5.1.2. En la búsqueda tabú generalmente se presenta un equilibrio estático, ya que cuando mayor sea el tamaño de la lista, se producirá una mayor diversificación, pero una menor intensificación.

5.2.3 Heurística MCH

La heurística constructiva basada en la memoria MCH (*Memory-based Constructive Heuristic*) fue propuesta por Fernandez-Viagas et al. (2018) para talleres de flujo regular híbridos (varias etapas con varias máquinas). Esta heurística está basada en la heurística NEH y en el concepto inverso de la metaheurística de búsqueda tabú. Se caracteriza porque en cada paso de construcción de la secuencia, es decir, tras cada inserción de un nuevo trabajo en la secuencia parcial, se vuelven a evaluar los movimientos más prometedores resultantes en los pasos de construcción anteriores (opuesto a la búsqueda tabú, los trabajos que se introducen en la lista tabú dejan de ser movimientos prohibidos, y pasan a ser movimientos prometedores, pasando a denominarse la lista prometedora (*promise*). Así, se mantienen en una lista los movimientos más prometedores (que no han sido seleccionados como mejor opción pero proporcionaban un buen valor de makespan) para volver a repetirlos en futuras iteraciones de la búsqueda, consiguiendo evitar que se desechen combinaciones de trabajos que a priori no eran la mejor opción, pero al considerarlas en iteraciones posteriores del algoritmo puedan proporcionar mejores resultados. Además, esta búsqueda local ayuda a escapar de óptimos locales ampliando la búsqueda, lo cual también es ventajoso.

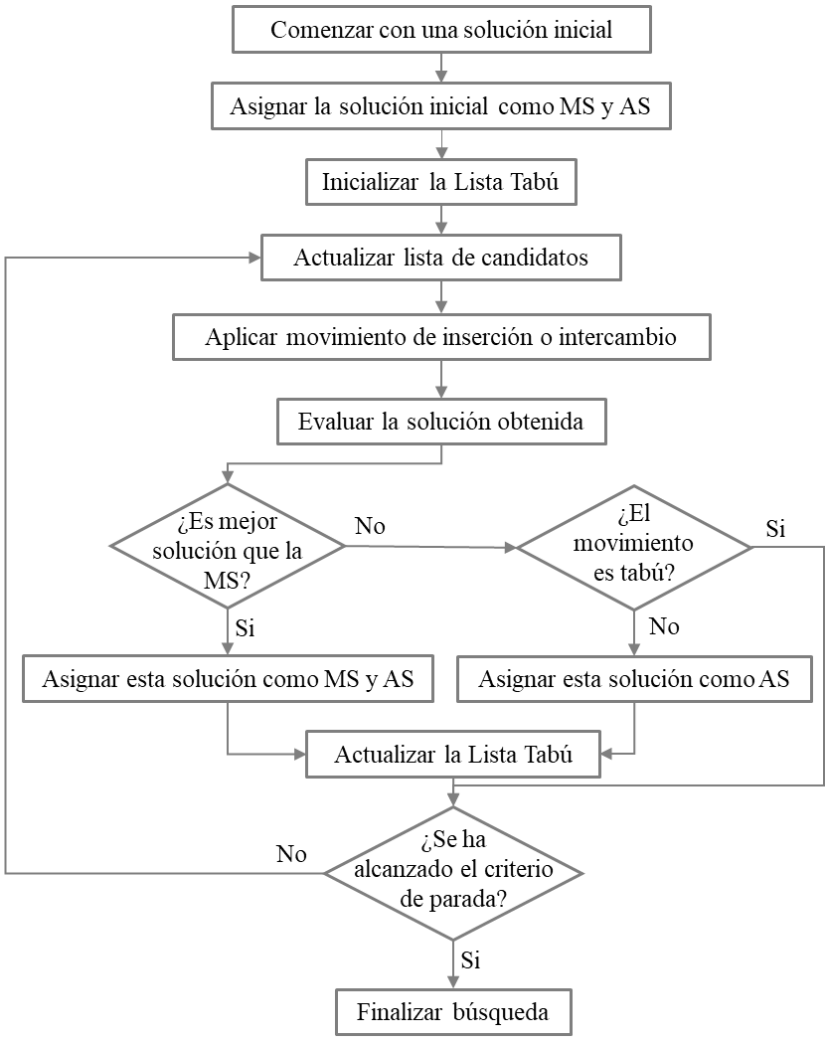

Figura 5.3. Diagrama de flujo Búsqueda Tabú. [Fuente: Adaptación al español de (Abiri et al., 2009)]

5.2.4 Heurística PF

Pan & Wang (2012) presentan una heurística constructiva para resolver el problema de flowshop con bloqueo, basada en el enfoque PF (*Profile fitting*) de McCormick et al. (1989), que trata de secuenciar los trabajos en función de cual proporcione un menor valor de la suma del tiempo ocioso y del tiempo de bloqueo en las máquinas.

Esta heurística PF consiste en ir construyendo la secuencia paso a paso, añadiendo en la última posición de la secuencia parcial el trabajo que menor tiempo ocioso y tiempo de bloqueo proporcione de entre todos los trabajos que aún no se han secuenciado. Inicialmente, la secuencia parcial se compone de un trabajo, que se corresponde con el trabajo que tenga menor tiempo de proceso total, como proponen (Ribas et al., 2011) y (Ronconi, 2004).

En la Figura 5.4 se representa el procedimiento seguido por la heurística PF mediante un diagrama de flujo. Adicionalmente, en la Figura 5.5, se presenta una adaptación del pseudocódigo propuesto por (Pan & Wang, 2012).

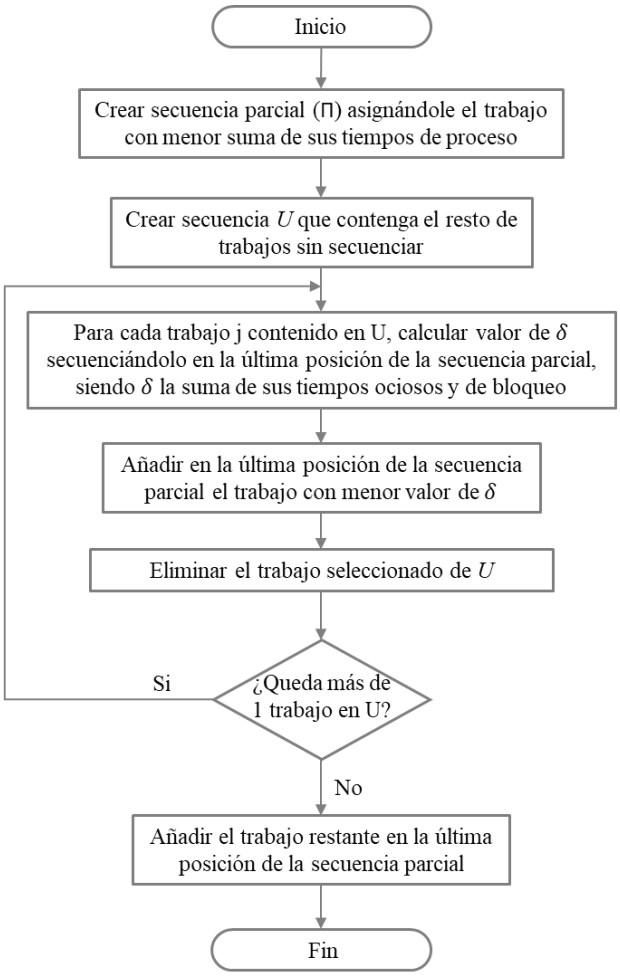

Figura 5.4. Diagrama de flujo heurística PF.

Entrada: datos de la instancia
Salida: Π, C_{max}
Inicio

Seleccionar el trabajo con menor suma de sus tiempos de proceso como el primer trabajo de la secuencia parcial, y denotar la secuencia parcial como $\Pi^{(1)} = \{\gamma_1\}$;

$U = \{\gamma_1,\gamma_2,\ldots,\gamma_n\}$ - $\Pi^{(1)}$;
Para $k = 1$ **hasta** $n - 1$ **hacer**

Calcular el tiempo de terminación $c_{k,i}$, $i = 1,2,\ldots,m$, para el último trabajo γ_k en la secuencia parcial $\Pi^{(k)} = \{\gamma_1,\gamma_2,\ldots,\gamma_k\}$;

Para cada trabajo $j \in U$, convertirlo en el *(k+1)-ésimo* trabajo de $\Pi^{(k)}$, calcular su tiempo de terminación $c_{(k+1),i}$, para $i = 1,2,\ldots,m$, y la suma de los tiempos ociosos y de bloqueo $\delta_{j,k} = \sum_{i=1}^{m}\left(c_{(k+1),i} - c_{k,i} - p_{j,i}\right)$;

Seleccionar el trabajo que resulte con menor valor de $\delta_{j,k}$ siendo el *(k+1)-ésimo* trabajo de $\Pi^{(k)}$, y eliminar el trabajo seleccionado de U;

Fin

El único trabajo restante en U se añade a $\Pi^{(n-1)}$ convirtiéndose en el *n-ésimo* trabajo de Π;
Calcular valor C_{max} para Π;
Devolver Π, C_{max}
Fin

Figura 5.5. Pseudocódigo algoritmo PF. [Fuente: Adaptación de (Pan & Wang, 2012)]

5.2.5 Heurística PF-NEH(*x*)

La heurística NEH ha sido reconocida como el método con mejor desempeño en la resolución del problema de secuenciación en talleres de flujo con permutación con el objetivo de makespan. Ronconi (2004) propone la heurística PFE, que combina esta heurística NEH con la PF. En concreto, usa el método PF para hallar la solución inicial de la NEH, que originalmente venía dada por la regla de despacho LPT, y así consigue mejores resultados para el problema de flowshop con bloqueo que los generados por la NEH original.

Basándose en la heurística PFE, Pan & Wang (2012) proponen la heurística PF-NEH, una combinación distinta de la heurística PF con la NEH. Sólo los últimos λ trabajos de la secuencia generada por la heurística PF pasan a ser secuenciados por el procedimiento NEH, que parte de la secuencia parcial construida por la PF que contiene los $n-\lambda$ trabajos restantes.

Así consigue acelerar la construcción de la secuencia, ya que realiza un menor número de evaluaciones, incurriendo en un menor tiempo de cómputo. Cuando $\lambda = n-1$, la heurística PF-NEH equivale a la PFE, mientras que si $\lambda = 0$, equivale a la PF.

Adicionalmente, Pan & Wang (2012) concluye que la elección del primer trabajo de la secuencia afecta al tiempo ocioso inicial de las máquinas y al tiempo de comienzo de los siguientes trabajos. Por ello, propone aplicar la heurística varias veces, variando el trabajo que se selecciona como primer trabajo de la secuencia, obteniendo así varias secuencias, y quedándose como resultado final con la mejor de ellas. Denotan esta versión como PF-NEH(*x*), donde *x* es el número de secuencias generadas. El pseudocódigo de este método se muestra en la Figura 5.6.

Entrada: datos de la instancia, parámetro *x*
Salida: Π, C_{max}
Inicio
  Generar una secuencia de trabajos $\alpha = [\alpha_1, \alpha_2, ..., \alpha_n]$ que contenga los trabajos ordenados de menor a mayor tiempo de proceso total;
  Para *l* = 1 **hasta** *x* **hacer** // Generar *x* secuencias
    Seleccionar el trabajo α_l como el primer trabajo de la secuencia parcial, y generar una secuencia $\beta = [\beta_1, \beta_2, ..., \beta_n]$ usando la heurística PF;
    $\Pi^{(l)} = [\beta_1, \beta_2, ..., \beta_n]$;
    Para *k* = n – λ + 1 **hasta** *n* **hacer** // Procedimiento NEH
      Escoger el trabajo β_k de β y probarlo en todas las *k* posibles posiciones de $\Pi^{(l)}$;
      Insertar el trabajo β_k en la posición de $\Pi^{(l)}$, que genere un menor valor de la función objetivo.
    Fin
  Fin
  Calcular valor C_{max} para Π;
  Devolver Π con menor valor de C_{max}
Fin

Figura 5.6. Pseudocódigo algoritmo PF-NEH(*x*). [Fuente: Adaptación de (Pan & Wang, 2012)]

6 APLICACIÓN DE LA METODOLOGÍA

Con la finalidad de encontrar una solución al problema abordado se proponen diversos métodos aproximados. En concreto, se proponen 6 heurísticas constructivas cuyas características se explicarán detalladamente en esta sección. Adicionalmente, se incluye una búsqueda local reducida que se aplicará a cada uno de los métodos propuestos para aumentar la calidad de la solución obtenida.

6.1 Heurística 1: NEH_FO

El primero de los métodos de resolución propuestos consiste en una variante de la heurística constructiva NEH, con el objetivo de mejorar la calidad de la solución obtenida manteniendo el tiempo computacional aproximadamente constante.

Tratando de adaptar la heurística al problema en cuestión, se introduce un cambio en la función objetivo que usa el algoritmo para la búsqueda de soluciones. Esta función originalmente se corresponde con el *makespan* o tiempo total de finalización de las tareas C_{max}. Teniendo en cuenta el efecto que produce la existencia de buffers en el taller, se considera adecuado considerar el papel que juegan el tiempo ocioso y el tiempo de bloqueo de las máquinas. Por ello, se introducen ambos conceptos en la función objetivo (FO), de forma que el valor que determine en cada paso de construcción de la secuencia en qué posición insertar el trabajo sea el valor del *makespan*, pero además, en caso de empate, se valoran los tiempos ociosos y de bloqueo, como se muestra en la ecuación (2.1).

$$FO = C_{\max} + \frac{idletime + blockingtime}{Q} \qquad (2.1)$$

Con esta función se tiene en cuenta principalmente el objetivo del problema (C_{max}), pero además, en caso de empate en cuanto al valor del *makespan* de dos soluciones distintas (el mínimo valor de *makespan* se dé simultáneamente al insertar el trabajo en distintas posiciones), se priorizará la opción que minimice tanto el tiempo ocioso (*idle-time*) como el tiempo de bloqueo (*blocking-time*). Esto se consigue penalizando la función objetivo de manera que su valor de *makespan* se vea incrementado por la suma de dichos valores de tiempo (divididos por *Q*, un número lo suficientemente grande para que $\frac{idle\ time + blocking\ time}{Q} < 1$). Para valores reales de tiempos de proceso de los trabajos, se considera suficiente que Q tome un valor de 1000000 unidades.

A continuación, en la Figura 6.1 se representa el diagrama de flujo de esta heurística para ayudar a comprender su funcionamiento. Se puede comprobar que el pseudocódigo del método propuesto, mostrado en la Figura 6.2, es idéntico al de la heurística constructiva NEH con la particularidad de que la función objetivo empleada es la mencionada anteriormente (FO).

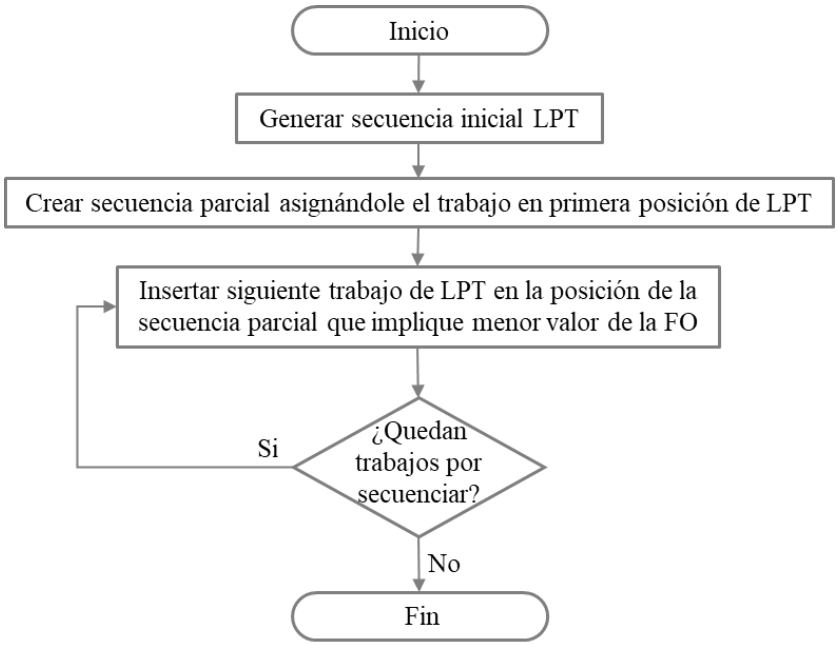

Figura 6.1. Diagrama de flujo heurística NEH_FO

Entrada: datos de la instancia
Salida: Π, C_{max}
Inicio
$P_{\gamma_j} = \sum_{i=1}^{m} p_{i\gamma_j}$;
Π = ∅, $J = \{ \gamma_1, \ldots, \gamma_n \}$ verificándose que $P_{\gamma_1} \geq P_{\gamma_2} \geq \ldots P_{\gamma_n}$;
$\Pi^{(1)} = \{ \gamma_1 \}$;
Para $k = 2$ **hasta** n **hacer**
Insertar el trabajo γ_k en la posición de $\Pi^{(k)}$ que conlleve el menor *FO*;
Fin
Calcular valor C_{max} para Π;
Devolver Π, C_{max}
Fin

Figura 6.2. Pseudocódigo algoritmo NEH_FO.

6.2 Heurística 2: MCH*cad*

El segundo método propuesto está basado en la heurística MCH. En concreto, se trata también de una heurística constructiva basada en memoria, en donde se utiliza la ecuación del apartado anterior (Ecuación 2.1) para evaluar cada uno de los candidatos. Adicionalmente, se le añade caducidad a la lista de movimientos prometedores, es decir, las combinaciones de trabajos contenidas en la lista tienen un tiempo de permanencia en ella, y serán eliminadas de la lista cuando se hayan evaluado un determinado número de veces sin éxito. Ese número de veces viene determinado por un parámetro de entrada al algoritmo, denominado *cad*.

La capacidad de la lista, es decir, el número de movimientos que se almacenan en ella viene determinado por el tamaño de la lista, que se define con el parámetro *tam*. Cabe mencionar que en esta heurística, el tamaño de la lista toma un valor fijo para todas las iteraciones, a diferencia de la MCH, donde la lista ampliaba su tamaño dinámicamente en cada paso de construcción de la secuencia.

Esta heurística inicialmente requiere de una solución inicial, la cual será dada por la regla LPT. El trabajo en primera posición de la secuencia inicial formará la secuencia parcial a la que se irán añadiendo uno a uno los siguientes trabajos de la secuencia inicial. El siguiente trabajo se insertará en cada una de las posiciones de la secuencia parcial, evaluando el valor resultante de FO para cada inserción, eligiendo la secuencia que proporcione un menor valor, y guardando en una lista los movimientos (combinaciones de dos trabajos) descartados que resulten prometedores. Para determinar si un movimiento es prometedor o no, se emplea un indicador que mide el incremento (*I*) del valor de FO con respecto al mejor valor de FO encontrado, como se muestra en la siguiente función:

$$I = FO_{movimiento} - FO_{mejor} \qquad (2.2)$$

Cada vez que se inserta un nuevo trabajo en la secuencia parcial, se vuelven a repetir las combinaciones prometedoras de la lista. Para ello se aplica la búsqueda local que genera vecinos de la secuencia parcial repitiendo los movimientos de la lista, es decir, reinsertando un trabajo en una posición distinta a la actual, y actualizando la secuencia parcial cada vez que se muestre una mejora respecto al objetivo marcado: minimización de FO. El movimiento que resulte mejor, se elimina de la lista de movimientos prometedores, intercambiandolo por el movimiento deshecho. Finalmente, la búsqueda termina una vez que todos los trabajos que conforman la secuencia inicial LPT han sido secuenciados, obteniendo una secuencia completa como solución del problema. Este procedimiento queda reflejado en el diagrama de flujo de la Figura 6.4, que muestra cómo se construye la secuencia paso a paso, y la dinámica seguida a la hora de introducir movimientos en la lista.

En la Figura 6.3 se muestra la estructura de la lista que contiene los movimientos prometedores. Cada fila de la matriz se corresponde con un movimiento (combinación del trabajo a reinsertar y el trabajo tras el cual se debe insertar en la secuencia), y lleva asignado el valor de la desviación respecto al mejor valor de FO (*I*) y un contador *c* que indica el número de veces que se ha evaluado.

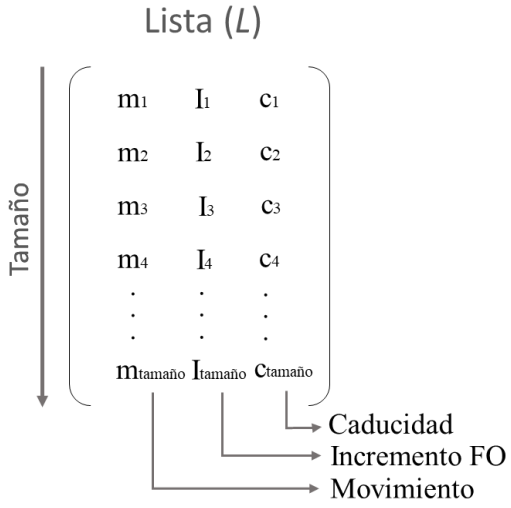

Figura 6.3. Estructura de la lista de movimientos prometedores.

Como la lista tiene una capacidad limitada, al insertar un nuevo movimiento cuando la lista esté completa, se usa el valor de *I* para determinar su inclusión o no en la lista. Se compara el valor de *I* del movimiento candidato, con los ya contenidos en la lista, y si es menor que el máximo de la lista se introduce, reemplazando al movimiento con el máximo incremento. Además, cuando un movimiento alcance su caducidad, es decir, el valor del contador *c* sea igual al parámetro *cad*, se elimina de la lista.

Adicionalmente, el pseudocódigo del algoritmo se presenta en la Figura 6.5.

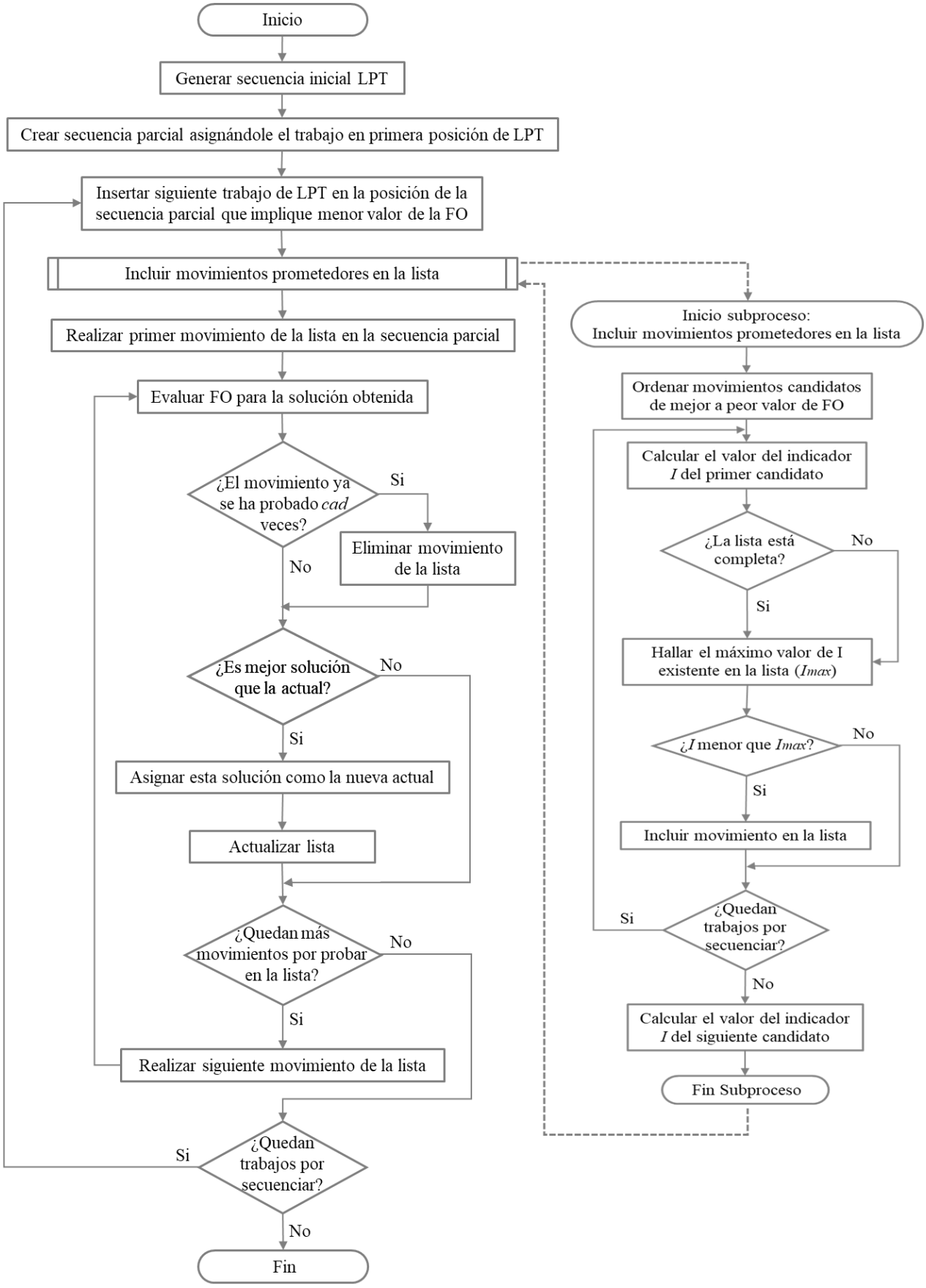

Figura 6.4. Diagrama de flujo heurística MCH*cad*.

Entrada: datos de la instancia, parámetros *tam* y *cad*
Salida: Π_b, C_{max}
Inicio
  $P_{\gamma_j} = \sum_{i=1}^{m} p_{i\gamma_j}$;
  $J = \{\gamma_1, \ldots, \gamma_n\}$ verificándose que $P_{\gamma_1} \geq P_{\gamma_2} \geq \ldots P_{\gamma_n}$; //Secuencia LPT
  Asignar a *Obj* el valor de la función objetivo para *J;*
  $\Pi^{(1)} = \{\gamma_1\}$;
  $L = \emptyset$; //Lista que contiene los movimientos prometedores, inicialmente vacía.
  $c_1 \ldots c_{tam} = 0$; *//Contador nº veces que se evalúa un trabajo de la lista, inicialmente 0.*
  Para $k = 2$ **hasta** *n* **hacer**
    Insertar el trabajo γ_k en todas las posibles posiciones *j* de $\Pi^{(k-1)}$, con $j \in \{1, \ldots, k\}$;
    $\Pi^{(k)}$ = secuencia obtenida insertando γ_k en la posición de $\Pi^{(k-1)}$ con menor valor de la FO (Obj_b);
    $CL_x^{(k)}$ (con $x \in \{1, \ldots, k\}$) = trabajo anterior a γ_k , tras evaluar γ_k en la posición *x*. Asignar a Obj_{CLx} tal valor de la función objetivo;
    Para $x = 2$ **hasta** *k* **hacer**
      $I_{CLx} = (Obj_{CLx} - Obj_b)$;
      Si *ocupación_L* < *tam* **entonces**
        Asignar movimiento ($CL_x^{(k)}$, γ_k , I_{CLx}, $c = 0$) a *L*;
      Fin
      Por el contrario
        I_{max} = Valor del mayor incremento (*I*) que contiene *L*;
        Si $I_{CLx} < I_{max}$ **entonces**
          Reemplazar el movimiento contenido en *L* correspondiente al I_{max} por ($CL_x^{(k)}$, γ_k , I_{CLx}, *c = 0*);
        Fin
      Fin
    Fin
    Si $k > 2$ **entonces**
      Para $h = 0$ **hasta** *tam* **hacer**
        $\Pi'^{(k)}$ = permutación obtenida realizando el movimiento *h* de la lista *L* en $\Pi^{(k)}$. Asignar a *Obj*' su valor de la función objetivo;
        Incrementar en una unidad el contador *c* del movimiento *h* en la lista *L*.
        Si $c_h = cad$ **entonces**
          $c_h = 0$;
          Eliminar movimiento *h* de la lista *L*;
        Fin
        Si *Obj*' < Obj_b **entonces**
          $\Pi_b^{(k)} = \Pi'^{(k)}$;
          $Obj_b = Obj'$;
          Actualizar *L* (eliminar movimiento realizado de la lista e incluir el recién deshecho en su lugar);
          Restablecer c_h a 0;
        Fin
      Fin
    Fin
  Fin
  Si *Obj* < Obj_b **entonces**
    $\Pi_b = J$;
  Fin
  Calcular valor C_{max} para Π_b;
  Devolver Π_b, C_{max}
Fin

Figura 6.5. Pseudocódigo algoritmo MCH*cad*.

6.3 Heurística 3: NEH_FO_RED

El siguiente de los algoritmos propuestos se basa en reducir el número de inserciones de trabajos que se prueban al construir la secuencia NEH.

Como se menciona en el apartado 5.2.1, la heurística NEH consiste en insertar un nuevo trabajo en cada paso de construcción de la secuencia. Para determinar cuál es la mejor posición donde insertarlo, se evalúa el makespan de la secuencia parcial insertando el mismo trabajo en todas las posibles posiciones de la secuencia. Una vez evaluados, se comparan y se elige la mejor posición (la posición donde se consiga un menor makespan). Teniendo en cuenta esto, se ha realizado un análisis para conocer cuáles eran las posiciones en las que más se insertaban los trabajos según la heurística NEH original. Se ha comprobado que los trabajos suelen insertarse mayormente al principio o al final de la secuencia, y con menor frecuencia en las posiciones centrales (véase Figura 6.6. en donde se presenta en el eje horizontal la posición relativa donde ha sido insertado el trabajo, teniendo en cuenta que 1 es la última posición de la secuencia parcial y 0 la primera). Además, cuando se empieza a construir la secuencia, la secuencia parcial tiene muy pocos trabajos y solo se puede insertar al principio o al final, por lo que las posiciones en las que más se insertan los trabajos suelen ser la primera o la última.

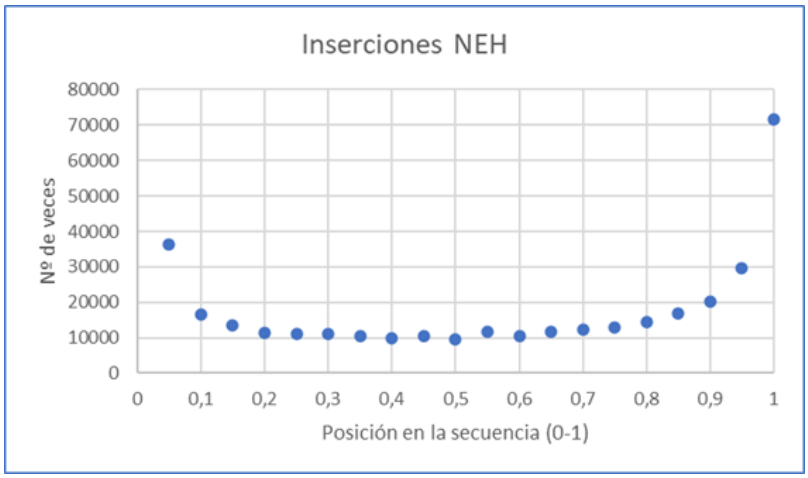

Figura 6.6. Número de inserciones según posición de la secuencia.

Conociendo esto, se propone una variante del primer método propuesto (véase apartado 6.1) que consiste en reducir de forma variable el número de inserciones de trabajos, y así conseguir disminuir el tiempo computacional, manteniendo la mejora de la función objetivo. Así se pretenden obtener mejores resultados que con el método NEH original, no sólo en calidad, sino también en tiempo.

Para reducir el número de inserciones, se define un intervalo, donde la posición 0 se corresponde con la primera posición, y la posición 1 se corresponde con la última. El intervalo que se reduce se define como *R*, y como puede verse en el ejemplo de la Figura 6.7, si el intervalo reducido es 0.5 (*R* = 0.5) se reduce la mitad, no se inserta ningún trabajo en las posiciones del centro, sino solo en las posiciones entre (0, 0.25) y (0.75, 1). Del mismo modo, si el intervalo es *R* = 0.6, solo se insertarán trabajos en las posiciones entre (0, 0.2) y (0.8, 1).

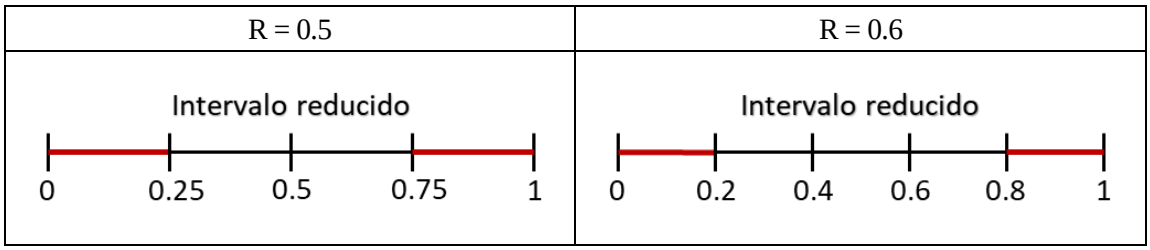

Figura 6.7. Ejemplo intervalos reducidos R

Esta reducción se produce de forma variable, de forma que en el primer paso de construcción de la secuencia, el intervalo reducido es 0, mientras que en el último paso, el intervalo reducido es *R*. Además, se comienzan a reducir las inserciones desde el centro de la secuencia hacia los extremos, de la manera que se muestra en el diagrama de flujo de la Figura 6.8. Cabe destacar que en cada paso de construcción de la secuencia, las posiciones en las que se insertan los trabajos quedan limitadas a las contenidas en el intervalo $\left[0,\left\lfloor Lim_{\text{inf erior}} * k \right\rfloor\right] \cup \left[\left\lfloor Lim_{\text{superior}} * k \right\rfloor, k\right]$, donde *k* es el número de trabajos que contiene la secuencia parcial en cada paso de construcción. Como se aprecia, las posiciones son números enteros, por lo que se truncan los valores de los intervalos al entero inferior.

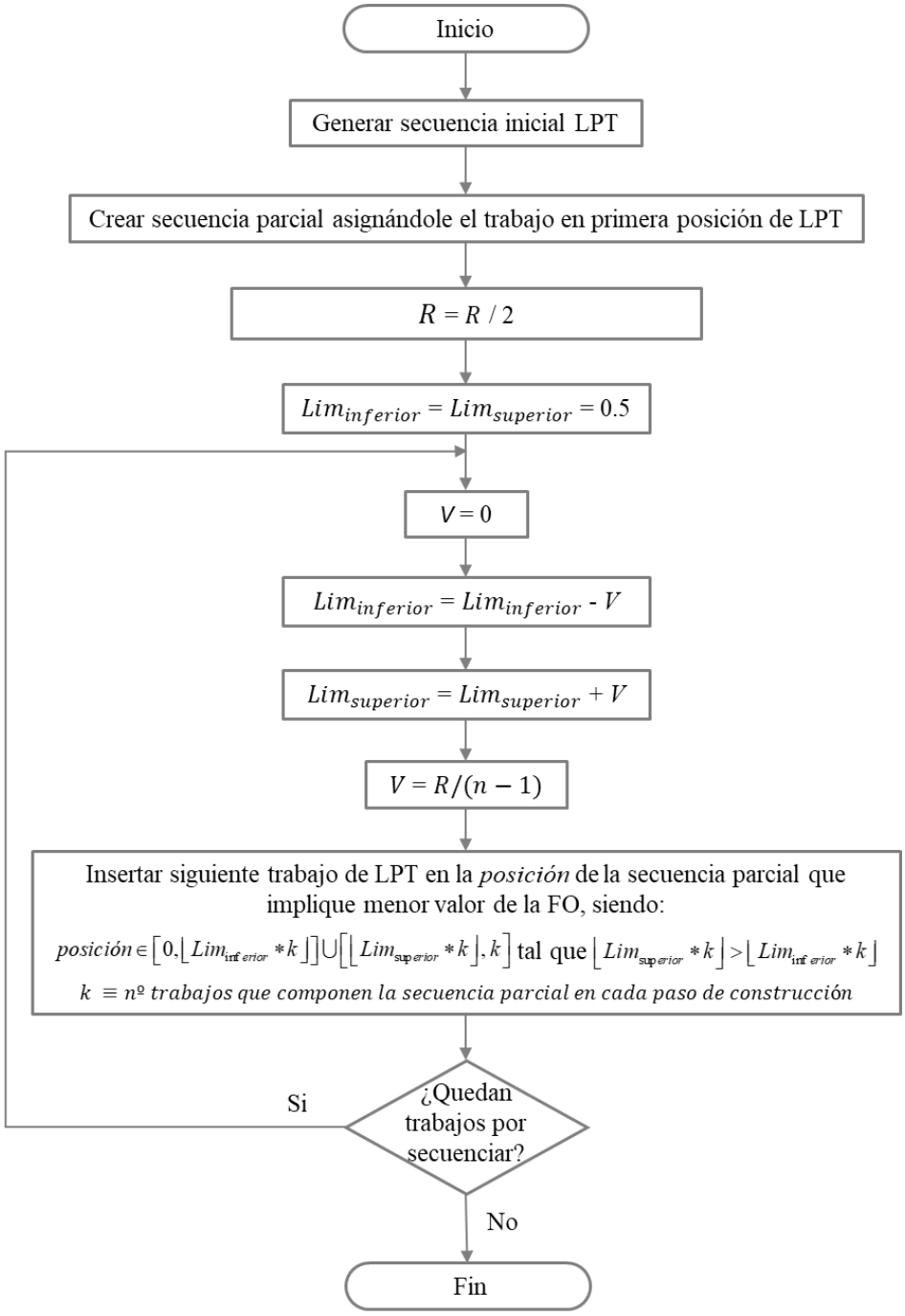

Figura 6.8. Diagrama de flujo heurística NEH_FO_RED.

A continuación, en la Figura 6.9, se muestra el pseudocódigo correspondiente al algoritmo.

Entrada: datos de la instancia, parámetro R
Salida: Π, C_{max}
Inicio

$P_{\gamma_j} = \sum_{i=1}^{m} p_{i\gamma_j}$;

$\Pi = \emptyset$, $J = \{ \gamma_1, \ldots, \gamma_n \}$ verificándose que $P_{\gamma_1} \geq P_{\gamma_2} \geq \ldots P_{\gamma_n}$;

$\Pi^{(1)} = \{ \gamma_1 \}$;

$R = {}^{R}/_{2}$;

$Lim_{inferior} = 0{,}5$;

$Lim_{superior} = 0{,}5$;

$R = 0$;

Para $k = 2$ **hasta** n **hacer**

$Lim_{inferior} = Lim_{inferior} - V$;

$Lim_{superior} = Lim_{superior} + V$;

$V = \frac{R}{n-1}$;

Insertar el trabajo γ_k en la *posición* de $\Pi^{(k)}$ que conlleve el menor *FO* para

$posición \in \left[0, \lfloor Lim_{\text{inf erior}} * k \rfloor\right] \cup \left[\lfloor Lim_{\text{superior}} * k \rfloor, k\right]$ tal que $\lfloor Lim_{\text{superior}} * k \rfloor > \lfloor Lim_{\text{inf erior}} * k \rfloor$;

Fin

Calcular valor C_{max} para Π;

Devolver Π, C_{max}

Fin

Figura 6.9. Pseudocódigo algoritmo NEH_FO_RED.

6.4 Heurística 4: PFX

La cuarta propuesta toma como base la heurística PF propuesta por Pan & Wang (2012), que construye la secuencia insertando en la última posición de la secuencia parcial los trabajos en función de cual proporcione un menor valor de un indicador.

El valor de dicho indicador se corresponde con la suma del tiempo ocioso y del tiempo de bloqueo en las máquinas, asumiendo que ambos tienen la misma influencia sobre el makespan. Sin embargo, su efecto resulta mayor cuando el tiempo de bloqueo se da en las primeras máquinas del taller en comparación con cuando se da en las últimas máquinas. Esto se debe a que un atasco en las primeras máquinas provoca un retraso en el procesado de futuros trabajos. Además, cuando el atasco es provocado por los primeros trabajos que se procesan en el taller conlleva un mayor efecto en el valor del makespan que si el atasco lo provocan los últimos trabajos de la secuencia (no tienen trabajos sucesores cuyo procesado se pueda retrasar).

Por otro lado, se ha observado que en el tipo de problema abordado, en el que se dispone de un buffer con capacidad para almacenar un número determinado de trabajos, es importante tener en cuenta si el buffer está ocupado o no a la hora de secuenciar. Disponer de un buffer entre dos máquinas nos aporta como ventaja, por un lado, que la máquina anterior disponga de un margen para ir procesando otro trabajo mientras el que acaba de procesar espera a que la siguiente máquina del taller se quede libre. Por otro lado, disponer de un buffer puede evitar que la máquina tras él se quede parada esperando a que el trabajo que tiene que procesar termine de ser procesado en la máquina anterior, y por tanto se produzca un tiempo ocioso. El hecho de que cuando

una máquina esté libre, la máquina anterior continúe procesando un trabajo y no haya ningún trabajo esperando en el buffer, provoca que haya tiempo ocioso, y por consecuencia, aumentaría el tiempo de finalización total (makespan).

Teniendo en consideración las conclusiones anteriores, surge la idea de modificar el indicador que usa la heurística PF para seleccionar el trabajo a secuenciar en cada iteración. Este indicador, cuyo valor es mejor cuanto mejor sea, originalmente se correspondía con la suma del tiempo ocioso y el tiempo de bloqueo. La modificación propuesta consiste en incluir en el indicador la influencia del tiempo de holgura mencionada en la descripción del problema (capítulo 3.2), de forma que el valor del indicador incremente cuando el idle time sea distinto de cero, sumándole al indicador original el valor del tiempo de holgura ponderado por un valor *S*. Además, se divide dicha suma entre *i*, el índice correspondiente a cada máquina. De esta forma se consigue que el numerador tenga mayor importancia en las primeras máquinas (se divide por un número más pequeño: 1, 2, 3…) que en las últimas máquinas (se divide por un número más grande: …, *m* - 1, *m*), y además, se penalizará que, cuando haya tiempo ocioso en una máquina, el buffer previo a ésta no tenga ningún trabajo en cola para ser procesado.

El indicador propuesto es el siguiente:

$$\delta = \sum_{i=1}^{m} \frac{t_{ocioso} + t_{bloqueo} + S * t_{holgura}}{i} \qquad (2.3)$$

El diagrama de flujo y el pseudocódigo de esta heurística son idénticos a los de la heurística PF (véase Figura 5.4 y Figura 5.5), variando únicamente el valor del indicador.

Esta heurística reduce en gran medida el tiempo computacional del algoritmo en comparación con los otros métodos propuestos, ya que es mucho más simple y no es necesario evaluar la secuencia parcial completa cada vez que se inserta un trabajo en una posición distinta, sino que al insertarse siempre en la última posición solo es necesario evaluar la influencia de ese trabajo sobre el valor del makespan.

6.5 Heurística 5: PFX_LS

Uniendo el concepto de la lista de movimientos prometedores a la heurística anterior, surge la siguiente heurística propuesta. Consiste en almacenar en una lista los movimientos prometedores encontrados durante el proceso de construcción de la secuencia. Una vez obtenida la secuencia completa mediante la heurística PFX, se realiza una pequeña búsqueda local reinsertando los movimientos de la lista en la secuencia, y actualizando la secuencia final cada vez que se encuentra una secuencia mejor (que proporcione menor valor de makespan). Así, se consigue mejorar la calidad de la solución de la heurística anterior.

En la Figura 6.10 se representa el procedimiento mediante un diagrama de flujo. La dinámica seguida a la hora de introducir movimientos prometedores en la lista es idéntica a la mencionada anteriormente para la heurística MCHcad, a diferencia de que el valor del incremento (*I*) en este caso viene dado por la diferencia del valor del indicador de la heurística PFX para el movimiento candidato con el mejor valor encontrado en esa iteración:

$$I' = \delta_{candidato} - \delta_{mínimo} \qquad (2.4)$$

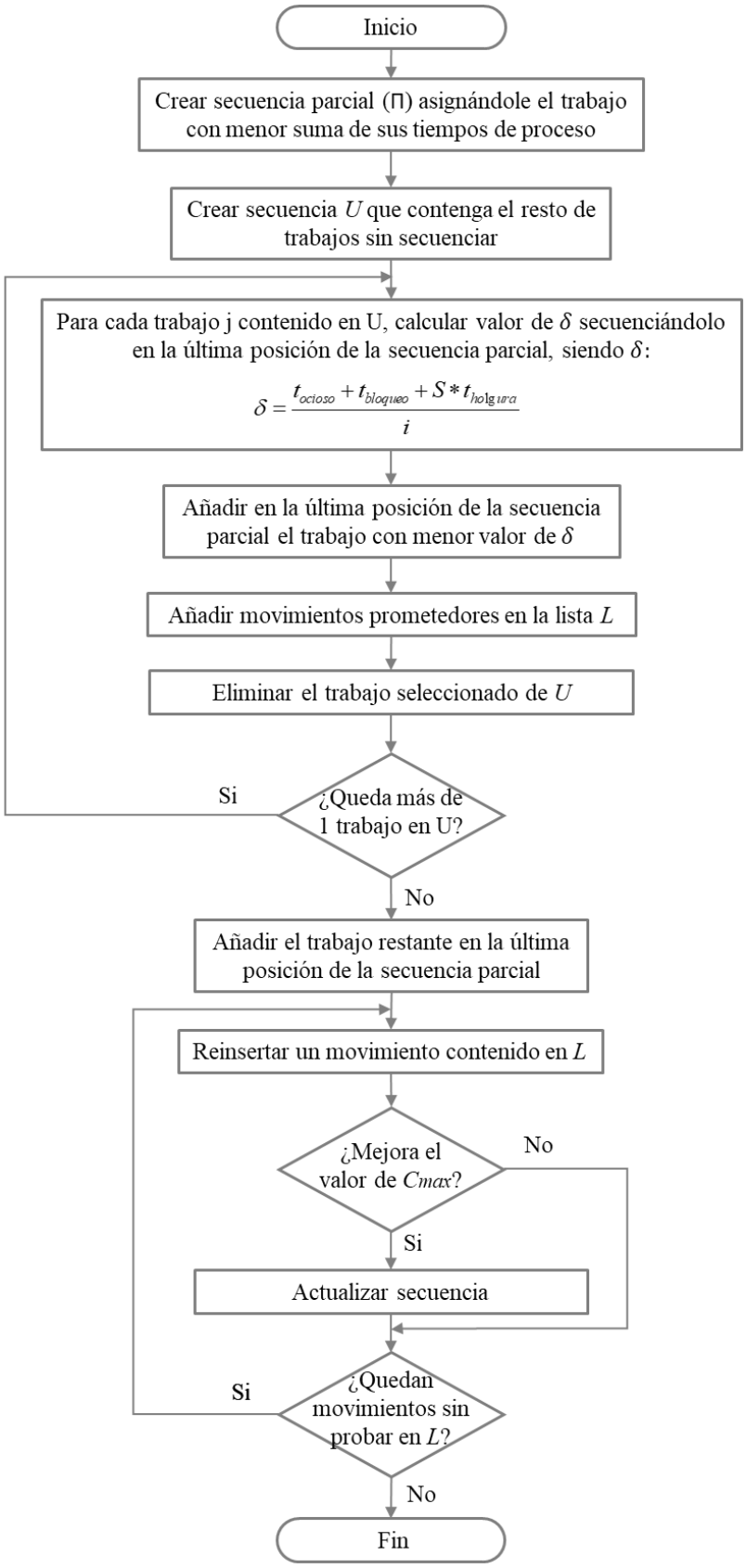

Figura 6.10. Diagrama de flujo heurística PFX_LS.

6.6 Heurística 6: PFX-MCH*cad*(*x*)

Analizando los métodos propuestos hasta el momento, se puede observar que la heurística PFX proporciona una solución de forma rápida, mientras que la MCH*cad* proporciona una solución de mayor calidad, pero en un tiempo computacional mayor. Por este motivo, se considera beneficioso combinar ambos métodos para unir las ventajas que cada uno proporciona, y conseguir una solución de calidad en un mejor tiempo computacional.

Por tanto, la cuarta propuesta de resolución del problema consiste en comenzar a construir la secuencia según la heurística PFX, y cuando quedan un número λ de trabajos por secuenciar, se continúa el procedimiento según la heurística MCH*cad*, del mismo modo que proponen Pan & Wang, (2012) con la heurística PF-NEH(x) mencionada en el apartado 5.2.5. Por tanto, cuando $\lambda = 0$, ningún trabajo se secuencia por el método MCH*cad*, obteniendo la solución mediante el método PFX. Por el contrario, si $\lambda = n-1$, todos los trabajos se secuencian según la heurística MCH*cad*, usando como secuencia inicial la secuencia obtenida por la PFX en lugar de usar la regla LPT.

El pseudocódigo de este algoritmo es idéntico al de la Figura 5.6, pero en lugar de la heurística PF se usa la PFX, y en lugar de la NEH se usa la MCH*cad*. También se conserva la generación de múltiples secuencias (tantas como indique el parámetro *x*) para encontrar una mejor solución.

6.7 Búsqueda local: RLS

Para mejorar la solución encontrada por los métodos propuestos, se propone añadir a las 6 heurísticas anteriores una búsqueda local que permita explorar las vecindades de la solución obtenida por los métodos y mejorar su calidad.

La búsqueda local de inserción consiste en extraer un trabajo de la secuencia e insertarlo en todas las posibles posiciones de la secuencia, evaluando su valor de la FO. De esta forma, cada vez que uno de los trabajos insertados proporcione una mejora en cuanto al objetivo fijado (minimización de la FO (2.1)), la secuencia se actualizará a la recién construida, siendo esta la nueva solución a partir de la cual se continuará la búsqueda. Este procedimiento sigue el funcionamiento de la búsqueda local de tipo "*First Best*", ya que nos quedamos con la mejor solución hasta ese instante y seguimos buscando a partir de ella, formando un vecindario variable.

En cada iteración se extraen uno a uno los trabajos de la secuencia, empezando por el trabajo en primera posición y acabando con el último. La búsqueda se detiene cuando transcurren n iteraciones en las que no se produce ninguna mejora.

En la Figura 6.11 se muestra el pseudocódigo de la búsqueda local, donde se puede comprender su dinámica de forma más clara.

Insertando esta búsqueda local en cada una de las heurísticas previas se desarrollan las siguientes heurísticas de mejora:

- NEH_FO_RLS: Heurística NEH_FO con búsqueda local RLS en la secuencia final.
- MCH*cad*_RLS: Heurística MCH*cad* con búsqueda local RLS en la secuencia final.
- PFX_RLS: Heurística PFX con búsqueda local RLS en la secuencia final.
- PFX_LS_RLS: Heurística PFX_LS con búsqueda local RLS en la secuencia final.
- NEH_FO_RED_RLS: Heurística NEH_FO_RED con búsqueda local RLS en la secuencia final.
- PFX_MCH*cad*_X_RLS: Heurística PFX_MCH*cad*(*x*) con búsqueda local RLS en la secuencia final.

Entrada: $\Pi = \{ \gamma_1, \gamma_2, ..., \gamma_n \}$, FO
Salida: Π
Inicio
  $i = 0$;
  $h = 0$;
  Mientras $i < n$ **hacer**
    $j = h \% n$;
    Extraer el trabajo de la posición j de la secuencia Π. Denotar $minFO_{Local}$ al mínimo valor de FO encontrado insertando el trabajo extraído en cada posición de Π;
    Si $minFO_{Local} < FO$ **hacer**
      $FO = minFO_{Local}$;
      $i = 0$;
    Fin
    Por el contrario
      $i = i + 1$;
    Fin
    $h = h + 1$;
  Fin
  Calcular valor C_{max} para Π;
  Devolver Π, C_{max}
Fin

Figura 6.11. Pseudocódigo búsqueda local RLS.

7 EVALUACIÓN COMPUTACIONAL

Los algoritmos aproximados correspondientes a los propuestos en el capítulo anterior han sido implementados en lenguaje de programación C# en el entorno de desarrollo Microsoft Visual Studio Community en su versión 2019. Se ha usado un conjunto de instancias para evaluar bajo las mismas condiciones cada uno de los algoritmos, y a continuación se presentan los bancos de pruebas e indicadores utilizados para evaluar su desempeño. Adicionalmente se presentan los resultados obtenidos en la comparación.

7.1 Bancos de pruebas

Para poder llevar a cabo la secuenciación de las tareas, se deben conocer los parámetros correspondientes al número de trabajos *n*, número de máquinas *m*, tiempos de proceso p_{ij}, y capacidad del buffer b_i. Por tanto, estos valores se definen mediante un conjunto de instancias que contengan los datos de entrada para el problema. En concreto, se utilizará el banco de pruebas presentado por Taillard (1993) para secuenciación en talleres de flujo regular.

El conjunto de instancias generadas se compone por la combinación de los siguientes números de trabajos y máquinas (*n* x *m*): {20x5, 20x10, 20x20, 50x5, 50x10, 50x20, 100x5, 100x10, 100x20, 200x10, 200x20}. Para cada una de estas combinaciones se han realizado 10 replicaciones, cada una de ellas con diferentes tiempos de proceso, obteniendo así un total de 110 instancias.

Respecto a la capacidad del buffer, se han evaluado las 110 instancias anteriores para los siguientes valores de capacidad: b_i. = {1, 2, 3, 4}. Por tanto, se obtienen un total de 440 instancias que se introducirán como datos de entrada en cada uno de los algoritmos, obteniendo como salida el valor del makespan de la secuencia proporcionada por cada método, y su tiempo computacional.

7.2 Indicadores de desempeño

Puesto que cada una de las heurísticas que se comparan proporcionan soluciones con distinto nivel de calidad en un tiempo de cómputo distinto, se usan indicadores que midan ambas magnitudes, como son:

- *Average Relative Percentage Deviation* ($ARPD_h$): mide la media de la calidad de las soluciones obtenidas por cada heurística (RPD_{ih}) para el conjunto de iteraciones evaluadas.

$$ARPD_h = \sum_{i=1}^{I} \frac{RPD_{ih}}{I}, \forall h = 1,...,H \qquad (3.1)$$

- *Average Computational CPU Times* (*ACT*)*:* mide el tiempo (*T*) que ha tardado cada heurística de media al evaluar las iteraciones.

$$ACT_h = \sum_{i=1}^{I} \frac{T_{ih}}{I}, \forall h = 1,...,H \qquad (3.2)$$

Donde:

- H es el número de algoritmos bajo comparación ($h \in \{1,...,H\}$)
- I es el número de instancias ($i \in \{1,...,I\}$)

La calidad de las soluciones se mide con la desviación del valor de la función objetivo obtenido con la heurística *h* para la iteración *i* (Obj_{ih}) respecto al mejor valor obtenido entre todas las heurísticas para esa iteración, como se puede ver en la siguiente ecuación:

$$RPD_{ih} = \frac{Obj_{ih} - Mejor_i}{Mejor_i} * 100, \forall h = 1, ..., H \qquad (3.3)$$

Para comparar los algoritmos aproximados en cuanto a su tiempo, se mide la desviación mediante el siguiente indicador:

- *Average Relative Percentage computation Time* (*ARPT*):

$$ARPT_h = 1 + \sum_{i=1}^{I} \frac{RPT_{ih}}{I}, \forall h = 1, ..., H \qquad (3.4)$$

Donde:

$$RPT_{ih} = \frac{T_{ih} - ACT_i}{ACT_i}, \forall h = 1, ..., H \qquad (3.5)$$

Siendo:

$$ACT_i = \sum_{h=1}^{H} \frac{T_{ih}}{H}, \forall i = 1, ..., I \qquad (3.6)$$

7.3 Valores parámetros

Algunas de las heurísticas propuestas requieren como dato de entrada el valor de uno o varios parámetros. Al influir el parámetro directamente en el esfuerzo computacional del algoritmo, se evaluarán los algoritmos para diferentes valores de los parámetros (ver Fernandez-Viagas et al., 2020 para enfoques similares). Para comparar valores diversos de los parámetros que sean representativos y proporcionen resultados distantes en cuanto a calidad y tiempo, se ha decidido establecer los siguientes valores:

- NEH_FO: este algoritmo no requiere ningún parámetro de entrada, únicamente fijar el valor de *Q* en la FO (2.1). Se considera suficiente que *Q* tome un valor de 1000000 unidades. Este valor aplicará al resto de métodos que también usan la misma FO.
- MCH*cad*: este algoritmo requiere dos parámetros de entrada:
    - Tamaño de la lista (*tam*): el tamaño de la lista tiene una influencia directa en el tiempo computacional del algoritmos, por lo que se analizarán diferentes variantes de la heurística utilizando diferentes valores del parámetro (ver Liu & Reeves, 2001, para un enfoque similar). Para poder analizar cómo se comporta el método en función del tamaño de la lista de movimientos prometedores, este parámetro tomará los siguientes valores: *tam* = {20, 40, 60, 80, 100}.
    - Caducidad (*cad*): se ha realizado un pequeño análisis de qué valores de caducidad proporcionan mejores resultados, evaluando el desempeño del método fijando el tamaño de la lista a un valor muy pequeño ($0.25 \cdot n$) para que el tamaño de la lista no tenga influencia en el tiempo computacional del algoritmo. Como se observa en la Tabla 7.1, el mejor valor de ARPD se da para *cad* = 10, que será el valor adoptado por el parámetro. Para comprobar si los diferentes valores que toma el parámetro afectan en el RPD, se ha realizado el test no paramétrico de Kruskal-Wallis (puesto que no se cumplen las condiciones paramétricas). Se

obyiene un p-valor de 0.0097, por lo que asumiendo un nivel de confianza al 95% (α=0.05) se puede afirmar que hay diferencias significativas, por lo que se justifica la elección del valor del parámetro *cad* = 10.

Tabla 7.1. Calibración parámetro *cad* heurística MCH*cad*

	***cad* = 5**	***cad* = 10**	***cad* = 15**	***cad* = 20**	***cad* = 25**
ARPD	0,46	0,40	0,42	0,43	0,45
ACT	3,18	3,27	3,31	3,33	3,35
ARPT	2,06	2,12	2,16	2,16	2,15

- NEH_FO_RED: el parámetro de entrada de este algoritmo es el intervalo reducido *R*. Cuanto mayor sea el intervalo que se reduce, se realizarán menos inserciones y el método se espera que sea más rápido. Por el contrario, cuanto menor sea el intervalo, se realizarán más inserciones y aunque el método será más lento, la calidad de la solución será mayor. Al igual que en el caso anterior, se evalúan distintos valores de *R*, en concreto: *R* = {0.2, 0.4, 0.6, 0.8}.

- PFX: el único parámetro de entrada de este algoritmo es el valor de *S* que se corresponde con la importancia que toma el valor del tiempo de holgura del buffer entre cada máquina en el indicador que emplea la heurística para determinar qué trabajo secuenciar en cada iteración. Se ha analizado la influencia de la holgura del buffer en este problema (véase Tabla 7.2), obteniendo los valores de ARPD, ACT y ARPT para distintos valores del parámetro. Para comprobar si los diferentes valores que toma el parámetro afectan en el RPD, se ha realizado el test no paramétrico de Kruskal-Wallis (puesto que no se cumplen las condiciones paramétricas), asumiendo un nivel de confianza al 95% (es decir, α = 0.05). Se obtiene un p-valor de 0.445, por lo que no afectan (no hay diferencias significativas) y por tanto, el valor adoptado por el parámetro en la evaluación computacional será valor que proporciona un mejor desempeño (*S* = 0.2).

Tabla 7.2. Calibración parámetro *S* heurística PFX

	***S* = 0**	***S* = 0.2**	***S* = 0.4**	***S* = 0.6**	***S* = 0.8**	***S* = 1**
ARPD	3,96	3,73	3,75	3,77	3,84	3,85
ACT	0,03	0,03	0,03	0,03	0,03	0,03
ARPT	0,06	0,06	0,06	0,06	0,06	0,06

- PFX_LS: este algoritmo requiere dos parámetros de entrada. Por un lado, el valor de *S*, que tomará el mismo valor que en PFX (*S* = 0.2). Por otro lado el tamaño de la lista que contiene los movimientos prometedores. Como la búsqueda local de reinserción de los movimientos de la lista se realiza una sola vez (al final del método), el tamaño de esta lista será mayor que el considerado para MCH*cad*, ya que en ese caso los movimientos de la lista se evaluaban en cada pasa de construcción de la secuencia. En este caso, tras un análisis previo, los valores considerados son: *tam_lista* = { 5, 50, 500, 5000}.

- PFX_MCH*cad*_X: este algoritmo tiene como entrada múltiples parámetros:

    - Peso de la holgura del buffer en el indicador de PFX (*S*): tomará el mismo valor que en los dos métodos anteriores (*S* = 0.2).

    - Caducidad (*cad*): tomará el mismo valor que para MCH*cad*, *cad* = 10.

    - Tamaño de la lista MCH*cad* (*tam*): se considerarán los mismos valores propuestos para MCH*cad*, *tam* = {20, 40, 60, 80, 100}.

    - Lambda (λ): Para determinar el valor de λ, es decir, el número de trabajos que pasan a

secuenciarse según MCH*cad*, se tiene en cuenta el valor usado por los autores en la heurística PF-NEH (Pan & Wang, 2012), que es $\lambda = 25$.

- Número de secuencias generadas (x): Al igual que en el caso anterior, se consideran los mismos valores que los propuestos para la heurística PF-NEH(x) (Pan & Wang, 2012), en concreto: $x = \{1, 2, 5\}$.

- Heurísticas con RLS: Además de evaluar todas las heurísticas anteriores, se evalúa también la aplicación de una búsqueda local a cada una de ellas. Los valores de los parámetros en este caso son los mismos que se han mencionado para cada heurística individualmente, excepto en el caso de las heurísticas MCH*cad y* PFX_MCH*cad*_X, donde para partir de una mejor solución en la búsqueda y ahorrar tiempo, se amplía el tamaño de la lista de movimientos prometedores, multiplicando su valor de tamaño por el número de trabajos de cada instancia (n). Por tanto, en este caso, $tam = \{20 \cdot n, 40 \cdot n, 60 \cdot n, 80 \cdot n, 100 \cdot n\}$.

Por tanto, se compararán entre sí los 6 algoritmos propuestos, algunas de las cuales se comparan para diferentes valores de los parámetros, teniendo así un total de 30 variantes analizadas. Además, a todas ellas se les añadirá la búsqueda local mencionada en el apartado 6.7, sumando otras 30 heurísticas a la comparativa.

Para tener una referencia en cuanto a los cambios producidos en calidad de la solución y tiempo de cómputo por los métodos propuestos, también han sido incluidas en la comparativa 24 heurísticas constructivas propuestas por otros autores en la literatura. En concreto, las siguientes heurísticas han sido reimplementadas, bajo las mismas condiciones computacionales, e incluidas en la comparación:

- NEH propuesta en (Nawaz et al., 1983) para el problema de flowshop general y comprobada su alta eficiencia para el problema de flowshop con buffers limitados en (Leisten, 1990).

- PF, wPF, PW, MME, MME_B, PF-NEH(x), wPF-NEH(x), PW-NEH(x), PF-NEH(x)_LS, wPF-NEH(x)_LS y PW-NEH(x)_LS, propuestas en (Pan & Wang, 2012) para el problema de flowshop con bloqueo. El valor de los parámetros de entrada de dichos algoritmos será $x = \{1, 2, 5\}$ y $\lambda = 25$.

Por tanto, se evaluarán un total de 84 heurísticas, 24 propuestas en la literatura y 60 propuestas en este documento.

7.4 Resultados

Todas las heurísticas han sido codificadas en lenguaje de programación C# usando Visual Studio en un Intel Core i7-8565U con 1.80 GHz, 8 GB RAM, con Microsoft Windows 10 64 bit, y usando las mismas funciones y librerías de forma común. En el Anexo de este documento se incluye el código generado para la implementación de cada heurística propuesta, así como los métodos usados en la programación.

A continuación en la Tabla 7.3 se presentan los valores obtenidos de *ARPD* para cada heurística, agrupados por número de trabajos, máquinas, y capacidad del buffer. Las filas sombreadas se corresponden con las heurísticas de la literatura. Además, se marcan en color verde la mejor (o mejores en caso de empate) heurística para cada columna de la tabla, y en color rojo la heurística con peor desempeño.

Tabla 7.3. Valores de *ARPD* agrupados por n, m y b_i.

Heurísticas	***n***				***m***			***b_i***				**Total**
	20	**50**	**100**	**200**	**5**	**10**	**20**	**1**	**2**	**3**	**4**	
NEH	3,03	2,71	2,31	2,75	1,92	3,11	2,87	3,71	2,54	2,28	2,26	2,70
PF	**15,47**	**14,13**	**10,57**	**9,94**	**7,17**	**12,06**	**17,65**	**11,42**	**12,89**	**13,27**	**13,47**	**12,76**
wPF	13,96	11,54	8,26	6,29	6,97	9,79	13,46	9,18	10,46	10,80	10,98	10,35
PW	10,83	8,75	6,05	4,97	5,35	7,49	10,20	7,02	7,70	8,33	8,53	7,89
MME	4,18	4,09	3,27	3,39	3,14	4,23	3,76	4,14	3,65	3,68	3,58	3,76
MME_B	5,77	4,61	3,87	3,66	4,10	4,78	4,67	5,02	4,51	4,36	4,33	4,55

PF_NEH_R(1)	5,19	5,29	4,31	4,72	2,70	4,70	6,73	4,50	5,06	4,93	5,08	4,89
PF_NEH_R(2)	4,19	4,75	3,81	4,34	2,36	3,83	6,13	3,86	4,37	4,35	4,48	4,27
PF_NEH_R(5)	3,36	4,10	3,47	3,93	1,82	3,32	5,48	3,25	3,74	3,87	3,92	3,69
wPF_NEH_R(1)	5,12	4,87	3,64	3,00	2,79	4,10	5,53	3,86	4,30	4,38	4,51	4,26
wPF_NEH_R(2)	4,06	4,30	3,08	2,56	2,30	3,45	4,68	3,28	3,55	3,70	3,81	3,58
wPF_NEH_R(5)	3,05	3,65	2,68	2,27	1,82	2,84	3,97	2,65	2,95	3,07	3,22	2,97
PW_NEH_R(1)	4,29	3,92	2,76	2,41	2,33	3,50	4,18	3,15	3,31	3,61	3,66	3,43
PW_NEH_R(2)	3,44	3,31	2,38	2,04	1,84	2,80	3,68	2,57	2,79	3,01	3,07	2,86
PW_NEH_R(5)	2,67	2,95	2,10	1,74	1,49	2,37	3,18	2,10	2,36	2,62	2,61	2,42
PF_NEH_R_LS(1)	2,12	1,90	1,49	1,41	1,27	1,66	2,23	1,64	1,72	1,75	1,93	1,76
PF_NEH_R_LS(2)	1,43	1,38	1,16	1,20	0,90	1,10	1,80	1,29	1,26	1,24	1,42	1,30
PF_NEH_R_LS(5)	0,95	0,84	0,80	1,02	0,51	0,74	1,34	0,79	0,89	0,95	0,95	0,90
wPF_NEH_R_ LS(1)	2,20	1,76	1,26	1,13	1,22	1,71	1,86	1,30	1,77	1,69	1,75	1,63
wPF_NEH_R_ LS(2)	1,40	1,37	1,04	0,91	0,81	1,27	1,44	1,00	1,22	1,25	1,35	1,21
wPF_NEH_R_ LS(5)	0,66	0,92	0,75	0,69	0,41	0,79	1,00	0,55	0,73	0,84	0,93	0,76
PW_NEH_R_ LS(1)	1,88	1,59	1,19	0,91	0,98	1,51	1,70	1,31	1,43	1,43	1,58	1,44
PW_NEH_R_ LS(2)	1,34	1,09	0,95	0,71	0,66	1,08	1,32	0,95	0,99	1,12	1,15	1,05
PW_NEH_R_ LS(5)	0,83	0,79	0,66	**0,54**	0,48	0,73	0,88	**0,51**	0,72	0,80	0,84	0,72
NEH_FO	2,95	2,52	2,11	2,58	1,88	2,94	2,63	3,51	2,40	2,16	2,07	2,54
PFX	12,62	9,88	6,75	5,70	6,12	8,71	11,49	8,75	9,09	8,92	9,30	9,01
PFX_LS(5)	12,26	9,78	6,74	5,69	5,97	8,59	11,37	8,66	8,89	8,82	9,18	8,89
PFX_LS(50)	8,52	8,08	5,91	5,25	4,70	6,93	9,06	6,93	7,07	7,12	7,26	7,10
PFX_LS(500)	7,08	5,63	4,69	4,33	3,89	5,34	6,96	5,67	5,61	5,41	5,42	5,53
PFX_LS(5000)	7,08	5,47	4,02	3,52	3,84	5,07	6,24	5,23	5,22	5,00	5,18	5,16
NEH_FO_RED(0.2)	2,91	2,66	2,23	2,74	1,89	2,98	2,83	3,56	2,57	2,23	2,15	2,63
NEH_FO_RED(0.4)	3,00	2,98	2,48	2,95	1,96	3,29	3,06	3,87	2,77	2,43	2,32	2,85
NEH_FO_RED(0.6)	3,41	3,41	2,74	3,27	2,15	3,64	3,56	4,23	3,14	2,80	2,64	3,20
NEH_FO_RED(0.8)	4,31	3,64	3,20	3,68	2,45	4,17	4,19	4,78	3,73	3,26	3,07	3,71
MCHcad(20)	1,93	1,85	1,70	2,19	1,35	2,17	2,02	2,69	1,74	1,59	1,56	1,89
MCHcad(40)	1,75	1,63	1,53	2,06	1,28	2,00	1,76	2,49	1,64	1,41	1,32	1,72
MCHcad(60)	1,82	1,42	1,41	1,89	1,25	1,87	1,62	2,38	1,50	1,32	1,26	1,61
MCHcad(80)	1,69	1,30	1,38	1,85	1,11	1,79	1,58	2,30	1,35	1,23	1,25	1,53
MCHcad(100)	1,64	1,30	1,35	1,82	1,10	1,82	1,49	2,30	1,36	1,19	1,16	1,50
PFX_MCHcad_X(20)(1)	3,28	4,09	3,03	2,97	2,28	3,31	4,27	3,27	3,20	3,47	3,57	3,38
PFX_MCHcad_X(20)(2)	2,63	3,67	2,76	2,58	1,87	2,87	3,81	2,82	2,79	3,05	3,11	2,94
PFX_MCHcad_X(20)(5)	1,92	3,05	2,43	2,26	1,37	2,42	3,23	2,30	2,27	2,50	2,65	2,43
PFX_MCHcad_X(40)(1)	2,83	3,76	2,92	2,92	1,99	3,13	3,97	2,87	3,04	3,25	3,33	3,12
PFX_MCHcad_X(40)(2)	2,15	3,29	2,70	2,53	1,56	2,65	3,54	2,51	2,68	2,76	2,77	2,68
PFX_MCHcad_X(40)(5)	1,55	2,81	2,33	2,23	1,22	2,20	3,01	2,13	2,15	2,26	2,37	2,23
PFX_MCHcad_X(60)(1)	2,77	3,59	2,87	2,89	1,98	2,98	3,91	2,95	2,91	3,13	3,19	3,04
PFX_MCHcad_X(60)(2)	2,03	3,20	2,61	2,49	1,59	2,51	3,42	2,50	2,55	2,69	2,62	2,59
PFX_MCHcad_X(60)(5)	1,43	2,60	2,31	2,19	1,17	2,08	2,89	2,01	2,02	2,21	2,27	2,13
PFX_MCHcad_X(80)(1)	2,69	3,52	2,85	2,88	1,96	2,93	3,83	2,97	2,80	3,03	3,17	2,99
PFX_MCHcad_X(80)(2)	1,97	3,06	2,57	2,45	1,53	2,41	3,37	2,50	2,37	2,57	2,64	2,52
PFX_MCHcad_X(80)(5)	1,44	2,56	2,24	2,19	1,15	2,04	2,87	2,04	1,98	2,13	2,26	2,10
PFX_MCHcad_X(100)(1)	2,57	3,53	2,78	2,86	1,95	2,78	3,86	2,83	2,83	3,05	3,06	2,94
PFX_MCHcad_X(100)(2)	1,89	3,00	2,58	2,43	1,48	2,34	3,36	2,36	2,41	2,59	2,54	2,48

PFX_MCHcad_X(100)(5)	1,24	2,54	2,20	2,15	1,07	1,95	2,80	1,91	1,92	2,05	2,20	2,02
NEH_FO_RLS	1,22	1,28	0,84	1,21	0,68	1,36	1,24	1,67	1,00	0,97	0,89	1,13
PFX_RLS	2,01	1,55	1,20	1,24	0,86	1,72	1,82	1,75	1,54	1,41	1,39	1,52
PFX_LS_RLS(5)	1,92	1,55	1,19	1,24	0,84	1,67	1,81	1,76	1,48	1,38	1,36	1,50
PFX_LS_RLS(50)	1,84	1,52	1,09	1,11	0,93	1,37	1,83	1,57	1,46	1,38	1,26	1,42
PFX_LS_RLS(500)	2,07	1,51	1,14	1,22	1,09	1,52	1,81	1,75	1,47	1,30	1,52	1,51
PFX_LS_RLS(5000)	2,07	1,53	1,14	1,14	1,10	1,53	1,77	1,72	1,46	1,36	1,46	1,50
NEH_FO_RED_RLS(0.2)	1,28	1,22	0,79	1,24	0,70	1,38	1,18	1,73	0,93	0,95	0,87	1,12
NEH_FO_RED_RLS(0.4)	1,33	1,10	0,91	1,27	0,69	1,41	1,21	1,75	0,95	0,93	0,93	1,14
NEH_FO_RED_RLS(0.6)	1,31	1,19	0,81	1,23	0,78	1,35	1,16	1,72	0,91	0,97	0,91	1,13
NEH_FO_RED_RLS(0.8)	1,50	1,06	0,89	1,29	0,91	1,37	1,19	1,72	1,03	1,04	0,91	1,18
MCHcad_RLS(20·n)	1,14	0,64	0,60	0,91	0,67	1,00	0,74	1,32	0,72	0,63	0,60	0,82
MCHcad_RLS(40·n)	1,14	0,64	0,58	0,87	0,64	0,99	0,73	1,32	0,67	0,62	0,60	0,80
MCHcad_RLS(60·n)	1,14	0,64	0,60	0,90	0,67	1,02	0,72	1,35	0,70	0,60	0,60	0,81
MCHcad_RLS(80·n)	1,14	0,64	0,60	0,89	0,67	1,01	0,72	1,36	0,70	0,59	0,60	0,81
MCHcad_RLS(100·n)	1,14	0,64	0,60	0,88	0,67	1,01	**0,71**	1,36	0,70	0,58	0,60	0,81
PFX_MCHcad_X_RLS(20·n)(1)	1,46	1,31	1,02	1,03	0,81	1,25	1,50	1,29	1,23	1,15	1,21	1,22
PFX_MCHcad_X_RLS(20·n)(2)	0,94	0,98	0,77	0,85	0,54	0,89	1,16	0,95	0,92	0,87	0,82	0,89
PFX_MCHcad_X_RLS(20·n)(5)	**0,41**	**0,53**	**0,53**	0,63	**0,28**	0,47	0,74	0,54	**0,52**	**0,51**	**0,49**	**0,51**
PFX_MCHcad_X_RLS(40·n)(1)	1,46	1,31	1,05	1,05	0,81	1,26	1,53	1,30	1,23	1,15	1,25	1,23
PFX_MCHcad_X_RLS(40·n)(2)	0,94	0,98	0,81	0,84	0,54	0,90	1,16	0,95	0,91	0,87	0,86	0,90
PFX_MCHcad_X_RLS(40·n)(5)	**0,41**	**0,53**	0,54	0,64	**0,28**	**0,46**	0,76	0,53	**0,52**	0,53	0,50	0,52
PFX_MCHcad_X_RLS(60·n)(1)	1,46	1,31	1,05	1,05	0,81	1,26	1,53	1,30	1,23	1,15	1,25	1,23
PFX_MCHcad_X_RLS(60·n)(2)	0,94	0,98	0,81	0,84	0,54	0,90	1,16	0,95	0,91	0,87	0,86	0,90
PFX_MCHcad_X_RLS(60·n)(5)	**0,41**	**0,53**	0,54	0,64	**0,28**	**0,46**	0,76	0,53	**0,52**	0,53	0,50	0,52
PFX_MCHcad_X_RLS(80·n)(1)	1,46	1,31	1,05	1,05	0,81	1,26	1,53	1,30	1,23	1,15	1,25	1,23
PFX_MCHcad_X_RLS(80·n)(2)	0,94	0,98	0,81	0,84	0,54	0,90	1,16	0,95	0,91	0,87	0,86	0,90
PFX_MCHcad_X_RLS(80·n)(5)	**0,41**	**0,53**	0,54	0,64	**0,28**	**0,46**	0,76	0,53	**0,52**	0,53	0,50	0,52
PFX_MCHcad_X_RLS(100·n)(1)	1,46	1,31	1,05	1,05	0,81	1,26	1,53	1,30	1,23	1,15	1,25	1,23
PFX_MCHcad_X_RLS(100·n)(2)	0,94	0,98	0,81	0,84	0,54	0,90	1,16	0,95	0,91	0,87	0,86	0,90
PFX_MCHcad_X_RLS(100·n)(5)	**0,41**	**0,53**	0,54	0,64	**0,28**	**0,46**	0,76	0,53	**0,52**	0,53	0,50	0,52

A continuación, en la Tabla 7.4 se muestran los resultados obtenidos para ARPD agrupados por tipo de instancia.

Tabla 7.4. Valores de *ARPD* agrupados por tamaño de instancia.

Heurísticas	***n* x *m***											**Total**
	20 x 5	**20 x 10**	**20 x 20**	**50 x 5**	**50 x 10**	**50 x 20**	**100 x 5**	**100 x 10**	**100 x 20**	**200 x 10**	**200 x 20**	
NEH	2,68	3,74	2,68	1,55	3,49	3,10	1,53	2,58	2,83	2,63	2,87	2,70
PF	10,34	**18,43**	**17,63**	**7,71**	**15,68**	**18,99**	**3,46**	**9,24**	**19,00**	**4,89**	**15,00**	**12,76**
wPF	**11,11**	15,11	15,68	6,73	12,79	15,11	3,06	7,33	14,41	3,92	8,66	10,35
PW	9,41	11,69	11,40	4,41	9,94	11,91	2,25	5,56	10,32	2,77	7,17	7,89
MME	4,26	4,88	3,38	2,80	5,42	4,06	2,37	3,50	3,94	3,12	3,67	3,76
MME_B	5,70	5,90	5,71	3,63	5,41	4,80	2,99	4,22	4,41	3,57	3,75	4,55
PF_NEH_R(1)	3,62	6,25	5,71	2,95	6,61	6,30	1,53	3,89	7,52	2,04	7,41	4,89
PF_NEH_R(2)	3,20	4,50	4,88	2,70	5,74	5,82	1,18	3,36	6,87	1,73	6,95	4,27
PF_NEH_R(5)	2,43	3,79	3,86	2,12	5,09	5,11	0,93	2,89	6,58	1,49	6,36	3,69
wPF_NEH_R(1)	3,95	5,83	5,57	2,99	5,31	6,32	1,42	3,55	5,94	1,70	4,31	4,26

wPF_NEH_R(2)	3,22	4,54	4,42	2,53	4,84	5,52	1,14	3,00	5,09	1,43	3,68	3,58
wPF_NEH_R(5)	2,41	3,37	3,36	2,13	4,25	4,59	0,94	2,51	4,60	1,21	3,33	2,97
PW_NEH_R(1)	3,80	4,99	4,07	2,18	5,31	4,27	1,02	2,50	4,77	1,19	3,62	3,43
PW_NEH_R(2)	2,94	3,80	3,56	1,83	4,19	3,91	0,75	2,23	4,15	0,98	3,11	2,86
PW_NEH_R(5)	2,20	3,08	2,74	1,64	3,61	3,62	0,62	1,95	3,73	0,83	2,65	2,42
PF_NEH_R_LS(1)	2,24	1,92	2,18	1,00	2,58	2,13	0,56	1,44	2,48	0,69	2,14	1,76
PF_NEH_R_LS(2)	1,57	1,15	1,58	0,77	1,70	1,68	0,37	1,04	2,08	0,52	1,88	1,30
PF_NEH_R_LS(5)	0,96	0,81	1,08	0,36	1,12	1,06	0,23	0,71	1,48	0,33	1,72	0,90
wPF_NEH_R_ LS(1)	2,07	2,75	1,76	1,02	2,23	2,04	0,56	1,22	2,01	0,63	1,62	1,63
wPF_NEH_R_ LS(2)	1,28	1,69	1,24	0,73	1,79	1,59	0,41	1,07	1,65	0,52	1,30	1,21
wPF_NEH_R_ LS(5)	0,66	0,74	0,58	0,30	1,42	1,04	0,25	0,68	1,33	0,32	1,06	0,76
PW_NEH_R_ LS(1)	1,70	2,27	1,68	0,80	2,17	1,80	0,45	1,14	1,97	0,48	1,35	1,44
PW_NEH_R_ LS(2)	1,18	1,62	1,23	0,48	1,50	1,30	0,33	0,86	1,66	0,35	1,08	1,05
PW_NEH_R_ LS(5)	0,92	0,93	0,63	0,30	1,13	0,94	0,23	0,61	1,14	**0,25**	0,82	0,72
NEH_FO	2,67	3,78	2,40	1,50	3,20	2,86	1,47	2,30	2,56	2,48	2,68	2,54
PFX	9,54	14,24	14,08	5,79	10,36	13,48	3,03	6,67	10,57	3,56	7,85	9,01
PFX_LS(5)	9,10	13,85	13,84	5,79	10,30	13,26	3,03	6,67	10,54	3,55	7,83	8,89
PFX_LS(50)	6,49	9,55	9,53	4,77	9,05	10,44	2,83	5,81	9,10	3,32	7,17	7,10
PFX_LS(500)	5,95	7,72	7,56	3,49	6,41	6,98	2,24	4,50	7,34	2,72	5,94	5,53
PFX_LS(5000)	5,95	7,72	7,56	3,57	6,16	6,69	1,98	4,06	6,01	2,32	4,71	5,16
NEH_FO_RED(0.2)	2,56	3,67	2,50	1,53	3,28	3,16	1,57	2,38	2,76	2,57	2,91	2,63
NEH_FO_RED(0.4)	2,60	4,02	2,38	1,50	3,83	3,63	1,80	2,61	3,04	2,71	3,19	2,85
NEH_FO_RED(0.6)	2,69	4,21	3,32	1,87	4,27	4,10	1,88	2,92	3,42	3,15	3,40	3,20
NEH_FO_RED(0.8)	3,31	5,27	4,35	1,81	4,62	4,50	2,23	3,30	4,07	3,50	3,86	3,71
MCHcad(20)	1,87	2,34	1,60	0,90	2,40	2,24	1,27	1,83	2,00	2,12	2,26	1,89
MCHcad(40)	1,67	2,09	1,49	0,95	2,19	1,76	1,22	1,65	1,73	2,07	2,04	1,72
MCHcad(60)	1,84	2,11	1,51	0,84	1,74	1,67	1,08	1,65	1,49	1,99	1,80	1,61
MCHcad(80)	1,51	1,96	1,60	0,80	1,67	1,44	1,03	1,59	1,53	1,95	1,76	1,53
MCHcad(100)	1,41	2,07	1,44	0,80	1,72	1,39	1,08	1,55	1,42	1,93	1,72	1,50
PFX_MCHcad_X(20)(1)	2,96	3,70	3,18	2,53	4,83	4,93	1,34	2,88	4,86	1,84	4,10	3,38
PFX_MCHcad_X(20)(2)	2,31	3,01	2,58	2,21	4,31	4,50	1,09	2,65	4,53	1,52	3,64	2,94
PFX_MCHcad_X(20)(5)	1,52	2,38	1,84	1,69	3,68	3,79	0,89	2,33	4,08	1,29	3,24	2,43
PFX_MCHcad_X(40)(1)	2,30	3,48	2,69	2,38	4,36	4,55	1,29	2,81	4,66	1,88	3,97	3,12
PFX_MCHcad_X(40)(2)	1,62	2,56	2,27	2,00	3,88	3,99	1,07	2,66	4,36	1,51	3,54	2,68
PFX_MCHcad_X(40)(5)	1,23	1,82	1,60	1,59	3,43	3,40	0,85	2,24	3,89	1,31	3,15	2,23
PFX_MCHcad_X(60)(1)	2,36	3,21	2,75	2,30	4,09	4,39	1,30	2,80	4,51	1,82	3,97	3,04
PFX_MCHcad_X(60)(2)	1,73	2,19	2,17	2,00	3,74	3,85	1,04	2,61	4,18	1,49	3,49	2,59
PFX_MCHcad_X(60)(5)	1,18	1,63	1,49	1,47	3,17	3,15	0,85	2,22	3,85	1,30	3,09	2,13
PFX_MCHcad_X(80)(1)	2,38	3,10	2,59	2,27	4,05	4,24	1,22	2,75	4,57	1,82	3,93	2,99
PFX_MCHcad_X(80)(2)	1,71	2,14	2,07	1,89	3,49	3,81	0,98	2,54	4,17	1,46	3,43	2,52
PFX_MCHcad_X(80)(5)	1,24	1,64	1,45	1,40	3,15	3,13	0,81	2,11	3,81	1,28	3,10	2,10
PFX_MCHcad_X(100)(1)	2,37	2,61	2,73	2,27	4,02	4,31	1,20	2,68	4,46	1,81	3,91	2,94
PFX_MCHcad_X(100)(2)	1,51	2,01	2,15	1,92	3,36	3,71	1,01	2,53	4,18	1,46	3,40	2,48
PFX_MCHcad_X(100)(5)	0,92	1,46	1,34	1,51	3,06	3,04	0,79	2,06	3,76	1,24	3,05	2,02
NEH_FO_RLS	0,79	1,56	1,30	0,62	1,70	1,53	0,63	0,86	1,02	1,33	1,09	1,13
PFX_RLS	1,32	2,73	1,97	0,70	2,09	1,84	0,57	1,16	1,88	0,89	1,59	1,52
PFX_LS_RLS(5)	1,27	2,57	1,91	0,70	2,06	1,88	0,57	1,14	1,87	0,89	1,59	1,50

PFX_LS_RLS(50)	1,57	1,70	2,26	0,68	1,86	2,03	0,56	1,07	1,65	0,87	1,35	1,42
PFX_LS_RLS(500)	1,98	2,21	2,03	0,75	1,81	1,97	0,53	1,15	1,74	0,92	1,52	1,51
PFX_LS_RLS(5000)	1,98	2,21	2,03	0,76	1,89	1,93	0,56	1,18	1,69	0,85	1,43	1,50
NEH_FO_RED_RLS(0.2)	0,84	1,53	1,45	0,64	1,73	1,29	0,61	0,94	0,82	1,31	1,16	1,12
NEH_FO_RED_RLS(0.4)	0,85	1,73	1,39	0,53	1,44	1,32	0,70	1,09	0,95	1,37	1,17	1,14
NEH_FO_RED_RLS(0.6)	1,01	1,72	1,20	0,70	1,38	1,49	0,63	0,99	0,83	1,33	1,12	1,13
NEH_FO_RED_RLS(0.8)	1,47	1,71	1,34	0,56	1,33	1,27	0,70	1,00	0,99	1,43	1,15	1,18
MCHcad_RLS(20·n)	0,97	1,24	1,21	0,45	0,82	0,65	0,59	0,79	**0,42**	1,17	0,65	0,82
MCHcad_RLS(40·n)	0,97	1,24	1,21	0,44	0,85	**0,63**	0,53	0,73	0,48	1,15	0,60	0,80
MCHcad_RLS(60·n)	0,97	1,24	1,21	0,44	0,85	**0,63**	0,61	0,78	**0,42**	1,19	0,61	0,81
MCHcad_RLS(80·n)	0,97	1,24	1,21	0,44	0,85	**0,63**	0,61	0,78	**0,42**	1,16	0,62	0,81
MCHcad_RLS(100·n)	0,97	1,24	1,21	0,44	0,85	**0,63**	0,61	0,78	**0,42**	1,18	**0,58**	0,81
PFX_MCHcad_X_RLS(20·n)(1)	1,18	1,75	1,44	0,72	1,47	1,74	0,52	1,05	1,48	0,72	1,35	1,22
PFX_MCHcad_X_RLS(20·n)(2)	0,69	1,12	1,00	0,55	1,13	1,26	0,37	0,74	1,21	0,56	1,15	0,89
PFX_MCHcad_X_RLS(20·n)(5)	**0,39**	**0,56**	**0,27**	**0,23**	**0,54**	0,82	**0,21**	**0,40**	0,97	0,38	0,89	**0,51**
PFX_MCHcad_X_RLS(40·n)(1)	1,18	1,75	1,44	0,72	1,47	1,74	0,53	1,07	1,56	0,73	1,38	1,23
PFX_MCHcad_X_RLS(40·n)(2)	0,69	1,12	1,00	0,55	1,13	1,26	0,37	0,79	1,25	0,55	1,13	0,90
PFX_MCHcad_X_RLS(40·n)(5)	**0,39**	**0,56**	**0,27**	**0,23**	**0,54**	0,82	**0,21**	**0,40**	0,99	0,35	0,94	0,52
PFX_MCHcad_X_RLS(60·n)(1)	1,18	1,75	1,44	0,72	1,47	1,74	0,53	1,07	1,56	0,73	1,38	1,23
PFX_MCHcad_X_RLS(60·n)(2)	0,69	1,12	1,00	0,55	1,13	1,26	0,37	0,79	1,25	0,55	1,13	0,90
PFX_MCHcad_X_RLS(60·n)(5)	**0,39**	**0,56**	**0,27**	**0,23**	**0,54**	0,82	**0,21**	**0,40**	0,99	0,35	0,94	0,52
PFX_MCHcad_X_RLS(80·n)(1)	1,18	1,75	1,44	0,72	1,47	1,74	0,53	1,07	1,56	0,73	1,38	1,23
PFX_MCHcad_X_RLS(80·n)(2)	0,69	1,12	1,00	0,55	1,13	1,26	0,37	0,79	1,25	0,55	1,13	0,90
PFX_MCHcad_X_RLS(80·n)(5)	**0,39**	**0,56**	**0,27**	**0,23**	**0,54**	0,82	**0,21**	**0,40**	0,99	0,35	0,94	0,52
PFX_MCHcad_X_RLS(100·n)(1)	1,18	1,75	1,44	0,72	1,47	1,74	0,53	1,07	1,56	0,73	1,38	1,23
PFX_MCHcad_X_RLS(100·n)(2)	0,69	1,12	1,00	0,55	1,13	1,26	0,37	0,79	1,25	0,55	1,13	0,90
PFX_MCHcad_X_RLS(100·n)(5)	**0,39**	**0,56**	**0,27**	**0,23**	**0,54**	0,82	**0,21**	**0,40**	0,99	0,35	0,94	0,52

En la Tabla 7.5 se presentan los resultados de ACT agrupados por instancias, y su valor para el total de las instancias en conjunto. Como era previsible, a mayor tamaño de instancia (mayor número de trabajos y máquinas) el tiempo de cómputo de los algoritmos es mayor.

Tabla 7.5. Valores de *ACT* agrupados por tamaño de instancia.

Heurísticas	***n* x *m***											**Total**
	20 x 5	**20 x 10**	**20 x 20**	**50 x 5**	**50 x 10**	**50 x 20**	**100 x 5**	**100 x 10**	**100 x 20**	**200 x 10**	**200 x 20**	
NEH	0,00	0,00	0,00	0,00	0,01	0,01	0,03	0,05	0,09	0,37	0,78	0,12
PF	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,01	0,01	**0,00**
wPF	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,01	0,02	**0,00**
PW	0,00	0,00	0,00	0,00	0,00	0,00	0,01	0,01	0,02	0,07	0,14	0,02
MME	0,00	0,00	0,00	0,00	0,01	0,01	0,03	0,05	0,10	0,37	0,85	0,13
MME_B	0,00	0,00	0,00	0,00	0,01	0,01	0,03	0,05	0,10	0,38	0,78	0,12
PF_NEH_R(1)	0,00	0,00	0,00	0,00	0,01	0,01	0,02	0,03	0,06	0,13	0,28	0,05
PF_NEH_R(2)	0,00	0,00	0,00	0,01	0,01	0,02	0,03	0,06	0,12	0,26	0,56	0,10
PF_NEH_R(5)	0,00	0,00	0,00	0,01	0,03	0,06	0,08	0,15	0,29	0,63	1,40	0,24
wPF_NEH_R(1)	0,00	0,00	0,00	0,00	0,01	0,01	0,02	0,03	0,06	0,13	0,28	0,05
wPF_NEH_R(2)	0,00	0,00	0,00	0,01	0,01	0,03	0,03	0,06	0,12	0,25	0,56	0,10
wPF_NEH_R(5)	0,00	0,00	0,00	0,01	0,03	0,06	0,08	0,15	0,32	0,63	1,39	0,24
PW_NEH_R(1)	0,00	0,00	0,00	0,00	0,01	0,02	0,02	0,04	0,08	0,19	0,40	0,07

PW_NEH_R(2)	0,00	0,00	0,00	0,01	0,01	0,03	0,04	0,08	0,16	0,37	0,80	0,14
PW_NEH_R(5)	0,00	0,00	0,01	0,02	0,04	0,08	0,11	0,19	0,39	0,93	1,99	0,34
PF_NEH_R_LS(1)	0,00	0,00	0,01	0,03	0,07	0,20	0,24	0,62	1,55	4,53	15,92	2,11
PF_NEH_R_LS(2)	0,00	0,01	0,01	0,06	0,15	0,38	0,48	1,19	2,96	9,00	31,58	4,17
PF_NEH_R_LS(5)	0,01	0,02	0,04	0,15	0,37	0,96	1,17	2,94	8,08	22,04	75,98	10,16
wPF_NEH_R_ LS(1)	0,00	0,00	0,01	0,03	0,07	0,18	0,25	0,60	1,59	4,02	13,90	1,88
wPF_NEH_R_ LS(2)	0,00	0,01	0,01	0,06	0,15	0,36	0,49	1,19	3,09	8,11	26,87	3,67
wPF_NEH_R_ LS(5)	0,01	0,02	0,04	0,15	0,36	0,90	1,21	2,91	7,62	20,55	67,23	9,18
PW_NEH_R_ LS(1)	0,00	0,00	0,01	0,03	0,07	0,18	0,24	0,55	1,43	4,09	13,39	1,82
PW_NEH_R_ LS(2)	0,00	0,01	0,01	0,06	0,15	0,36	0,46	1,15	2,80	8,09	26,90	3,64
PW_NEH_R_ LS(5)	0,01	0,02	0,03	0,14	0,36	0,96	1,12	2,73	7,25	20,30	65,34	8,93
NEH_FO	0,00	0,00	0,00	0,00	0,01	0,02	0,03	0,05	0,11	0,41	0,86	0,14
PFX	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,01	0,02	**0,00**
PFX_LS(5)	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,01	0,02	**0,00**
PFX_LS(50)	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,01	0,01	0,02	**0,00**
PFX_LS(500)	0,00	0,00	0,00	0,00	0,00	0,01	0,01	0,01	0,02	0,02	0,05	0,01
PFX_LS(5000)	0,00	0,00	0,00	0,00	0,01	0,02	0,04	0,07	0,15	0,18	0,35	0,08
NEH_FO_RED(0.2)	0,00	0,00	0,00	0,00	0,01	0,01	0,03	0,05	0,09	0,35	0,74	0,12
NEH_FO_RED(0.4)	0,00	0,00	0,00	0,00	0,00	0,01	0,02	0,04	0,08	0,29	0,62	0,10
NEH_FO_RED(0.6)	0,00	0,00	0,00	0,00	0,00	0,01	0,02	0,03	0,06	0,23	0,49	0,08
NEH_FO_RED(0.8)	0,00	0,00	0,00	0,00	0,00	0,01	0,01	0,02	0,05	0,17	0,36	0,06
MCHcad(20)	0,00	0,00	0,00	0,01	0,01	0,02	0,04	0,07	0,13	0,46	0,97	0,16
MCHcad(40)	0,00	0,00	0,00	0,01	0,01	0,03	0,05	0,08	0,16	0,51	1,09	0,18
MCHcad(60)	0,00	0,00	0,00	0,01	0,02	0,04	0,06	0,09	0,19	0,57	1,20	0,20
MCHcad(80)	0,00	0,00	0,00	0,01	0,02	0,05	0,06	0,11	0,22	0,62	1,32	0,22
MCHcad(100)	0,00	0,00	0,00	0,01	0,02	0,05	0,07	0,12	0,25	0,67	1,43	0,24
PFX_MCHcad_X(20)(1)	0,00	0,00	0,00	0,00	0,01	0,02	0,02	0,04	0,08	0,15	0,33	0,06
PFX_MCHcad_X(20)(2)	0,00	0,00	0,00	0,01	0,02	0,04	0,04	0,08	0,16	0,31	0,66	0,12
PFX_MCHcad_X(20)(5)	0,00	0,01	0,01	0,02	0,04	0,10	0,11	0,19	0,39	0,77	1,65	0,30
PFX_MCHcad_X(40)(1)	0,00	0,00	0,00	0,01	0,01	0,03	0,02	0,04	0,09	0,16	0,35	0,07
PFX_MCHcad_X(40)(2)	0,00	0,00	0,01	0,01	0,02	0,05	0,05	0,09	0,18	0,33	0,71	0,13
PFX_MCHcad_X(40)(5)	0,00	0,01	0,01	0,03	0,06	0,13	0,13	0,22	0,45	0,82	1,77	0,33
PFX_MCHcad_X(60)(1)	0,00	0,00	0,00	0,01	0,01	0,03	0,03	0,05	0,10	0,18	0,38	0,07
PFX_MCHcad_X(60)(2)	0,00	0,00	0,01	0,01	0,03	0,07	0,06	0,10	0,20	0,35	0,75	0,14
PFX_MCHcad_X(60)(5)	0,01	0,01	0,02	0,04	0,07	0,16	0,14	0,25	0,50	0,88	1,90	0,36
PFX_MCHcad_X(80)(1)	0,00	0,00	0,00	0,01	0,02	0,04	0,03	0,05	0,11	0,19	0,41	0,08
PFX_MCHcad_X(80)(2)	0,00	0,00	0,01	0,02	0,03	0,08	0,06	0,11	0,22	0,37	0,81	0,16
PFX_MCHcad_X(80)(5)	0,01	0,01	0,02	0,04	0,08	0,19	0,16	0,27	0,56	0,93	2,01	0,39
PFX_MCHcad_X(100)(1)	0,00	0,00	0,00	0,01	0,02	0,04	0,04	0,06	0,12	0,20	0,42	0,08
PFX_MCHcad_X(100)(2)	0,00	0,01	0,01	0,02	0,04	0,08	0,07	0,12	0,25	0,40	0,85	0,17
PFX_MCHcad_X(100)(5)	0,01	0,01	0,03	0,05	0,10	0,21	0,17	0,30	0,61	0,99	2,13	0,42
NEH_FO_RLS	0,00	0,00	0,01	0,03	0,08	0,15	0,27	0,73	1,72	6,88	20,50	2,76
PFX_RLS	0,00	0,00	0,01	0,03	0,09	0,23	0,26	0,86	1,91	5,83	19,56	2,62
PFX_LS_RLS(5)	0,00	0,00	0,01	0,03	0,08	0,22	0,25	0,86	1,91	5,81	19,82	2,64
PFX_LS_RLS(50)	0,00	0,00	0,01	0,03	0,08	0,20	0,25	0,76	1,94	5,99	21,71	2,82
PFX_LS_RLS(500)	0,00	0,00	0,01	0,03	0,09	0,20	0,23	0,69	1,99	4,99	18,19	2,40
PFX_LS_RLS(5000)	0,00	0,00	0,01	0,03	0,09	0,21	0,25	0,74	2,01	5,51	20,23	2,64

NEH_FO_RED_RLS(0.2)	0,00	0,00	0,01	0,03	0,07	0,18	0,27	0,75	2,02	7,63	19,99	2,81
NEH_FO_RED_RLS(0.4)	0,00	0,00	0,01	0,03	0,08	0,19	0,25	0,69	1,86	6,91	20,38	2,76
NEH_FO_RED_RLS(0.6)	0,00	0,00	0,01	0,03	0,08	0,18	0,29	0,69	2,13	6,55	21,29	2,84
NEH_FO_RED_RLS(0.8)	0,00	0,00	0,01	0,03	0,08	0,18	0,24	0,80	1,90	6,56	17,05	2,44
MCHcad_RLS(20·n)	0,00	0,00	0,01	0,05	0,12	0,26	0,45	1,00	2,42	9,57	22,04	3,26
MCHcad_RLS(40·n)	0,00	0,00	0,01	0,05	0,12	0,26	0,45	1,12	2,50	9,12	25,02	3,51
MCHcad_RLS(60·n)	0,00	0,00	0,01	0,05	0,12	0,26	0,44	1,01	2,51	9,30	24,02	3,43
MCHcad_RLS(80·n)	0,00	0,00	0,01	0,05	0,12	0,26	0,44	1,01	2,51	9,62	22,27	3,30
MCHcad_RLS(100·n)	0,00	0,00	0,01	0,05	0,12	0,26	0,44	1,01	2,51	9,51	25,20	3,56
PFX_MCHcad_X_RLS(20·n)(1)	0,00	0,00	0,01	0,05	0,12	0,27	0,30	0,88	2,33	6,17	21,62	2,89
PFX_MCHcad_X_RLS(20·n)(2)	0,01	0,01	0,02	0,10	0,24	0,54	0,63	1,83	4,52	12,10	42,42	5,67
PFX_MCHcad_X_RLS(20·n)(5)	0,01	0,03	0,06	0,25	0,60	1,32	1,62	4,54	11,41	30,59	106,45	**14,26**
PFX_MCHcad_X_RLS(40·n)(1)	0,00	0,00	0,01	0,05	0,12	0,27	0,30	0,89	2,29	6,17	21,56	2,88
PFX_MCHcad_X_RLS(40·n)(2)	0,01	0,01	0,02	0,10	0,23	0,54	0,63	1,85	4,44	12,11	41,12	5,55
PFX_MCHcad_X_RLS(40·n)(5)	0,01	0,03	0,06	0,25	0,60	1,32	1,61	4,50	11,33	31,04	103,75	14,05
PFX_MCHcad_X_RLS(60·n)(1)	0,00	0,00	0,01	0,05	0,12	0,26	0,30	0,89	2,31	6,15	21,65	2,89
PFX_MCHcad_X_RLS(60·n)(2)	0,01	0,01	0,02	0,10	0,23	0,54	0,63	1,85	4,43	12,09	40,92	5,53
PFX_MCHcad_X_RLS(60·n)(5)	0,02	0,03	0,06	0,25	0,60	1,32	1,61	4,50	11,34	31,06	103,89	14,06
PFX_MCHcad_X_RLS(80·n)(1)	0,00	0,00	0,01	0,05	0,12	0,27	0,30	0,89	2,29	6,17	21,62	2,88
PFX_MCHcad_X_RLS(80·n)(2)	0,01	0,01	0,02	0,10	0,23	0,54	0,63	1,85	4,43	12,16	40,91	5,54
PFX_MCHcad_X_RLS(80·n)(5)	0,02	0,03	0,06	0,25	0,60	1,32	1,61	4,49	11,30	31,04	103,97	14,06
PFX_MCHcad_X_RLS(100·n)(1)	0,00	0,00	0,01	0,05	0,12	0,26	0,30	0,89	2,31	6,17	21,58	2,88
PFX_MCHcad_X_RLS(100·n)(2)	0,01	0,01	0,02	0,10	0,23	0,54	0,63	1,85	4,42	12,10	40,91	5,53
PFX_MCHcad_X_RLS(100·n)(5)	0,01	0,03	0,06	0,25	0,60	1,31	1,61	4,50	11,33	31,14	103,73	14,05

Adicionalmente, se presenta una recopilación de los indicadores de desempeño de calidad y tiempo tanto para las heurísticas de la literatura (Tabla 7.6) como para las heurísticas propuestas, con y sin búsqueda local (Tabla 7.7). Se resaltan en negrita las heurísticas más eficientes, es decir, las heurísticas que generan soluciones de calidad en un tiempo razonable (es decir, heurísticas que no son dominadas por otras en ambos indicadores, ARPD y ARPT).

Tabla 7.6. Resumen de los resultados computacionales (heurísticas literatura).

Heurística	ARPD	ACT	ARPT
NEH	2,70	0,12	0,05
PF	12,76	0,00	0,00
wPF	10,35	0,00	0,00
PW	7,89	0,02	0,01
MME	3,76	0,13	0,05
MME_B	4,55	0,12	0,05
PF_NEH_R(1)	4,89	0,05	0,03
PF_NEH_R(2)	4,27	0,10	0,08
PF_NEH_R(5)	3,69	0,24	0,27
wPF_NEH_R(1)	4,26	0,05	0,03
wPF_NEH_R(2)	3,58	0,10	0,07
wPF_NEH_R(5)	2,97	0,24	0,26
PW_NEH_R(1)	3,43	0,07	0,05
PW_NEH_R(2)	2,86	0,14	0,12

PW_NEH_R(5)	2,42	0,34	0,33
PF_NEH_R_LS(1)	1,76	2,11	0,72
PF_NEH_R_LS(2)	1,30	4,17	1,47
PF_NEH_R_LS(5)	0,90	10,16	3,70
wPF_NEH_R_ LS(1)	1,63	1,88	0,70
wPF_NEH_R_ LS(2)	1,21	3,67	1,43
wPF_NEH_R_ LS(5)	**0,76**	**9,18**	**3,63**
PW_NEH_R_ LS(1)	**1,44**	**1,82**	**0,69**
PW_NEH_R_ LS(2)	1,05	3,64	1,41
PW_NEH_R_ LS(5)	**0,72**	**8,93**	**3,52**

Tabla 7.7. Resumen de los resultados computacionales (heurísticas propuestas).

Sin búsqueda local				**Con búsqueda local**			
Heurística	**ARPD**	**ACT**	**ARPT**	**Heurística**	**ARPD**	**ACT**	**ARPT**
NEH_FO	**2,54**	**0,14**	**0,05**	**NEH_FO_RLS**	**1,13**	**2,76**	**0,79**
PFX	9,01	0,00	0,00	PFX_RLS	1,52	2,62	0,87
PFX_LS(5)	8,89	0,00	0,00	PFX_LS_RLS(5)	1,50	2,64	0,86
PFX_LS(50)	**7,10**	**0,00**	**0,00**	**PFX_LS_RLS(50)**	**1,42**	**2,82**	**0,83**
PFX_LS(500)	5,53	0,01	0,02	PFX_LS_RLS(500)	1,51	2,40	0,80
PFX_LS(5000)	5,16	0,08	0,07	PFX_LS_RLS(5000)	1,50	2,64	0,83
NEH_FO_RED(0.2)	**2,63**	**0,12**	**0,04**	**NEH_FO_RED_RLS(0.2)**	**1,12**	**2,81**	**0,80**
NEH_FO_RED(0.4)	2,85	0,10	0,04	NEH_FO_RED_RLS(0.4)	1,14	2,76	0,79
NEH_FO_RED(0.6)	**3,20**	**0,08**	**0,03**	NEH_FO_RED_RLS(0.6)	1,13	2,84	0,80
NEH_FO_RED(0.8)	**3,71**	**0,06**	**0,02**	**NEH_FO_RED_RLS(0.8)**	**1,18**	**2,44**	**0,78**
MCHcad(20)	**1,89**	**0,16**	**0,09**	MCHcad_RLS(20·n)	0,82	3,26	1,17
MCHcad(40)	**1,72**	**0,18**	**0,12**	**MCHcad_RLS(40·n)**	**0,80**	**3,51**	**1,19**
MCHcad(60)	**1,61**	**0,20**	**0,18**	MCHcad_RLS(60·n)	0,81	3,43	1,18
MCHcad(80)	**1,53**	**0,22**	**0,22**	**MCHcad_RLS(80·n)**	**0,81**	**3,30**	**1,17**
MCHcad(100)	**1,50**	**0,24**	**0,24**	MCHcad_RLS(100·n)	0,81	3,56	1,18
PFX_MCHcad_X(20)(1)	3,38	0,06	0,08	PFX_MCHcad_X_RLS(20·n)(1)	1,22	2,89	1,07
PFX_MCHcad_X(20)(2)	2,94	0,12	0,19	PFX_MCHcad_X_RLS(20·n)(2)	0,89	5,67	2,18
PFX_MCHcad_X(20)(5)	2,43	0,30	0,49	**PFX_MCHcad_X_RLS(20·n)(5)**	**0,51**	**14,26**	**5,54**
PFX_MCHcad_X(40)(1)	3,12	0,07	0,10	PFX_MCHcad_X_RLS(40·n)(1)	1,23	2,88	1,07
PFX_MCHcad_X(40)(2)	2,68	0,13	0,24	PFX_MCHcad_X_RLS(40·n)(2)	0,90	5,55	2,17
PFX_MCHcad_X(40)(5)	2,23	0,33	0,67	**PFX_MCHcad_X_RLS(40·n)(5)**	**0,52**	**14,05**	**5,51**
PFX_MCHcad_X(60)(1)	3,04	0,07	0,15	PFX_MCHcad_X_RLS(60·n)(1)	1,23	2,89	1,08
PFX_MCHcad_X(60)(2)	2,59	0,14	0,32	PFX_MCHcad_X_RLS(60·n)(2)	0,90	5,53	2,17
PFX_MCHcad_X(60)(5)	2,13	0,36	0,83	PFX_MCHcad_X_RLS(60·n)(5)	0,52	14,06	5,54
PFX_MCHcad_X(80)(1)	2,99	0,08	0,18	PFX_MCHcad_X_RLS(80·n)(1)	1,23	2,88	1,07
PFX_MCHcad_X(80)(2)	2,52	0,16	0,37	PFX_MCHcad_X_RLS(80·n)(2)	0,90	5,54	2,18
PFX_MCHcad_X(80)(5)	2,10	0,39	0,98	PFX_MCHcad_X_RLS(80·n)(5)	0,52	14,06	5,53
PFX_MCHcad_X(100)(1)	2,94	0,08	0,19	PFX_MCHcad_X_RLS(100·n)(1)	1,23	2,88	1,07
PFX_MCHcad_X(100)(2)	2,48	0,17	0,41	PFX_MCHcad_X_RLS(100·n)(2)	0,90	5,53	2,17
PFX_MCHcad_X(100)(5)	2,02	0,42	1,10	PFX_MCHcad_X_RLS(100·n)(5)	0,52	14,05	5,53

Con relación a los resultados anteriores, cabe destacar que, por lo general, las heurísticas que proporcionan mejores resultados son las que requieren mayor tiempo computacional, ya que se evalúa un conjunto mayor de posibles soluciones. En concreto, la heurística que proporciona mejores soluciones es la PF_MCH*cad*_X_RLS(20·*n*), con un valor ARPD de 0.51, pero a su vez es la heurística más lenta, con un ACT de 14.26 y un ARPT de 5.54. Además, la heurística de la literatura que mejores resultados proporciona es la PW_NEH_R_LS(5), con un ARPD de 0.72. En la Figura 7.1 se comparan gráficamente las medianas del RPD de estas dos heurísticas (mediana PF_MCH*cad*_X_RLS(20·*n*)(5) = 0.37 y mediana PW_NEH_R_LS(5) = 0.60). Para examinar si existen diferencias significativas entre ellas se han comparado estadísticamente usando el test no paramétrico de Mann-Whitney (ya que no se cumplen las condiciones paramétricas), asumiendo un nivel de confianza al 95% (es decir, $\alpha = 0.05$) para establecer el p-valor. Esta mejora resulta estadísticamente significativa, dando un p-valor de $7.765 \cdot 10^{-7}$ (p-valor < 0.05).

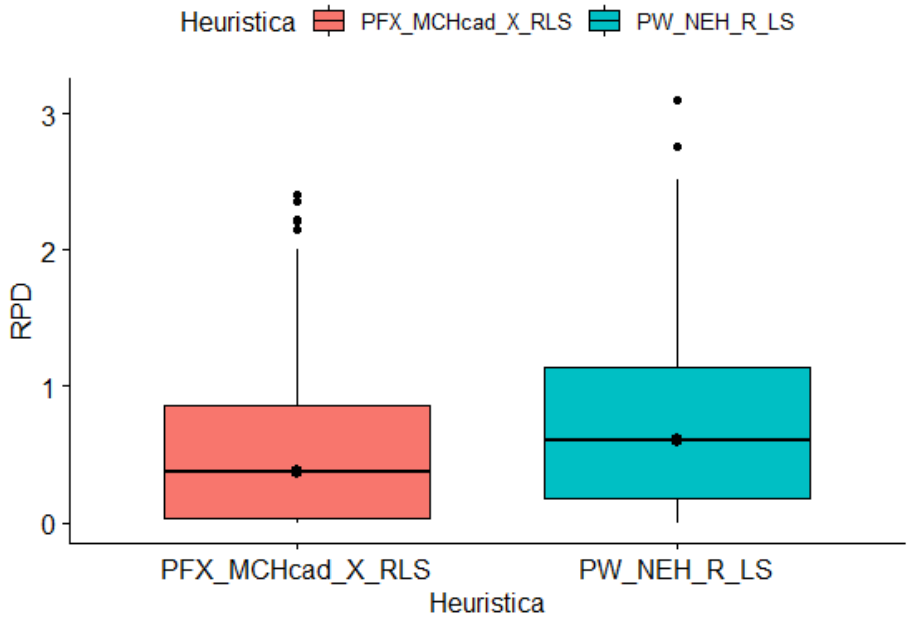

Figura 7.1. Comparativa medianas RPD de las heurísticas PFX_MCH*cad*_X_RLS Y PW_NEH_R_LS

En cuanto al método con peor desempeño, se trata de la heurística PF, pero a su vez también es el método más rápido. Sin embargo, hay heurísticas, como PFX o PFX_LS que consiguen una mejor solución siendo igual de rápidas. En la Figura 7.2 se muestra una comparativa de la mediana de los valores de RPD para las heurísticas PF (mediana = 12.41), wPF (mediana = 9.77), PW (mediana = 7.55) y PFX_LS(50) (mediana = 6.72). Se usa de nuevo el test no paramétrico Mann-Withney, asumiendo un nivel de confianza al 95% (es decir, $\alpha = 0.05$), para comparar la diferencia entre las medianas del RPD de las heurísticas PF, wPF y PW con la heurística PFX_LS(50), dando un p-valor de:

- PF vs. PFX_LS(50): p-valor = $2.2 \cdot 10^{-16}$. Se comprueba que PFX_LS mejora estadísticamente a PF para un mismo tiempo computacional.
- wPF vs. PFX_LS(50): p-valor = $2.2 \cdot 10^{-16}$. Se comprueba que PFX_LS mejora estadísticamente a wPF para un mismo tiempo computacional.
- PW vs. PFX_LS(50): p-valor = 0,0258. Se comprueba que, a pesar de requerir menor tiempo de cómputo, PFX_LS mejora estadísticamente a PW.

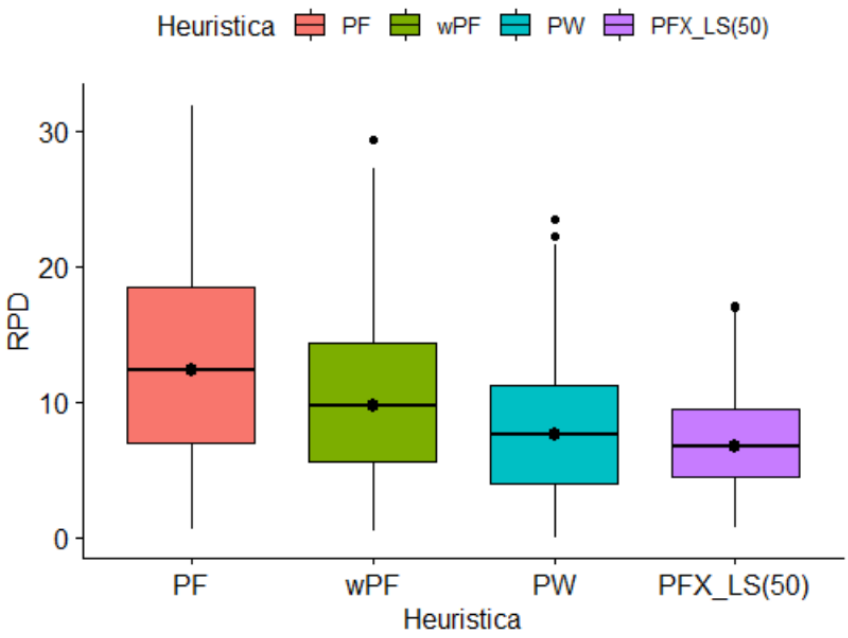

Figura 7.2. Comparativa medianas RPD de las heurísticas PF, wPF, PW Y PFX_LS

Cabe destacar también que al aplicar la búsqueda local RLS a las heurísticas, se mejora notablemente la calidad de la solución, pero, sin embargo, el tiempo computacional incrementa en gran medida. Además, la influencia de la búsqueda local provoca que las heurísticas alcancen resultados similares para distintos valores de sus parámetros.

Para comparar visualmente el desempeño de las heurísticas en cuanto a tiempo y calidad, se representan los valores de ARPD y ARPT de las heurísticas de la literatura (Figura 7.3), de las heurísticas propuestas sin búsqueda local (Figura 7.4) y de las heurísticas propuestas añadiéndoles la búsqueda local (Figura 7.5). Adicionalmente se representan todas ellas en la misma figura para poder compararlas entre ellas (Figura 7.6).

Para visualizar gráficamente la relación calidad-tiempo que presenta cada heurística se expone la siguiente Figura 7.7, donde el eje de abscisas se representa la desviación en cuanto a tiempo y en el eje de ordenadas la desviación en cuanto a calidad. Como se puede observar, hay heurísticas que dominan a otras, mejorando la calidad de las soluciones obtenidas, en un tiempo similar o menor. Por lo general, se obtienen mejores soluciones en un tiempo mayor. Dependiendo de la prioridad dada a cada aspecto (calidad de la función objetivo o esfuerzo computacional requerido), será más útil una heurística que proporcione resultados muy buenos en mayor tiempo, o se preferirá una solución rápida de una calidad intermedia.

Por último, para ver de forma más clara el desempeño de las heurísticas propuestas, la Tabla 7.8 muestra las heurísticas propuestas que dominan a cada una de las heurísticas de la literatura, es decir, que las mejoran tanto en calidad (ARPD) como en tiempo (ARPT). Se comprueba que todas las heurísticas de la literatura son mejoradas por alguna de las propuestas, excepto para wPF_NEH_R_ LS(5), PW_NEH_R_ LS(1) y PW_NEH_R_ LS(5), que ninguna de las propuestas consigue alcanzar mejores soluciones en un menor tiempo. Sin embargo, se consigue una mejor solución con el método PFX_MCHcad_X_RLS(20·n)(5), aunque en un tiempo mayor.

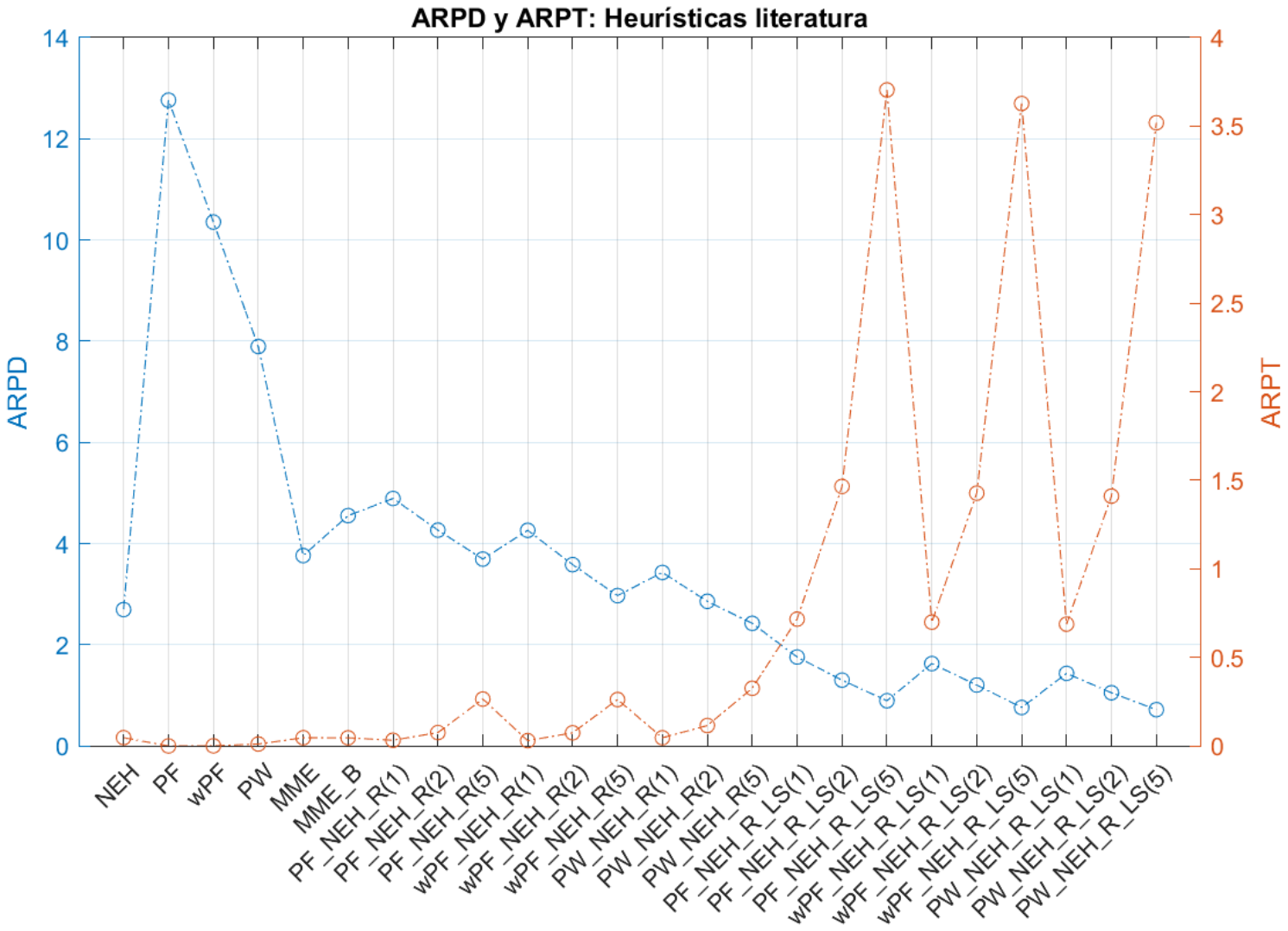

Figura 7.3. ARPD y ARPT (Heurísticas literatura).

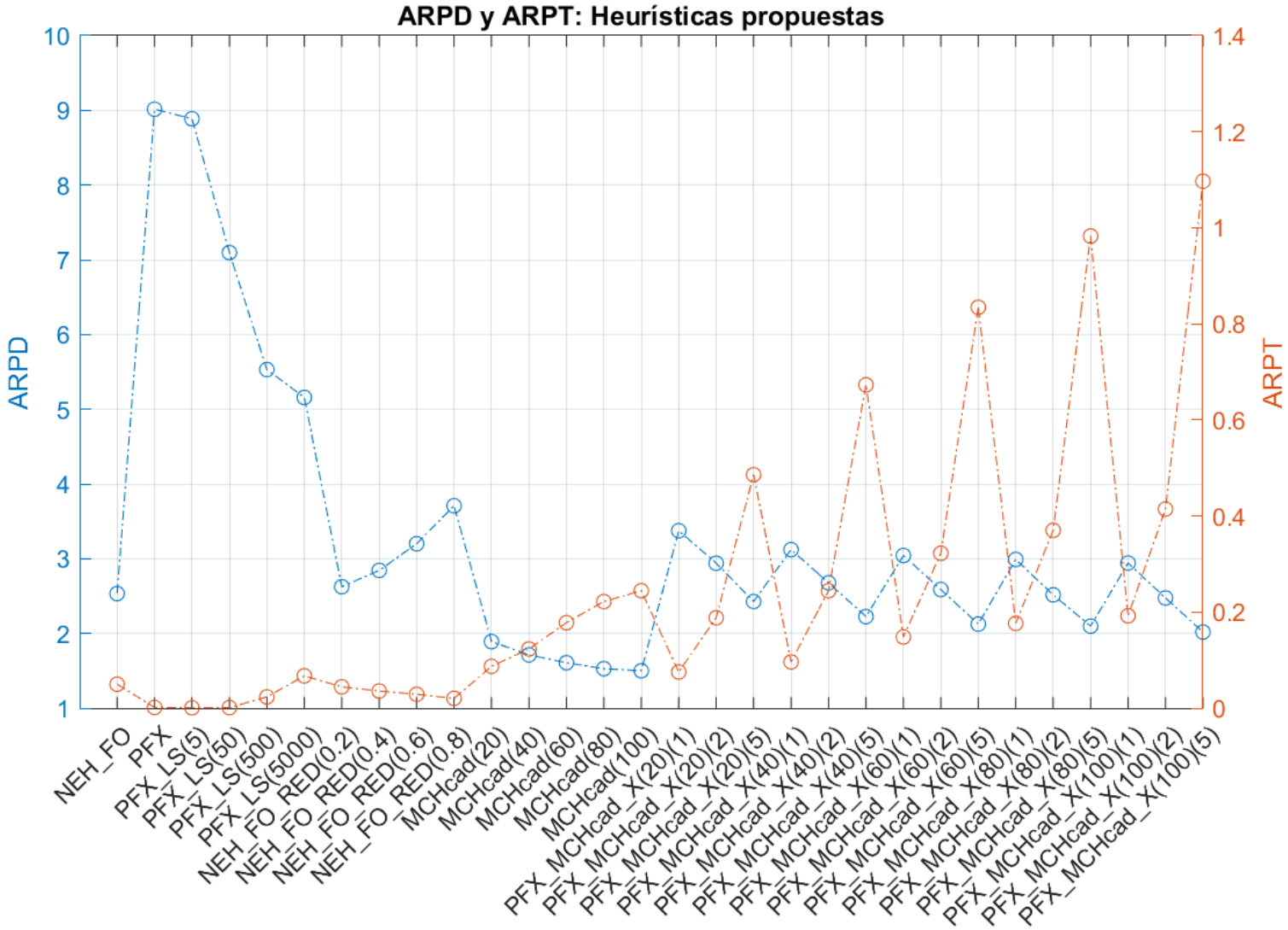

Figura 7.4. ARPD y ARPT (Heurísticas propuestas).

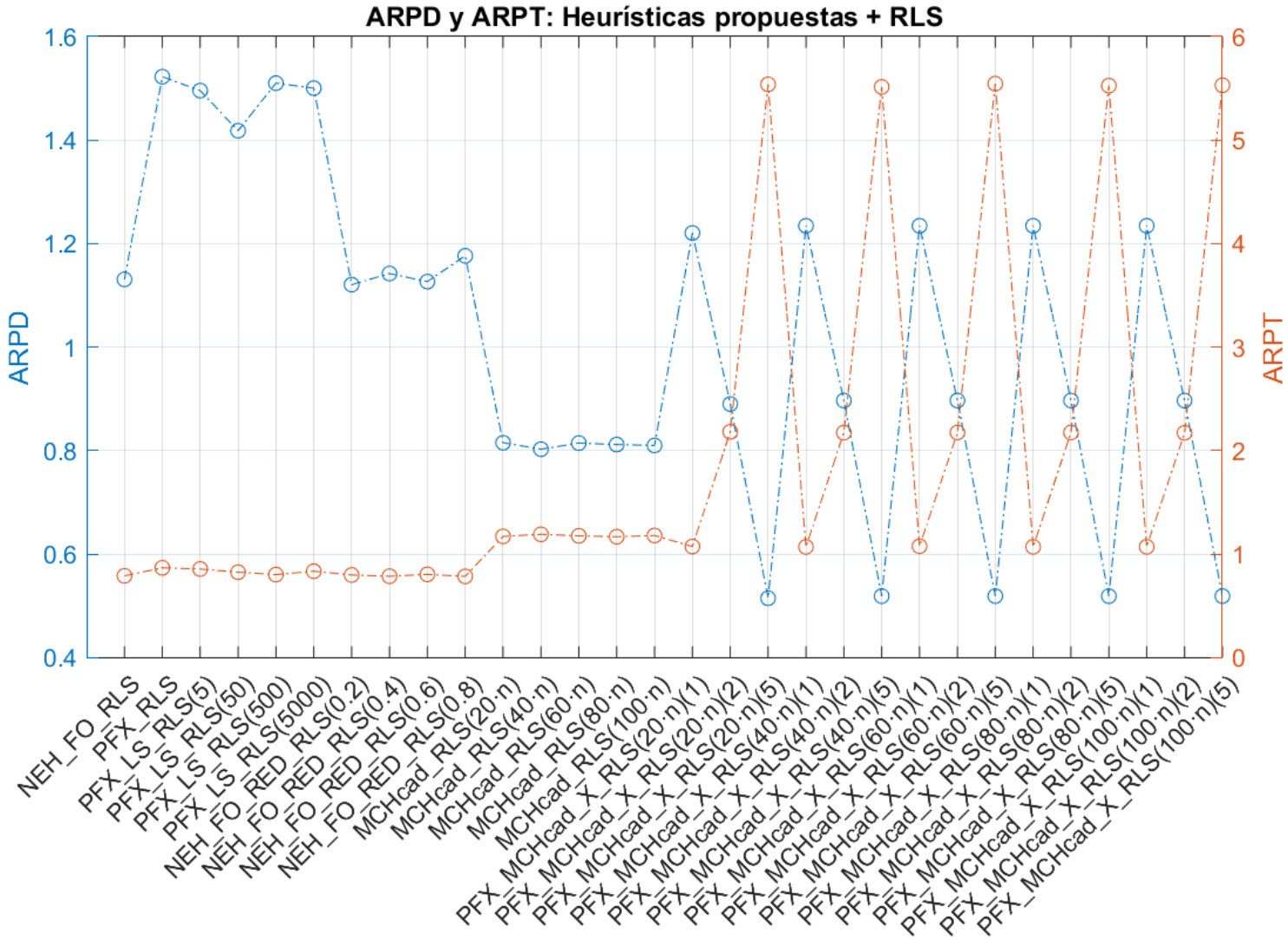

Figura 7.5. ARPD y ARPT (Heurísticas propuestas + RLS).

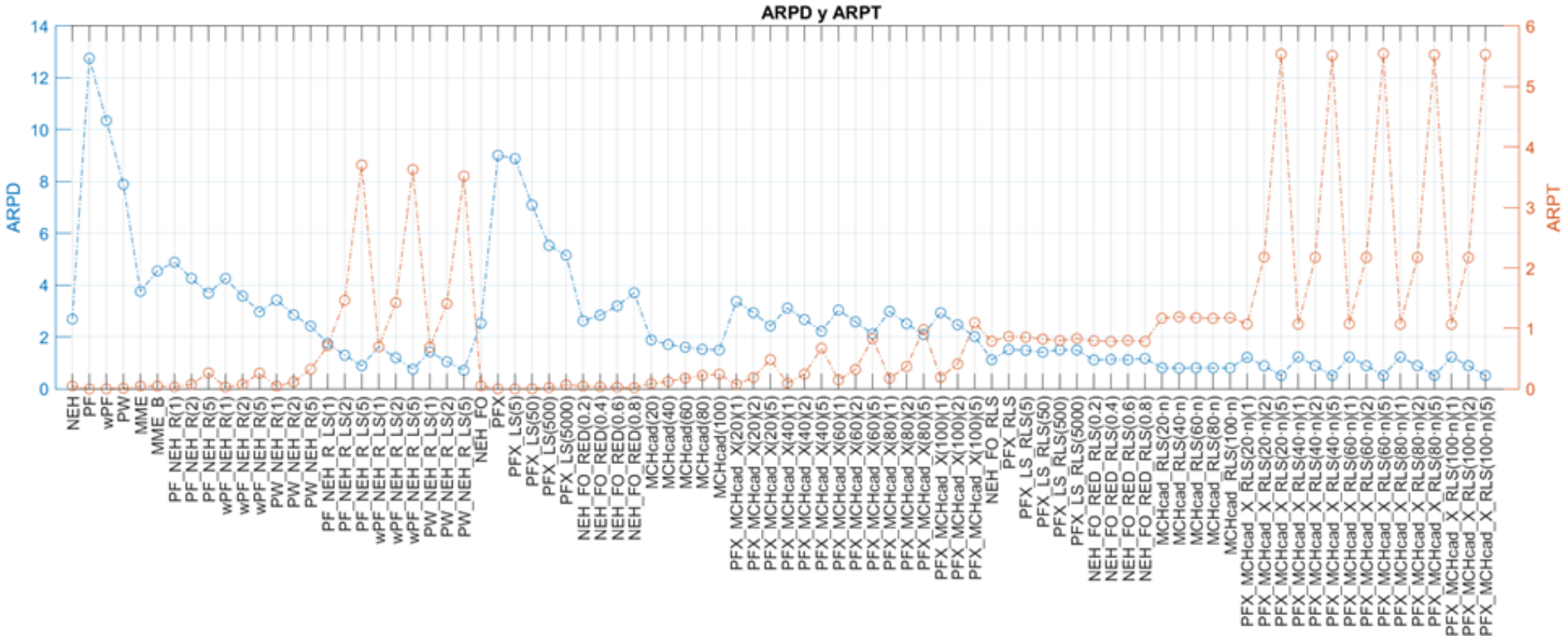

Figura 7.6. ARPD y ARPT (Todas las heurísticas).

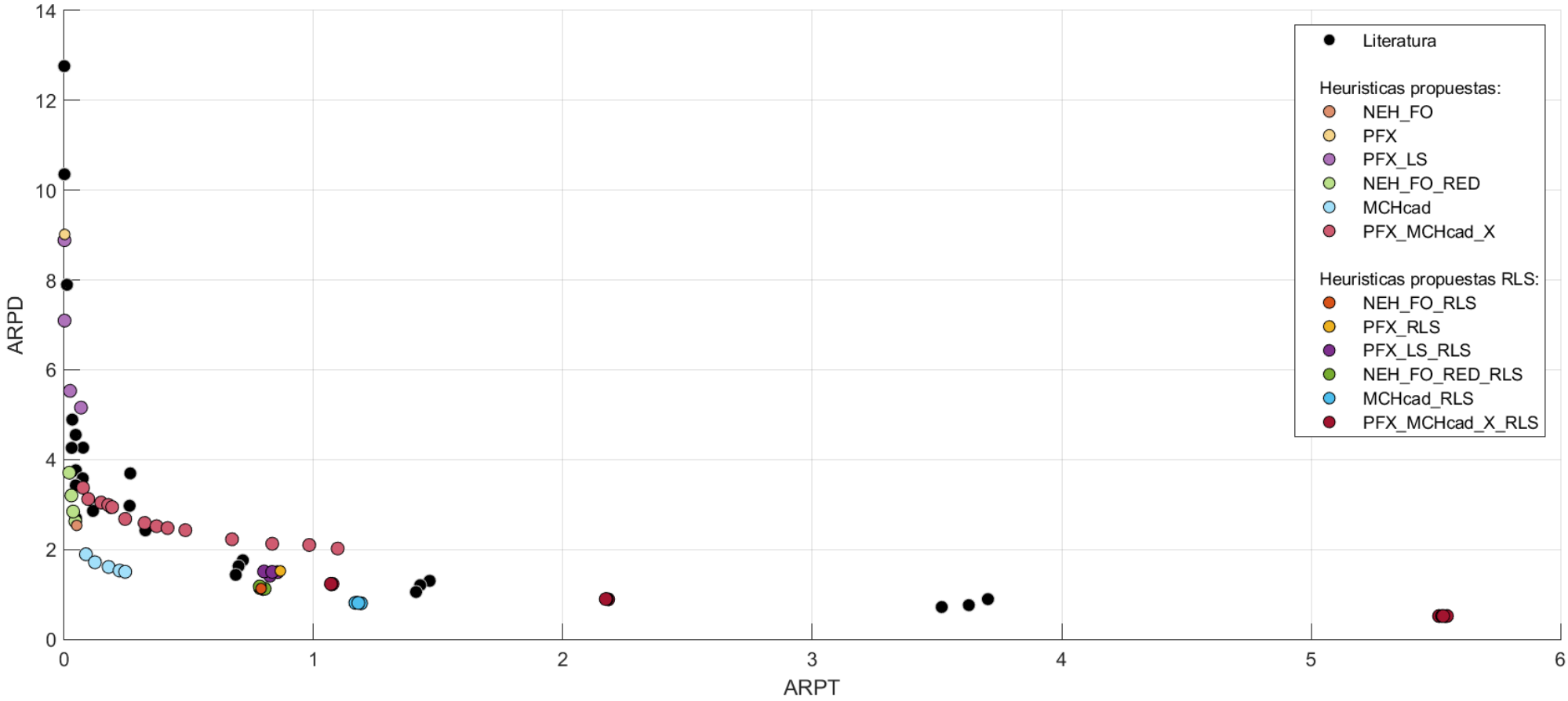

Figura 7.7. ARPD frente a ARPT (Todas las heurísticas).

Tabla 7.8. Heurísticas propuestas que mejoran a las heurísticas de la literatura

PARAM. MEJORA		PROPUESTAS					
		NEH_FO	NEH_FO_RED	PFX	PFX_LS	MCHcad	PFX_MCHcad_X
LITERATURA	NEH	X	0.2				
	PF			X	5 | 50		
	wPF			X	5 | 50		
	PW				50		
	MME	X	0.2 | 0.4 | 0.6 | 0.8				
	MME_B	X	0.2 | 0.4 | 0.6 | 0.8				
	PF_NEH_R(1)		0.6 | 0.8				
	PF_NEH_R(2)	X	0.2 | 0.4 | 0.6 | 0.8				20_1
	PF_NEH_R(5)	X	0.2 | 0.4 | 0.6			20 | 40 | 60 | 80 | 100	20_1 | 20_2 | 40_1 | 40_2 | 60_1 | 80_1 | 100_1
	wPF_NEH_R(1)		0.6 | 0.8				
	wPF_NEH_R(2)	X	0.2 | 0.4 | 0.6				
	wPF_NEH_R(5)	X	0.2 | 0.4			20 | 40 | 60 | 80 | 100	20_2 | 40_2 | 100_1
	PW_NEH_R(1)	X	0.2 | 0.4 | 0.6				
	PW_NEH_R(2)	X	0.2 | 0.4			20 | 40	
	PW_NEH_R(5)					20 | 40 | 60 | 80 | 100	
	PF_NEH_R_LS(1)					40 | 60 | 80 | 100	
	PF_NEH_R_LS(2)	(RLS) X	(RLS) 0.2 | 0.4 | 0.6 | 0.8			(RLS) 20·n | 40·n | 60·n | 80·n | 100·n	(RLS) 20·n_1 | 40·n_1 | 60·n_1 | 80·n_1 | 100·n_1
	PF_NEH_R_LS(5)					(RLS) 20·n | 40·n | 60·n | 80·n | 100·n	(RLS) 20·n_2 | 40·n_2 | 60·n_2 | 80·n_2 | 100·n_2
	wPF_NEH_R_ LS(1)					60 | 80 | 100	
	wPF_NEH_R_ LS(2)	(RLS) X	(RLS) 0.2 | 0.4 | 0.6 | 0.8			(RLS) 20·n | 40·n | 60·n | 80·n | 100·n	
	wPF_NEH_R_ LS(5)						
	PW_NEH_R_ LS(1)						
	PW_NEH_R_ LS(2)					(RLS) 20·n | 40·n | 60·n | 80·n | 100·n	
	PW_NEH_R_ LS(5)						

8 CONCLUSIONES

Para concluir, en este último capítulo del documento se comentan las conclusiones alcanzadas con la realización del proyecto. Adicionalmente, se comentan las posibles futuras líneas de investigación a las que da lugar.

8.1 Conclusiones

En primer lugar, en este documento se ha llevado a cabo un análisis del estado del arte sobre la programación de la producción en el entorno flowshop considerando capacidad limitada de almacenamiento entre las máquinas.

A continuación, se ha estudiado del problema objeto de este proyecto ($F_m \mid prmu, b_i \mid C_{max}$), concluyendo que por su complejidad para obtener el óptimo en un tiempo admisible, el procedimiento más adecuado de resolución es mediante métodos aproximados. Seis heurísticas constructivas diferentes han sido propuestas para resolver el problema en cuestión.

En vista de los resultados obtenidos, la combinación de una heurística constructiva con una búsqueda local de enfoque iterativo es una buena elección para resolver este problema. En concreto, la metodología propuesta para su resolución ha presentado mejoras en la calidad de las soluciones respecto a las heurísticas de la literatura para resolver este problema (NEH) o problemas similares (PF, wPF, PW, y sus combinaciones con NEH y RLS). Por lo general, existe relación entre calidad de la solución y tiempo computacional: las heurísticas desarrolladas proporcionan mejores soluciones en un mayor tiempo, mientras que las soluciones más rápidas son de menor calidad. Según el valor establecido para los parámetros de entrada a los algoritmos aproximados, la solución será mejor o peor, y requerirá de un mayor o menor tiempo. Valorando ambos criterios (tiempo y calidad), destaca el desempeño de las heurísticas MCH*cad*(100), PFX_LS(500) y NEH_FO_RED(0.2).

Respecto a las heurísticas PFX y PFX_LS, la amplia mejora de la calidad de la solución con respecto a la PF justifica la modificación realizada al indicador. Se comprueba que es beneficioso tener en cuenta la influencia del tiempo de holgura del buffer cuando hay tiempo ocioso en la siguiente máquina, además de ponderar los tiempos ociosos, de bloqueo y de holgura en función de la posición en el taller de la máquina donde se produzcan.

Además, evaluando los algoritmos propuestos, se puede concluir que el tamaño de la lista de movimientos prometedores de las heurísticas tiene una gran influencia sobre la calidad de la solución alcanzada. En concreto, se ha comprobado que es mejor establecer un tamaño de lista fijo que no dependa de los datos de cada instancia, y que, cuando se le aplica una búsqueda local al método con un tamaño de lista pequeño se incrementa el tiempo de ejecución desproporcionadamente, interesando tamaños de lista grandes en ese caso.

8.2 Futuras líneas de investigación

La secuenciación de tareas mediante heurísticas constructivas en un taller con la configuración abordada está poco explorada en la literatura, siendo posible enfocar futuros trabajos sobre esta área. Respecto a las posibles líneas de investigación desprendidas de este trabajo, se pueden encontrar:

- Desarrollar variantes de los métodos propuestos que incluyan mejoras en cuanto a la inserción de movimientos prometedores de forma más eficiente, o un mejor ajuste de los tamaños de las listas que contienen dichos movimientos.

- Convendría analizar si hay alguna alternativa para alcanzar mejores soluciones con los métodos propuestos que no incremente excesivamente el tiempo de ejecución, como ocurre al añadir a los métodos la búsqueda local RLS.

REFERENCIAS

Abiri, M. B., Zandieh, M., & Alem-Tabriz, A. (2009). A Tabu Search approach to hybrid flow shops scheduling with sequence-dependent setup times. In *Journal of Applied Sciences* (Vol. 9, Issue 9, pp. 1740–1745). https://doi.org/10.3923/jas.2009.1740.1745

Allahverdi, A. (2015). The third comprehensive survey on scheduling problems with setup times/costs. *European Journal of Operational Research*, *246*(2), 345–378. https://doi.org/10.1016/j.ejor.2015.04.004

Allahverdi, A., Ng, C. T., Cheng, T. C. E., & Kovalyov, M. Y. (2008). A survey of scheduling problems with setup times or costs. *European Journal of Operational Research*, *187*(3), 985–1032. https://doi.org/10.1016/j.ejor.2006.06.060

Companys Pascual, R. & Ribas Vila, I. (2012). *El curioso comportamiento del método de inserción de la heurística NEH en el problema Fm|block|Cmax. 6th International Conference on Industrial Engineering and Industrial Management. Vigo.*

Dutta, S. K., & Cunningham, A. A. (1975). Sequencing Two-Machine Flow-Shops With Finite Intermediate Storage. *Management Science*, *21*(9), 989–996. https://doi.org/10.1287/mnsc.21.9.989

Fernández-Viagas, V. (n.d.). *Organización de la producción. Entornos de fabricación.* http://www.organizaciondelaproduccion.com/entornos fabricacion.php

Fernandez-Viagas, V., Costa, A., & Framinan, J. M. (2020). Hybrid flow shop with multiple servers: A computational evaluation and efficient divide-and-conquer heuristics. *Expert Systems with Applications*, *153*, 113462. https://doi.org/10.1016/j.eswa.2020.113462

Fernandez-Viagas, V., & Framinan, J. M. (2014). On insertion tie-breaking rules in heuristics for the permutation flowshop scheduling problem. *Computers and Operations Research*, *45*, 60–67. https://doi.org/10.1016/j.cor.2013.12.012

Fernandez-Viagas, V., Molina-Pariente, J. M., & Framinan, J. M. (2018). New efficient constructive heuristics for the hybrid flowshop to minimise makespan: A computational evaluation of heuristics. *Expert Systems with Applications*, *114*, 345–356. https://doi.org/10.1016/j.eswa.2018.07.055

Framinan, J. M., Leisten, R., & Ruiz García, R. (2014). Manufacturing Scheduling Systems. In *Manufacturing Scheduling Systems*. https://doi.org/10.1007/978-1-4471-6272-8

Graham, R. L., Lawler, E. L., Lenstra, J. K., & Kan, A. H. G. R. (1979). Optimization and heuristic in deterministic sequencing and scheduling: a survey. *Annals of Discrete Mathematics*, *5*, 287–326. https://ac.els-cdn.com/S016750600870356X/1-s2.0-S016750600870356X-main.pdf?_tid=cbf345a3-808d-42df-9560-bf8a52069d0e&acdnat=1550949881_e9837bfdf6316caefaf71ae2f3779b10

Gupta, J. N. D., & Stafford, E. F. (2006). Flowshop scheduling research after five decades. *European Journal of Operational Research*, *169*(3), 699–711. https://doi.org/10.1016/j.ejor.2005.02.001

Johnson, S. M. (1954). With Setup Times Included. *Naval Research Logistics Quarterly*, *1*, 61–68.

Krajewski, L. J., King, B. E., Ritzman, L. P., & Wong, D. S. (1987). Kanban, Mrp, and Shaping the Manufacturing Environment. *Management Science*, *33*(1), 39–57. https://doi.org/10.1287/mnsc.33.1.39

Leisten, R. (1990). Flowshop sequencing problems with limited buffer storage. *International Journal of*

Production Research, *28*(11), 2085–2100. https://doi.org/10.1080/00207549008942855

Li, S., & Tang, L. (2005). A tabu search algorithm based on new block properties and speed-up method for permutation flow-shop with finite intermediate storage. *Journal of Intelligent Manufacturing*, *16*(4–5), 463–477. https://doi.org/10.1007/s10845-005-1658-1

Liu, B., Wang, L., & Jin, Y. H. (2008). An effective hybrid PSO-based algorithm for flow shop scheduling with limited buffers. *Computers and Operations Research*, *35*(9), 2791–2806. https://doi.org/10.1016/j.cor.2006.12.013

Liu, J., & Reeves, C. R. (2001). P Constructive and composite heuristic solutions to the P// C. *European Journal Of Operational Research*, *132*.

McCormick, S. T., Pinedo, M. L., Shenker, S., & Wolf, B. (1989). Sequencing in an assembly line with blocking to minimize cycle time. *Operations Research*, *37*(6), 925–935. https://doi.org/10.1287/opre.37.6.925

Miyata, H. H., & Nagano, M. S. (2019). The blocking flow shop scheduling problem: A comprehensive and conceptual review. *Expert Systems with Applications*, *137*, 130–156. https://doi.org/10.1016/j.eswa.2019.06.069

Moslehi, G., & Khorasanian, D. (2014). A hybrid variable neighborhood search algorithm for solving the limited-buffer permutation flow shop scheduling problem with the makespan criterion. *Computers and Operations Research*, *52*, 260–268. https://doi.org/10.1016/j.cor.2013.09.014

Nawaz, M., Enscore, E. E., & Ham, I. (1983). A heuristic algorithm for the m-machine, n-job flow-shop sequencing problem. *Omega*, *11*(1), 91–95. https://doi.org/10.1016/0305-0483(83)90088-9

Nowicki, E. (1999). Permutation flow shop with buffers: a tabu search approach. *European Journal of Operational Research*, *116*(1), 205–219. https://doi.org/10.1016/S0377-2217(98)00017-4

Pan, Q. K., & Wang, L. (2012). Effective heuristics for the blocking flowshop scheduling problem with makespan minimization. *Omega*, *40*(2), 218–229. https://doi.org/10.1016/j.omega.2011.06.002

Pan, Q. K., Wang, L., & Gao, L. (2011). A chaotic harmony search algorithm for the flow shop scheduling problem with limited buffers. *Applied Soft Computing Journal*, *11*(8), 5270–5280. https://doi.org/10.1016/j.asoc.2011.05.033

Pan, Q. K., Wang, L., Gao, L., & Li, W. D. (2011). An effective hybrid discrete differential evolution algorithm for the flow shop scheduling with intermediate buffers. *Information Sciences*, *181*(3), 668–685. https://doi.org/10.1016/j.ins.2010.10.009

Papadimitriou, C. H., & Kanellakis, P. C. (1980). Flowshop Scheduling with Limited Temporary Storage. *Journal of the ACM (JACM)*, *27*(3), 533–549. https://doi.org/10.1145/322203.322213

Pinedo, M. L. (2016). Scheduling: Theory, algorithms, and systems, fifth edition. In *Scheduling: Theory, Algorithms, and Systems, Fifth Edition*. https://doi.org/10.1007/978-3-319-26580-3

Qian, B., Wang, L., Huang, D. X., & Wang, X. (2009). An effective hybrid DE-based algorithm for flow shop scheduling with limited buffers. *International Journal of Production Research*, *47*(1), 1–24. https://doi.org/10.1080/00207540701528750

Ribas, I., Companys, R., & Tort-Martorell, X. (2011). An iterated greedy algorithm for the flowshop scheduling problem with blocking. *Omega*, *39*(3), 293–301. https://doi.org/10.1016/j.omega.2010.07.007

Ronconi, D. P. (2004). A note on constructive heuristics for the flowshop problem with blocking. *International Journal of Production Economics*, *87*(1), 39–48. https://doi.org/10.1016/S0925-5273(03)00065-3

Smutnicki, C. (1998). A two-machine permutation flow shop scheduling problem with buffers. *OR Spectrum*, *20*(4), 229–235. https://doi.org/10.1007/bf01539740

Storer, R. H., Wu, S. D., & Vaccari, R. (1992). New search spaces for sequencing problems with application to job shop scheduling. *Management Science*, *38*(10), 1495–1509. https://doi.org/10.1287/mnsc.38.10.1495

Taillard, E. (1993). Benchmarks for basic scheduling problems. *European Journal of Operational Research*, *64*(2), 278–285. https://doi.org/10.1016/0377-2217(93)90182-M

Wang, L., Zhang, L., & Zheng, D. Z. (2006). An effective hybrid genetic algorithm for flow shop scheduling with limited buffers. *Computers and Operations Research*, *33*(10), 2960–2971. https://doi.org/10.1016/j.cor.2005.02.028

Zanakis, S. H., & Evans, J. R. (1981). *Heuristic " Optimization ": Why , When , and How to Use It WHY , WHEN , AND HOW TO USE IT*. *August 2015*.

Zhao, F., Tang, J., Wang, J., & Jonrinaldi, J. (2014). An improved particle swarm optimisation with a linearly decreasing disturbance term for flow shop scheduling with limited buffers. *International Journal of Computer Integrated Manufacturing*, *27*(5), 488–499. https://doi.org/10.1080/0951192X.2013.814165

Heurísticas

H1: NEH_FO → NEH reemplazando FO (evalúa tiempos ocioso y de bloqueo para desempates)

```
private static void NEH_FO(int jobs, int machines, int[,] P, int[] B, System.IO.StreamWriter FicheroCSV1, System.IO.StreamWriter
FicheroCSV2)
{
        var timer = Stopwatch.StartNew(); //Iniciar contador de tiempo
        int[] secuencia = new int[jobs];
        for (int i = 0; i < jobs; i++) { secuencia[i] = i; }
        LPT(secuencia, P, machines); //Ordenar secuencia según tiempos proceso (de mayor a menor)

        //Búsqueda de la mejor secuencia
        int[] best = new int[1];
        best[0] = secuencia[0];
        double maximo = FO_idle_and_blocking_time(machines, B, P, secuencia);
        double limiteSup = maximo;

        for (int size = 1; size < jobs; size++)
        {
               int valor = secuencia[size]; //Trabajo a insertar
               int[] actual = new int[size + 1];
               Array.Copy(best, 0, actual, 0, size);
               actual[size] = valor;
               limiteSup = maximo; //Establecer límite superior
               for (int position = 0; position < size + 1; position++)
               {
                      //Insertar valor en position
                      int[] aux = new int[size + 1];
                      Array.Copy(best, 0, aux, 0, size);
                      aux.SetValue(valor, position);
                      Array.Copy(best, position, aux, position + 1, size - position);
```

```
                        //Evaluar FO
                        double valorFO = FO_idle_and_blocking_time(machines, B, P, aux);
                        if (valorFO <= limiteSup)
                        {
                                limiteSup = valorFO;
                                Array.Copy(aux, actual, size + 1); //Actualizar la secuencia actual solo si es mejor (o igual)
                        }
                }
                //Guardar la mejor secuencia hasta el momento
                Array.Resize(ref best, size + 1);
                Array.Copy(actual, best, size + 1);
        }

        int makespan = (int)limiteSup; //Cmax de la última mejor secuencia encontrada
        timer.Stop(); //Detener contador de tiempo
        double tiempoms = timer.ElapsedMilliseconds;
        double tiempo = tiempoms / 1000;

        //Escribir resultados en un fichero .csv
        FicheroCSV1.Write(makespan + ";");
        FicheroCSV2.Write(tiempo + ";");
}
```

H2: MCHcad → MCH con caducidad e Incremento = (FO – FObest)

```
public static void V6_2(int jobs, int machines, int[,] P, int[] B, int tam, int cad, System.IO.StreamWriter FicheroCSV1,
System.IO.StreamWriter FicheroCSV2)
{
      var timer = Stopwatch.StartNew(); //Iniciar contador de tiempo
      int[] secuencia = new int[jobs];
      for (int i = 0; i < jobs; i++) { secuencia[i] = i; }
      LPT(secuencia, P, machines); //Secuencia inicial

      //Búsqueda de la mejor secuencia
      int[] best = new int[1]; //Vector que almacena la mejor secuencia (parcial) encontrada en cada iteración
      best[0] = secuencia[0];
      double maximo = FO_idle_and_blocking_time(machines, B, P, secuencia); //Límite superior (valor de FO para sec. inicial LPT)
      double limiteSup = maximo;
      int lastinsert = 0;
      double[,] promise = new double[4, tam]; //Matriz que contiene los movimientos prometedores, el incremento de la FO, y el nº de
      veces que se ha probado cada movimiento
      int vacia = 10000;
      for (int i = 0; i < tam; i++)
      {
          promise[2, i] = vacia; //Rellenar matriz con valores altos
      }

      for (int size = 1; size < jobs; size++)
      {
          int[] candidatos = new int[size + 1]; //Vector que almacena los trabajos candidatos a movimientos prometedores
          double[] candidatos_FO = new double[size + 1]; //Vector que almacena los valores de FO de los movimientos candidatos
          int[] actual = new int[size + 1]; //Vector que guarda la secuencia actual
          Array.Copy(best, 0, actual, 0, size);
          actual[size] = secuencia[size]; //Trabajo a insertar
          limiteSup = maximo; //Establecer límite superior para Cmax y FO

          for (int position = 0; position < size + 1; position++)
          {
              int[] aux = new int[size + 1];
              Array.Copy(best, aux, size);
              aux.SetValue(secuencia[size], position); //Insertar trabajo "secuencia[size]" en "position"
              Array.Copy(best, position, aux, position + 1, size - position);

              double valorFO = FO_idle_and_blocking_time(machines, B, P, aux); //Calcular valor de la FO para la nueva secuencia
```

```
        if (valorFO < limiteSup) //Si es mejor:
        {
            limiteSup = valorFO; //Actualizar el límite superior de FO al nuevo calculado
            Array.Copy(aux, actual, size + 1); //Actualizar la secuencia actual
        }

        candidatos_FO[position] = valorFO; //Guardar el valor de la FO para la inserción actual
        if (Array.IndexOf(actual, actual[position]) == 0) //Guardar el trabajo anterior al insertado
        {
            candidatos[position] = -1; //Si no tiene ningún trabajo anterior, se le da el valor -1
        }
        else
        {
            candidatos[position] = aux[position - 1];
        }
    }

    Array.Sort(candidatos_FO, candidatos); //Ordenar los vectores que almacenan los trabajos anteriores y sus valores de FO,
    de menor a mayor valor de FO
    for (int n = 1; n < size + 1; n++)
    {
        double incremento = (candidatos_FO[n] - limiteSup); //Hallar diferencia con el mejor valor de la FO hasta el momento
        double max = 0;
        int indice_max = 0;

        if (lastinsert < tam)
        {
            promise[1, lastinsert] = secuencia[size]; //El trabajo a insertar
            promise[0, lastinsert] = candidatos[n]; //Su trabajo anterior
            promise[2, lastinsert] = incremento; //Su incremento respecto a la FO en ese momento
            lastinsert++;
        }
        else
        {
            for (int z = 0; z < tam; z++)
            {
                if (promise[2, z] >= max) //Hallar el movimiento almacenado en la matriz que proporciona un valor de la FO más
                alejado del mejor valor hasta el momento, y guardar:
                {
                    max = promise[2, z]; //Su valor de incremento de la FO
```

```
                    indice_max = z; //Su posición en la matriz (fila)
                }
            }
            if (max > incremento) //Cuando el incremento es mejor que el peor de los que contiene la matriz, sustituir:
            {
                promise[1, indice_max] = secuencia[size]; //El trabajo a insertar
                promise[0, indice_max] = candidatos[n]; //Su trabajo anterior
                promise[2, indice_max] = incremento; //Su incremento respecto a la FO en ese momento
                promise[3, indice_max] = 0; //El nº de veces que se ha probado se restablece a 0
            }
            else
            {
                n = size + 1; //Forzar salida del bucle cuando el incremento de FO no mejora a ninguno de la matriz
            }
        }
    }

    //Reinserción de los movimientos más prometedores en la secuencia actual
    for (int k = 0; k < tam; k++)
    {
        if (k == lastinsert) //Comprobar que esa fila de la matriz está rellena, si no lo está forzar salida del bucle
        {
            k = tam;
        }
        else
        {
            if (secuencia[size] != promise[1, k] && promise[2, k] != vacia) //Comprobar que no se vuelven a probar los
            movimientos que se acaban de añadir a la lista, y que la fila no esté vacía
            {
                //Repetir movimiento prometedor
                int[] nueva = new int[size + 1];
                int pos = Repetir_Movimiento_Prometedor_2(nueva, actual, size, (int)(promise[1, k]), (int)(promise[0, k]));

                //Calcular valor de FO para la nueva secuencia obtenida
                double nuevoValorFO = FO_idle_and_blocking_time(machines, B, P, nueva);

                promise[3, k]++; //Incrementar contador de nº de veces probado
                if (promise[3, k] == cad) //Si el movimiento se ha probado 'cad' veces, se elimina de la lista
                {
                    promise[3, k] = 0; //El nº de veces que se ha probado se restablece a 0
                    promise[2, k] = vacia; //Se elimina el valor del incremento de FO para que el movimiento salga de la lista
```

```
                    }
                    //Comparar FO obtenida con la mejor hasta el momento y si es mejor actualizar valor y secuencia actuales
                    if (nuevoValorFO < limiteSup)
                    {
                        //Intercambiar el movimiento probado de la matriz por el que se descarta
                        if (pos == 0)
                        {
                            promise[0, k] = -1;
                        }
                        else
                        {
                            promise[0, k] = actual[pos - 1];
                        }
                        promise[2, k] = limiteSup - nuevoValorFO; //Actualizar incremento de la FO
                        promise[3, k] = 0; //Establecer a 0 el nº de veces que se ha probado el movimiento

                        limiteSup = nuevoValorFO;
                        Array.Copy(nueva, actual, size + 1);
                    }
                }
            }
        }
        Array.Resize(ref best, size + 1);
        Array.Copy(actual, best, size + 1); //Guardar la mejor secuencia hasta el momento
    }

    if (limiteSup < maximo) //Cuando acaba la búsqueda se comprueba si el valor de la FO ha mejorado respecto al inicial
    {
         //Si ha mejorado, actualizar secuencia a la última mejor encontrada (si no mejora se devuelve la secuencia inicial LPT)
        Array.Copy(best, secuencia, jobs);
    }
    limiteSup = Cmax(machines, B, P, secuencia); //Valor makespan de la mejor secuencia
    timer.Stop(); //Detener contador de tiempo
    double tiempoms = timer.ElapsedMilliseconds;
    double tiempo = tiempoms / 1000;
    //Escribir resultados en un fichero .csv
    FicheroCSV1.Write(limiteSup + ";");
    FicheroCSV2.Write(tiempo + ";");
}
```

H3: NEH_FO_RED → NEH con FO y reduciendo el número de inserciones de forma variable

```
public static void NEH_RED_variable(int jobs, int machines, int[,] P, int[] B, double intervalo_reducido, System.IO.StreamWriter
FicheroCSV1, System.IO.StreamWriter FicheroCSV2)
{
	var timer = Stopwatch.StartNew(); //Iniciar contador de tiempo
	int[] secuencia = new int[jobs];
	for (int i = 0; i < jobs; i++) { secuencia[i] = i; }
	LPT(secuencia, P, machines); //Ordenar secuencia según tiempos proceso (de mayor a menor)

	//Búsqueda de la mejor secuencia
	int[] best = new int[1];
	best[0] = secuencia[0];
	double maximo = FO_idle_and_blocking_time(machines, B, P, secuencia);
	double limiteSup=maximo;

	double variable = 0;
	double intervalo_inf = 0.5;
	double intervalo_sup = 0.5;
	intervalo_reducido /= 2;
	for (int size = 1; size < jobs; size++)
	{
	    int valor = secuencia[size];//Trabajo a insertar
	    int[] actual = new int[size + 1];
	    Array.Copy(best, 0, actual, 0, size);
	    actual[size] = valor;
	    limiteSup = maximo; //Establecer límite superior
	    intervalo_inf -= variable;
	    intervalo_sup += variable;
	    variable = intervalo_reducido / (jobs - 1);

	    for (int position = 0; position <= (int)(intervalo_inf * size); position++)
	    {
	        //Insertar valor en position
	        int[] aux = new int[size + 1];
	        Array.Copy(best, 0, aux, 0, size);
	        aux.SetValue(valor, position);
	        Array.Copy(best, position, aux, position + 1, size - position);

	        //Evaluar FO
	        double valorFO = FO_idle_and_blocking_time(machines, B, P, aux);
```

```
                if (valorFO <= limiteSup)
                {
                    limiteSup = valorFO;
                    Array.Copy(aux, actual, size + 1);//Actualizar la actual solo si es mejor (o igual)
                }
            }

            for (int position = (int)(intervalo_sup * size); position < size + 1; position++)
            {
                if (position > (int)(intervalo_inf * size))
                {
                    //Insertar valor en position
                    int[] aux = new int[size + 1];
                    Array.Copy(best, 0, aux, 0, size);
                    aux.SetValue(valor, position);
                    Array.Copy(best, position, aux, position + 1, size - position);

                    //Evaluar FO
                    double valorFO = FO_idle_and_blocking_time(machines, B, P, aux);
                    if (valorFO <= limiteSup)
                    {
                        limiteSup = valorFO;
                        Array.Copy(aux, actual, size + 1);//Actualizar la actual solo si es mejor (o igual)
                    }
                }
            }
            //Guardar la mejor secuencia hasta el momento
            Array.Resize(ref best, size + 1);
            Array.Copy(actual, best, size + 1);
        }
        int makespan = (int)limiteSup; //Cmax de la última mejor secuencia encontrada
        timer.Stop(); //Detener contador de tiempo
        double tiempoms = timer.ElapsedMilliseconds;
        double tiempo = tiempoms / 1000;
        //Escribir resultados en un fichero .csv
        FicheroCSV1.Write(makespan + ";");
        FicheroCSV2.Write(tiempo + ";");

    }
```

H4: PFX → PF modificando el indicador para la selección de trabajos

```
public static claseSolucion PFX(int M, int N, int[,] tiemposProceso_ij, int[] buffer_i, double paramS, System.IO.StreamWriter
FicheroCSV1, System.IO.StreamWriter FicheroCSV2)
{
	var timer = Stopwatch.StartNew(); //Iniciar contador de tiempo
	//Seleccionar el primer trabajo (trabajo con menor tiempo de proceso total)
	int primerJob = 0;
	int tiempoPrimerJob = 0;
	bool primeraVez = true;
	for (int j = 0; j < N; j++)
	{
	    int tiempoAux = 0;
	    for (int i = 0; i < M; i++)
	    {
	        tiempoAux += tiemposProceso_ij[i, j];
	    }
	    if (primeraVez == true)
	    {
	        primeraVez = false;
	        primerJob = j;
	        tiempoPrimerJob = tiempoAux;
	    }
	    else if (tiempoAux < tiempoPrimerJob)
	    {
	        primerJob = j;
	        tiempoPrimerJob = tiempoAux;
	    }
	}

	//Calcular Ck_i inicial
	int[,] ct = new int[M + 1, N + 1]; //Matriz de tiempos de terminación
	ct[1, 1] = tiemposProceso_ij[0, primerJob];
	for (int i = 1; i < M; i++)
	{
	    ct[i + 1, 1] = ct[i, 1] + tiemposProceso_ij[i, primerJob];
	}
	int[] subSecuenciaAux = new int[1]; //Secuencia con el trabajo ya secuenciado
	int[] pi_1 = new int[N]; //Vector con todos los trabajos
	for (int j = 0; j < N; j++)
	{
```

```
    pi_1[j] = j;
}
int temp = pi_1[0];
pi_1[0] = primerJob;
pi_1[primerJob] = temp;
subSecuenciaAux[0] = primerJob;
for (int k = 1; k < N; k++)
{
    int[] secuenciaU = new int[N - k]; //Secuencia que contiene todos los trabajos que quedan por insertar
    for (int j = 0; j < N - k; j++)
    {
        secuenciaU[j] = pi_1[k + j];
    }
    double[] Xik_j; //Indicador: (idle time + blocking time + paramS*slack)/i
    int[,] C_ij; //Cmax en última posición para todos los trabajos
    Funcion_PFX(N, M, tiemposProceso_ij, buffer_i, ct, k, pi_1, secuenciaU, paramS, out Xik_j, out C_ij);
    //Hallar la tarea con menor Xi
    bool inicio = true;
    double Xi_Min = 0;
    int posicionMin = 0;
    for (int j = 0; j < N - k; j++)
    {
        if (inicio == true)
        {
            Xi_Min = Xik_j[j];
            posicionMin = j;
            inicio = false;
        }
        else if (Xik_j[j] < Xi_Min)
        {
            Xi_Min = Xik_j[j];
            posicionMin = j;
        }
    }
    temp = pi_1[k];
    pi_1[k] = secuenciaU[posicionMin];
    pi_1[k + posicionMin] = temp;
    //Calcular Ck_i
    for (int i = 0; i < M; i++)
    {
```

```
                ct[i + 1, k + 1] = C_ij[i, posicionMin];
            }
        }
        int[] secuencia = new int[N];
        for (int j = 0; j < N; j++)
        {
            secuencia[j] = pi_1[j];
        }
        claseSolucion solucion = new claseSolucion();
        solucion.secuenciaSolucion = new int[secuencia.Length];
        for (int i = 0; i < N; i++)
        {
            solucion.secuenciaSolucion[i] = secuencia[i];
        }
        int makespan = Cmax(M, buffer_i, tiemposProceso_ij, solucion.secuenciaSolucion); //Cmax de la mejor secuencia encontrada
        solucion.makespan = makespan;
        timer.Stop(); //Detener contador de tiempo
        double tiempoms = timer.ElapsedMilliseconds;
        double tiempo = tiempoms / 1000;
        solucion.CPUTime_Seconds = tiempo;
        //Escribir resultados en un fichero .csv
        FicheroCSV1.Write(makespan + ";");
        FicheroCSV2.Write(tiempo + ";");
        return solucion;
    }
```

H5: PFX_LS → PF modificando el indicador para la selección de trabajos y aplicando LS de movimientos prometedores

```
public static claseSolucion PFX_LS(int M, int N, int[,] tiemposProceso_ij, int[] buffer_i, int tam, double paramS,
System.IO.StreamWriter FicheroCSV1, System.IO.StreamWriter FicheroCSV2)
{
      var timer = Stopwatch.StartNew(); //Iniciar contador de tiempo
      //Seleccionar el primer trabajo (trabajo con menor tiempo de proceso total)
      int primerJob = 0;
      int tiempoPrimerJob = 0;
      bool primeraVez = true;
      for (int j = 0; j < N; j++)
      {
          int tiempoAux = 0;
          for (int i = 0; i < M; i++)
          {
              tiempoAux += tiemposProceso_ij[i, j];
          }
          if (primeraVez == true)
          {
              primeraVez = false;
              primerJob = j;
              tiempoPrimerJob = tiempoAux;
          }
          else if (tiempoAux < tiempoPrimerJob)
          {
              primerJob = j;
              tiempoPrimerJob = tiempoAux;
          }
      }

      //Crear la lista
      int lastinsert = 0;
      double[,] lista = new double[3, tam]; //Matriz que contiene los movimientos prometedores e incremento de indicador
      int vacia = 1000000;
      for (int i = 0; i < tam; i++)
      {
          lista[2, i] = vacia;
      }
```

```
//Calcular Ck_i inicial
int[,] ct = new int[M + 1, N + 1]; //Matriz de tiempos de terminación
ct[1, 1] = tiemposProceso_ij[0, primerJob];
for (int i = 1; i < M; i++)
{
    ct[i + 1, 1] = ct[i, 1] + tiemposProceso_ij[i, primerJob];
}
int[] subSecuenciaAux = new int[1]; //Secuencia con el trabajo ya secuenciado
int[] pi_1 = new int[N]; //Vector con todos los trabajos
for (int j = 0; j < N; j++)
{
    pi_1[j] = j;
}
int temp = pi_1[0];
pi_1[0] = primerJob;
pi_1[primerJob] = temp;
subSecuenciaAux[0] = primerJob;
for (int k = 1; k < N; k++)
{
    int[] secuenciaU = new int[N - k]; //Secuencia que contiene todos los trabajos que quedan por insertar
    for (int j = 0; j < N - k; j++)
    {
        secuenciaU[j] = pi_1[k + j];
    }
    double[] Xik_j; //Indicador: (idle time + blocking time + paramS*slack)/i
    int[,] C_ij; //Cmax en última posición para todos los trabajos
    Funcion_PFX(N, M, tiemposProceso_ij, buffer_i, ct, k, pi_1, secuenciaU, paramS, out Xik_j, out C_ij);
    //Hallar la tarea con menor Xi
    bool inicio = true;
    double Xi_Min = 0;
    int posicionMin = 0;
    for (int j = 0; j < N - k; j++)
    {
        if (inicio == true)
        {
            Xi_Min = Xik_j[j];
            posicionMin = j;
            inicio = false;
        }
        else if (Xik_j[j] < Xi_Min)
        {
```

```
            Xi_Min = Xik_j[j];
            posicionMin = j;
        }
    }
    temp = pi_1[k];
    pi_1[k] = secuenciaU[posicionMin];
    pi_1[k + posicionMin] = temp;

    //Calcular Ck_i
    for (int i = 0; i < M; i++)
    {
        ct[i + 1, k + 1] = C_ij[i, posicionMin];
    }

    //Almacenamos movimientos prometedores en la lista
    for (int j = 0; j < secuenciaU.Length; j++)
    {
        double incremento = (Xik_j[j] - Xi_Min); //Hallar la diferencia con el mejor valor de Xik_j hasta el momento (Xi_Min)
        double max = 0;
        int indice_max = 0;

        if (lastinsert < tam) //Si la lista no está completa se añade el movimiento
        {
            lista[1, lastinsert] = secuenciaU[j]; //El trabajo a insertar
            lista[0, lastinsert] = pi_1[k - 1]; //Su trabajo anterior
            lista[2, lastinsert] = incremento; //Su incremento de X
            lastinsert++;
        }
        else //Si la lista está completa se compara con el peor movimiento, y si es mejor se introduce en la lista
        {
            for (int z = 0; z < tam; z++)
            {
                if (lista[2, z] >= max) //Hallar el movimiento de la lista que proporcione el mayor valor de X
                {
                    max = lista[2, z]; //Su valor de incremento
                    indice_max = z; //Su posición en la matriz (fila)
                }
            }
            if (max > incremento) //Cuando el incremento es mejor que el peor de los que contiene la lista, sustituir:
            {
```

```
                lista[1, indice_max] = secuenciaU[j]; //El trabajo a insertar
                lista[0, indice_max] = pi_1[k - 1]; //Su trabajo anterior
                lista[2, indice_max] = incremento; //Su incremento de X en ese momento
            }
            else
            {
                j = secuenciaU.Length + 1; //Forzar salida del bucle cuando el incremento no mejore a ninguno de la lista
            }
        }
    }
}

//Reinserción de los movimientos más prometedores en la secuencia actual
double limiteSup = Cmax(M, buffer_i, tiemposProceso_ij, pi_1);
for (int w = 0; w < tam; w++)
{
    if (w == lastinsert) //Recorrer la lista hasta la última fila insertada, forzar salida del bucle para evitar filas vacías
    {
        w = tam;
    }
    else
    {
        if (lista[2, w] != vacia) //Comprobar que la fila de la lista está rellena
        {
            //Repetir movimiento prometedor
            int[] nueva = new int[N];
            Repetir_Movimiento_Prometedor(nueva, pi_1, N, (int)(lista[1, w]), (int)(lista[0, w]));
            double nuevoValorFO = Cmax(M, buffer_i, tiemposProceso_ij, nueva); //Calcular Cmax para la nueva secuencia
            if (nuevoValorFO < limiteSup) //Si mejora, actualizar valor y secuencia actuales
            {
                limiteSup = nuevoValorFO;
                Array.Copy(nueva, pi_1, N);
            }
        }
    }
}

claseSolucion solucion = new claseSolucion();
solucion.secuenciaSolucion = new int[pi_1.Length];
for (int i = 0; i < N; i++)
{
```

```
            solucion.secuenciaSolucion[i] = pi_1[i];
        }

        int makespan = Cmax(M, buffer_i, tiemposProceso_ij, solucion.secuenciaSolucion); //Cmax de la mejor secuencia encontrada
        solucion.makespan = makespan;

        timer.Stop(); //Detener contador de tiempo
        double tiempoms = timer.ElapsedMilliseconds;
        double tiempo = tiempoms / 1000;
        solucion.CPUTime_Seconds = tiempo;

        //Escribir resultados en un fichero .csv
        FicheroCSV1.Write(makespan + ";");
        FicheroCSV2.Write(tiempo + ";");
        return solucion;
    }
```

H6: PFX+MCHcad_x → Combinación PFX con MCHcad

```
public static claseSolucion PFX_MCHcad_x(int M, int N, int[,] tiemposProceso_ij, int[] buffer_i, int x, int lambda, int tam_V6, int
cad, double paramS, System.IO.StreamWriter FicheroCSV1, System.IO.StreamWriter FicheroCSV2)
{
       var timer = Stopwatch.StartNew(); //Iniciar contador de tiempo
       int job_limite = N - lambda;
       int mejorSolucion = 0;
       int[] mejorSecuencia = new int[N];
       //Seleccionar el primer trabajo (trabajo con menor tiempo de proceso total)
       int primerJob = 0;
       int[] tiempoAux = new int[N];
       double[,] Elements = new double[2, N];
       int[] ordenTareas = new int[N];
       for (int j = 0; j < N; j++)
       {
           for (int i = 0; i < M; i++)
           {
               tiempoAux[j] = tiempoAux[j] + tiemposProceso_ij[i, j];
           }
           Elements[0, j] = j;
           Elements[1, j] = tiempoAux[j];
       }
       //Ordenar trabajos
       double[,] ElementsSalida = Quicksort(Elements, 0, N - 1);
       for (int j = 0; j < N; j++)
       {
           ordenTareas[j] = (int)ElementsSalida[0, j];
       }
       for (int ix = 0; ix < x; ix++)
       {
           //Hallar secuencia PF
           primerJob = ordenTareas[ix];
           //Calcular Ck_i inicial
           int[,] ct = new int[M + 1, N + 1];
           ct[1, 1] = tiemposProceso_ij[0, primerJob];
           for (int i = 1; i < M; i++)
           {
               ct[i + 1, 1] = ct[i, 1] + tiemposProceso_ij[i, primerJob];
           }
           int[] subSecuenciaAux = new int[1];
```

```
int[] pi_1 = new int[N];
for (int j = 0; j < N; j++)
{
    pi_1[j] = j;
}
int temp = pi_1[0];
pi_1[0] = primerJob;
pi_1[primerJob] = temp;
subSecuenciaAux[0] = primerJob;
for (int k = 1; k < N; k++)
{
    int[] secuenciaU = new int[N - k];
    for (int j = 0; j < N - k; j++)
    {
        secuenciaU[j] = pi_1[k + j];
    }
    double[] Xik_j;
    int[,] C_ij;
    Funcion_PFX(N, M, tiemposProceso_ij, buffer_i, ct, k, pi_1, secuenciaU, paramS, out Xik_j, out C_ij);
    //Hallar tarea con menor Xi
    bool inicio = true;
    double Xi_Min = 0;
    int posicionMin = 0;
    for (int j = 0; j < N - k; j++)
    {
        if (inicio == true)
        {
            Xi_Min = Xik_j[j];
            posicionMin = j;
            inicio = false;
        }
        else if (Xik_j[j] < Xi_Min)
        {
            Xi_Min = Xik_j[j];
            posicionMin = j;
        }
    }
    temp = pi_1[k];
    pi_1[k] = secuenciaU[posicionMin];
    pi_1[k + posicionMin] = temp;
```

```
    //Calcular Ck_i
    for (int i = 0; i < M; i++)
    {
        ct[i + 1, k + 1] = C_ij[i, posicionMin];
    }
}

int[] secuencia = new int[N];
for (int j = 0; j < N; j++)
{
    secuencia[j] = pi_1[j];
}
int solAux = Cmax(M, buffer_i, tiemposProceso_ij, secuencia);
if (ix == 0)
{
    mejorSolucion = solAux;
    for (int i = 0; i < N; i++)
    {
        mejorSecuencia[i] = secuencia[i];
    }
}
else if (solAux < mejorSolucion)
{
    mejorSolucion = solAux;
    for (int i = 0; i < N; i++)
    {
        mejorSecuencia[i] = secuencia[i];
    }
}

//Aplicar MCHcad a partit del trabajo límite
MCHcad_para_mitad_PF(job_limite, M, pi_1, tiemposProceso_ij, buffer_i, tam_V6, cad);

double makespan = Cmax(M, buffer_i, tiemposProceso_ij, pi_1);
if (makespan < mejorSolucion)
{
    mejorSolucion = (int)(makespan);
    for (int i = 0; i < N; i++)
    {
        mejorSecuencia[i] = pi_1[i];
    }
```

```
            }
        }
        claseSolucion solucion = new claseSolucion();
        solucion.secuenciaSolucion = new int[mejorSecuencia.Length];
        int bestFlowTime = mejorSolucion;
        for (int i = 0; i < N; i++)
        {
            solucion.secuenciaSolucion[i] = mejorSecuencia[i];
        }
        solucion.makespan = bestFlowTime;

        timer.Stop(); //Detener contador de tiempo
        double tiempoms = timer.ElapsedMilliseconds;
        double tiempo = tiempoms / 1000;
        solucion.CPUTime_Seconds = tiempo;

        //Escribir resultados en un fichero .csv
        FicheroCSV1.Write(makespan + ";");
        FicheroCSV2.Write(tiempo + ";");
        return solucion;

    }
```

Búsqueda local aplicada a las heurísticas

```
public static void RLS_FO(int M, int[,] tiemposProceso_ij, int[] B, int[] secuenciaInicial, double FOInicial, out int[]
secuenciaFinal, out int FOFinal)
{
       double minimoFO = FOInicial;
       double minimoFOLocal = 0;
       int longitudSecuencia = secuenciaInicial.Length;
       int N = longitudSecuencia;
       int[] subSecuencia = new int[longitudSecuencia];
       secuenciaFinal = new int[longitudSecuencia];
       Array.Copy(secuenciaInicial, secuenciaFinal, longitudSecuencia);
       Array.Copy(secuenciaInicial, subSecuencia, longitudSecuencia);
       int[] subSecuenciaAux;
       int numJobs = longitudSecuencia - 1;
       subSecuenciaAux = new int[numJobs];
       int i = 0;
       int h = 0;
       int[] secuenciaAuxiliar = new int[longitudSecuencia];
       double FO;
       while (i < longitudSecuencia)
       {
           int j = h % longitudSecuencia;
           int jAuxiliar = Array.IndexOf(subSecuencia, secuenciaInicial[j]);
           //Extraer trabajo de la secuencia
           int tareaElegida = subSecuencia[jAuxiliar];
           for (int j_aux2 = 0; j_aux2 < N - 1; j_aux2++)
           {
               if (j_aux2 >= jAuxiliar)
               {
                   subSecuenciaAux[j_aux2] = subSecuencia[j_aux2 + 1];
               }
               else
               {
                   subSecuenciaAux[j_aux2] = subSecuencia[j_aux2];
               }
           }
           int mejorPosicion = 0;
           bool primeraVez = true;
           for (int j_aux = 0; j_aux < longitudSecuencia; j_aux++)
           {
```

```
        if (j_aux != jAuxiliar)
        {
            //Insertar trabajo en la secuencia
            for (int j_aux2 = 0; j_aux2 < N; j_aux2++)
            {
                if (j_aux2 == j_aux)
                {
                    secuenciaAuxiliar[j_aux2] = tareaElegida;
                }
                else if (j_aux2 < j_aux)
                {
                    secuenciaAuxiliar[j_aux2] = subSecuenciaAux[j_aux2];
                }
                else if (j_aux2 > j_aux)
                {
                    secuenciaAuxiliar[j_aux2] = subSecuenciaAux[j_aux2 - 1];
                }
            }
            FO = FO_idle_and_blocking_time(M, B, tiemposProceso_ij, secuenciaAuxiliar);
            if (primeraVez == true)
            {
                minimoFOLocal = FO;
                mejorPosicion = j_aux;
                primeraVez = false;
            }
            else
            {
                if (FO < minimoFOLocal)
                {
                    mejorPosicion = j_aux;
                    minimoFOLocal = FO;
                }
            }
        }
    }
    if (minimoFOLocal < minimoFO)
    {
        //Crear la nueva secuencia formada por la posición del mejorMakespan
        for (int j_aux2 = 0; j_aux2 < longitudSecuencia; j_aux2++)
        {
```

```
                    if (j_aux2 == mejorPosicion)
                    {
                        subSecuencia[j_aux2] = tareaElegida;
                    }
                    else if (j_aux2 < mejorPosicion)
                    {
                        subSecuencia[j_aux2] = subSecuenciaAux[j_aux2];
                    }
                    else if (j_aux2 > mejorPosicion)
                    {
                        subSecuencia[j_aux2] = subSecuenciaAux[j_aux2 - 1];
                    }
                }
                minimoFO = minimoFOLocal;
                i = 0;
            }
            else
            {
                i++;
            }
            h++;
        }
        FOFinal = (int)minimoFO;
        Array.Copy(subSecuencia, secuenciaFinal, longitudSecuencia);
    }
```

Métodos usados en las heurísticas

LPT

```
public static void LPT(int[] sec, int[,] P, int machines)
{
      int[] tiemposProceso = new int[sec.Length];
      //Sumar columnas (tiempo proceso total de un trabajo en todas las máquinas)
      SumaCol(P, tiemposProceso, machines);
      //Ordenar decreciente
      Array.Sort(tiemposProceso, sec);
      Array.Reverse(sec);
}
```

SumaCol

```
public static void SumaCol(int[,] matriz, int[] vector_suma, int filas)
{
      for (int i = 0; i < vector_suma.Length; i++)
      {
          for (int j = 0; j < filas; j++)
          {
              vector_suma[i] = vector_suma[i] + matriz[j, i];
          }
      }
}
```

FO_idle_and_blocking_time

```
public static double FO_idle_and_blocking_time(int m, int[] B, int[,] P, int[] sec)
{
      int n = sec.Length;
      int[,] ct = new int[m + 1, n + 1]; //Matriz tiempos de terminación de cada trabajo en cada máquina
      CompletionTime(ct, m, B, P, sec);
      return ct[m, n] + (Idle_and_blocking_Time(m, sec, P, ct) / 1000000);
}
```

CompletionTime

```
public static void CompletionTime(int[,] ct, int m, int[] B, int[,] P, int[] sec)
{
      int n = sec.Length;
      //Calcular tiempo de fin de cada trabajo en cada máquina
      for (int j = 1; j <= n; j++)
      {
          ct[0, j] = 0;//Fila ficticia
          for (int i = 1; i <= m; i++)
          {
              ct[i, 0] = 0; //Columna ficticia
              int k;//Índice para cuando no hay que comprobar el término del buffer
              if (j - 1 - B[i - 1] <= 0 || i == m) k = 0;
              else
              {
                  k = (ct[i + 1, j - 1 - B[i - 1]]) - P[i, sec[j - 2 - B[i - 1]]];
              }

              //Fórmula: ct[i,j] = max ( ct[i-1,j] , ct[i,j-1] , ct[i+1,j-1-B[i]])-P[i+1,j-1-bi] )  +  p[i,j]
              if (ct[i - 1, j] >= ct[i, j - 1] && ct[i - 1, j] >= k) //Si el mayor de los tres términos es el primero: ct[i-1,j]
              {
                  //T.Fin del trab. j en la maq. i = T.Fin del trabajo j en la máquina anterior (i-1) + T.Proceso[i,j]
                  ct[i, j] = ct[i - 1, j] + P[i - 1, sec[j - 1]];
              }
              else if (ct[i, j - 1] >= ct[i - 1, j] && ct[i, j - 1] >= k) //Si el mayor término es el segundo: ct[i,j-1]
              {
                  //T.Fin del trab. j en la maq. i = T.Fin del trabajo anteriormente procesado (j-1) en maq i + T.Proceso[i,j]
                  ct[i, j] = ct[i, j - 1] + P[i - 1, sec[j - 1]];
              }
              else //Si el mayor de los tres términos es el tercero: ct[i+1,j-1-bi)-P[i+1,j-1-bi] (= k)
              {
                  //T.Fin del trab. j en la maq. i = T.Fin del trabajo "anterior-capacidad buffer" (j-1-bi) en la siguiente maq
                  (i+1) + T.Proceso[i,j]
                  ct[i, j] = k + P[i - 1, sec[j - 1]];
              }
          }
      }
}
```

Cmax

```
public static int Cmax(int m, int[] B, int[,] P, int[] sec)
{
      int n = sec.Length;
      int[,] ct = new int[m + 1, n + 1]; //Matriz tiempos de terminación de cada trabajo en cada máquina
      CompletionTime(ct, m, B, P, sec);
      return ct[m, n]; //Makespan
}
```

Funcion_PFX

```
private static void Funcion_PFX(int n, int m, int[,] P, int[] B, int[,] ct, int iteracion, int[] sec, int[] secuenciaU, double
paramS, out double[] Xik_j, out int[,] C_ij)
{
      int posActual = iteracion + 1;//Posición a la que corresponde con la referencia que tiene más 1
      //Tiempo finalización del trabajo secuenciaU[j] insertado al final
      C_ij = new int[m, n - iteracion];
      Xik_j = new double[n - iteracion];
      //Calcular tiempo de fin de cada trabajo en cada máquina
      for (int j = 0; j < n - iteracion; j++)
      {
          ct[0, posActual] = 0;//Fila ficticia
          for (int i = 1; i <= m; i++)
          {
              ct[i, 0] = 0; //Columna ficticia
              int k;//Índice para cuando no hay que comprobar el término del buffer
              if (posActual - 1 - B[i - 1] <= 0 || i == m) k = 0;
              else
              {
                  k = (ct[i + 1, posActual - 1 - B[i - 1]]) - P[i, sec[posActual - 2 - B[i - 1]]];
              }
              if (ct[i - 1, posActual] >= ct[i, posActual - 1] && ct[i - 1, posActual] >= k)
              {
                  ct[i, posActual] = ct[i - 1, posActual] + P[i - 1, secuenciaU[j]];
              }
              else if (ct[i, posActual - 1] >= ct[i - 1, posActual] && ct[i, posActual - 1] >= k)
              {
                  ct[i, posActual] = ct[i, posActual - 1] + P[i - 1, secuenciaU[j]];
```

```
                }
                else
                {
                    ct[i, posActual] = k + P[i - 1, secuenciaU[j]];
                }
                C_ij[i - 1, j] = ct[i, posActual];
                //Calcular tiempo ocioso
                double idleTime = 0;
                if (i != 1)
                {
                    idleTime = IdleTimeCalculator(posActual, i, ct, B[i - 1], P[i - 1, secuenciaU[j]]);
                }
                //Calcular tiempo de bloqueo y holgura
                double[] slack = new double[1];
                slack[0] = 0;
                double blockingtime = BlockingTimeCalculator_s(posActual, i, ct, B[i - 1], slack);
                if (idleTime == 0) slack[0] = 0; //La holgura solo se tiene en cuenta cuando hay idle time
                Xik_j[j] += (double)(idleTime + blockingtime + paramS * slack[0]) / (double)(i);
            }
        }
}
```

BlockingTimeCalcular_s

```
private static double BlockingTimeCalculator_s(int k, int maquina, int[,] cT, int buffer, double[] slack)
{
      //Además del tiempo de bloqueo, también devuelve la holgura (slack)
      if ((k - 1 - buffer) > 0)
      { // Hasta que los trabajos quepan en el buffer, se rellena
          double blockTime = cT[maquina + 1, k - 1 - buffer] - cT[maquina, k];
          if (blockTime > 0)
          { // Si no hay bloqueo, el tiempo será 0 o negativo, en caso de que exista bloqueo, se devuelve
              return blockTime;
          }
          else slack[0] = -blockTime;
          return 0;
      }
      return 0;
}
```

IdleTimeCalculator

```
private static double IdleTimeCalculator(int k, int maquina, int[,] cT, int buffer, int tiempoProceso)
{
      return cT[maquina, k] - cT[maquina, k - 1] - BlockingTimeCalculator(k - 1, maquina, cT, buffer) - tiempoProceso;
}
```

MCHcad_para_mitad_PF

```
public static void MCHcad_para_mitad_PF(int job_limite, int machines, int[] secuencia, int[,] P, int[] B, int tam, int cad)
{
      int jobs = secuencia.Length;

      //Búsqueda de la mejor secuencia
      int[] best = new int[job_limite];
      Array.Copy(secuencia, best, job_limite);
      double maximo = FO_idle_and_blocking_time(machines, B, P, secuencia);
      double limiteSup;

      int lastinsert = 0;
      double[,] promise = new double[4, tam]; //Matriz que contiene los movimientos prometedores, el incremento de la FO, y el nº de
      veces que se ha probado cada movimiento
      int vacia = 10000;
      for (int i = 0; i < tam; i++)
      {
          promise[2, i] = vacia; //Rellenar matriz con valores altos
      }


      for (int size = job_limite; size < jobs; size++)
      {
          int[] candidatos = new int[size + 1]; //Vector que almacena los trabajos candidatos a movimientos prometedores
          double[] candidatos_FO = new double[size + 1]; //Vector que almacena los valores de FO de los movimientos candidatos
          int[] actual = new int[size + 1]; //Vector que guarda la secuencia actual
          Array.Copy(best, 0, actual, 0, size);
          actual[size] = secuencia[size]; //Trabajo a insertar
          limiteSup = maximo; //Establecer límite superior para Cmax y FO

          for (int position = 0; position < size + 1; position++)
```

```
{
    int[] aux = new int[size + 1];
    Array.Copy(best, aux, size);
    aux.SetValue(secuencia[size], position); //Insertar trabajo "secuencia[size]" en "position"
    Array.Copy(best, position, aux, position + 1, size - position);

    double valorFO = FO_idle_and_blocking_time(machines, B, P, aux); //Calcular valor de la FO para la nueva secuencia
    if (valorFO < limiteSup) //Si es mejor:
    {
        limiteSup = valorFO; //Actualizar el límite superior de FO al nuevo calculado
        Array.Copy(aux, actual, size + 1); //Actualizar la secuencia actual
    }

    candidatos_FO[position] = valorFO; //Guardar el valor de la FO para la inserción actual
    if (Array.IndexOf(actual, actual[position]) == 0) //Guardar el trabajo anterior al insertado
    {
        candidatos[position] = -1; //Si no tiene ningún trabajo anterior, se le da el valor -1
    }
    else
    {
        candidatos[position] = aux[position - 1];
    }
}

Array.Sort(candidatos_FO, candidatos); //Ordenar los vectores que almacenan los trabajos anteriores y sus valores de FO, de
menor a mayor valor de FO

for (int n = 1; n < size + 1; n++)
{
    double incremento = (candidatos_FO[n] - limiteSup); //Hallar diferencia con el mejor valor de la FO hasta el momento
    double max = 0;
    int indice_max = 0;

    if (lastinsert < tam) //Si la lista aún no está completa, introducir movimiento
    {
        promise[1, lastinsert] = secuencia[size]; //El trabajo a insertar
        promise[0, lastinsert] = candidatos[n]; //Su trabajo anterior
        promise[2, lastinsert] = incremento; //Su incremento respecto a la FO en ese momento
        lastinsert++;
    }
    else //Si la lista ya está completa, introducir movimiento sólo si mejora al peor movimiento de la lista
```

```
        {
            for (int z = 0; z < tam; z++)
            {
                if (promise[2, z] >= max) //Hallar el movimiento de la lista con mayor valor del incremento, y guardar:
                {
                    max = promise[2, z]; //Su valor de incremento de la FO
                    indice_max = z; //Su posición en la matriz (fila)
                }
            }
            if (max > incremento) //Cuando el incremento es mejor que el peor de los que contiene la matriz, sustituir:
            {
                promise[1, indice_max] = secuencia[size]; //El trabajo a insertar
                promise[0, indice_max] = candidatos[n]; //Su trabajo anterior
                promise[2, indice_max] = incremento; //Su incremento respecto a la FO en ese momento
                promise[3, indice_max] = 0; //El nº de veces que se ha probado se restablece a 0
            }
            else
            {
                n = size + 1; //Forzar salida del bucle cuando el incremento de la FO no mejora a ninguno de la lista
            }
        }
    }

    //Reinserción de los movimientos más prometedores en la secuencia actual
    for (int k = 0; k < tam; k++)
    {
        if (k == lastinsert) //Comprobamos que esa fila de la matriz está rellena, si no lo está forzar salida del bucle
        {
            k = tam;
        }
        else
        {
            if (secuencia[size] != promise[1, k] && promise[2, k] != vacia) //Comprobar que no se vuelven a probar los
            movimientos que se acaban de añadir a la lista, y que la fila no esté vacía
            {
                //Repetir movimiento prometedor
                int[] nueva = new int[size + 1];
                int pos = Repetir_Movimiento_Prometedor_2(nueva, actual, size, (int)(promise[1, k]), (int)(promise[0, k]));

                //Calcular valor de FO para la nueva secuencia obtenida
```

```
                    double nuevoValorFO = FO_idle_and_blocking_time(machines, B, P, nueva);
                    promise[3, k]++; //Incrementar contador de nº de veces probado
                    if (promise[3, k] == cad) //Si el movimiento se ha probado 'cad' veces, se elimina de la lista
                    {
                        promise[3, k] = 0; //El nº de veces que se ha probado se restablece a 0
                        promise[2, k] = vacia; //Se elimina el valor del incremento de FO para que el movimiento salga de la lista
                    }
                    //Comparar FO obtenida con la mejor hasta el momento y si es mejor actualizar valor y secuencia actuales
                    if (nuevoValorFO < limiteSup)
                    {
                        //Intercambiar el movimiento probado de la matriz por el que se descarta
                        if (pos == 0)
                        {
                            promise[0, k] = -1;
                        }
                        else
                        {
                            promise[0, k] = actual[pos - 1];
                        }
                        promise[2, k] = limiteSup - nuevoValorFO; //Actualizar incremento de la FO
                        promise[3, k] = 0; //Establecer a 0 el nº de veces que se ha probado el movimiento

                        limiteSup = nuevoValorFO;
                        Array.Copy(nueva, actual, size + 1);
                    }
                }
            }
        }
        Array.Resize(ref best, size + 1);
        Array.Copy(actual, best, size + 1); //Guardar la mejor secuencia hasta el momento
    }
    //Actualizar secuencia a la última mejor encontrada
    Array.Copy(best, secuencia, jobs);
}
```

Quicksort

```
public static double[,] Quicksort(double[,] elements, int left, int right)
{
      int i = left, j = right;
      double pivot = elements[1, (left + right) / 2];
      while (i <= j)
      {
          while (elements[1, i].CompareTo(pivot) < 0)
          {
              i++;
          }
          while (elements[1, j].CompareTo(pivot) > 0)
          {
              j--;
          }

          if (i <= j)
          {
              //Swap
              double tmp0 = elements[0, i];
              double tmp1 = elements[1, i];
              elements[0, i] = elements[0, j];
              elements[1, i] = elements[1, j];
              elements[0, j] = tmp0;
              elements[1, j] = tmp1;
              i++;
              j--;
          }
      }
      //Recursive calls
      if (left < j)
      {
          Quicksort(elements, left, j);
      }
      if (i < right)
      {
          Quicksort(elements, i, right);
      }
```

```
        return elements;
}
```

Repetir_Movimiento_Prometedor

```
public static void Repetir_Movimiento_Prometedor(int[] nueva, int[] actual, int size, int job, int job_before)
{
        Array.Copy(actual, nueva, size);
        int pos_origen = Array.IndexOf(nueva, job); //Obtener la posición en la secuencia actual del trabajo insertado en la iteración
        anterior (job)
        Array.Copy(actual, pos_origen + 1, nueva, pos_origen, size - 1 - pos_origen); //Eliminar dicho trabajo de la secuencia
        int pos_destino = Array.IndexOf(nueva, job_before) + 1; //Obtener la posición en la que hay que insertarlo: tras el trabajo
        job_before
        Array.Copy(nueva, pos_destino, nueva, pos_destino + 1, size - 1 - pos_destino); //Insertar trabajo en la nueva secuencia
        nueva[pos_destino] = actual[pos_origen];
}
```

Repetir_Movimiento_Prometedor_2

```
public static int Repetir_Movimiento_Prometedor_2(int[] nueva, int[] actual, int size, int job, int job_before)
{
        //Mismo proc. que Repetir_Movimiento_Prometedor, pero además devuelve la posición de origen
        Array.Copy(actual, nueva, size + 1);
        int pos_origen = Array.IndexOf(nueva, job); //Obtener la posición en la secuencia actual del trabajo insertado en la iteración
        anterior (job)
        Array.Copy(actual, pos_origen + 1, nueva, pos_origen, size - pos_origen); //Eliminar dicho trabajo de la secuencia
        int pos_destino = Array.IndexOf(nueva, job_before) + 1; //Obtener la posición en la que hay que insertarlo (tras el trabajo
        job_before
        Array.Copy(nueva, pos_destino, nueva, pos_destino + 1, size - pos_destino); //Insertar trabajo en la nueva secuencia
        nueva[pos_destino] = actual[pos_origen];

        return pos_origen;
}
```