Proyecto Fin de Carrera
Grado en Ingeniería de Organización Industrial

Análisis del problema Distributed Permutation Flowshop con tiempos de transporte post-procesado

Autor: David Bellido Rodríguez
Tutor: Paz Pérez González

Dpto. de Organización Industrial y Gestión de Empresas I
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla
Sevilla, 2024

Proyecto Fin de Carrera
Ingeniería de Organización Industrial

Análisis del problema Distributed Permutation Flowshop con tiempos de transporte post-procesado

Autor:

David Bellido Rodríguez

Tutor:

Paz Pérez González

Dpto. de Organización Industrial y Gestión de Empresas I

Escuela Técnica Superior de Ingeniería

Universidad de Sevilla

Sevilla, 2024

Autor: David Bellido Rodríguez

Tutor: Paz Pérez González

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

Presidente:

Vocales:

Secretario:

Acuerdan otorgarle la calificación de:

Sevilla, 2024

El Secretario del Tribunal

A mi familia

A mis maestros

Resumen

El presente Trabajo de Fin de Grado tiene como objetivo principal identificar el mejor método para generar soluciones a un problema Distributed Flowshop con permutación y tiempos de transporte post-procesado. Este entorno está compuesto por varias fábricas idénticas, en las que todos los trabajos que se procesan deben pasar por todas las máquinas siguiendo la misma ruta. Este problema ha ganado relevancia en la actualidad debido a la evolución de las necesidades industriales y los avances tecnológicos que han provocado una globalización y descentralización de la producción donde las empresas han distribuido sus operaciones en diferentes ubicaciones geográficas para aprovechar ventajas como menores costes laborales y proximidad con proveedores, entre muchas otras.

La metodología implementada en este estudio incluye la aplicación de 22 heurísticas basadas en la heurística NEH, adaptadas específicamente para abordar este problema. El rendimiento de estos algoritmos se evaluó mediante el ARPD (Average Relative Percentage Deviation), que mide la desviación respecto a la mejor solución obtenida, y el ACT (Average CPU Time), que mide los tiempos de cómputo. Para llevar a cabo estas evaluaciones, se han desarrollado e implementado los algoritmos en lenguaje de programación Python.

Índice

Resumen **ix**
Índice **xi**
Índice de Tablas **xiii**
Índice de Figuras **xiv**
1 Introducción **1**
2 Fundamentos teóricos **4**
3 Descripción del problema **10**
3.1. Distributed Permutation Flowshop Scheduling Problem (DPFSP) *10*
3.2. Restricciones *10*
3.3. Función objetivo *11*
4 Metodología **14**
4.1. Heurísticas NEH *14*
4.2. Representación de las soluciones *14*
4.3. Reglas de asignación *14*
4.4. NEH con formato R1Ai *16*
4.5. NEH con formato R2Ai *17*
4.6. Métodos de aceleración *17*
5 Análisis de resultados **20**
5.1. Instancias e índices de rendimiento *20*
5.2. Análisis instancias pequeñas *20*
5.3. Análisis instancias grandes *21*
6 Conclusiones **29**

Bibliografía 31

Anexo A: Código de programación 32

ÍNDICE DE TABLAS

Tabla 1. ARPD y ACT para instancias pequeñas 21

Tabla 2. ARPD de las NEH(R1,Ai) agrupada por *n* y *m* 22

Tabla 3. ARPD de las NEH(R2,Ai) agrupada por *n* y *m* 22

Tabla 4. ARPD de las NEH(R1,Ai) agrupada por F 22

Tabla 5. ARPD de las NEH(R2,Ai) agrupada por F 23

Tabla 6. ACT de las NEH(R1,Ai) agrupada por n y m 23

Tabla 7. ACT de las NEH(R2,Ai) agrupada por n y m 23

Tabla 8. ACT de las NEH(R1,Ai) agrupada por F 23

Tabla 9. ACT de las NEH(R2,Ai) agrupada por F 24

Tabla 10. ARPD y ACT para instancias grandes 25

Tabla 11. ARPD de las NEH(R2,Ai) en las últimas 50 instancias agrupada por n y m 26

Tabla 12. ARPD de las NEH(R2,Ai) en las últimas 50 instancias agrupada por F 26

Tabla 13. ACT de las NEH(R2,Ai) en las últimas 50 instancias agrupada por n y m 26

Tabla 14. ACT de las NEH(R2,Ai) en las últimas 50 instancias agrupada por F 26

ÍNDICE DE FIGURAS

Ilustración 1. Clasificación de los modelos de programación de la producción (Framinan et al., 2014). 5
Ilustración 2. Categorías de los objetivos (Framinan et al., 2014). 6
Ilustración 3. Representación R1 con regla de asignación A1 15
Ilustración 4. Representación R1 con regla de asignación A2 15
Ilustración 5. Representación R2 con regla de asignación Ai 16
Ilustración 6. NEH(R1,Ai) adaptada de (Fernandez-Viagas et al., 2018) 16
Ilustración 7. NEH(R2,Ai) adaptada de (Fernandez-Viagas et al., 2018) 17
Ilustración 8. Diagrama de caja y bigote para instancias pequeñas (RPD) 21
Ilustración 9. Diagrama de caja y bigote para instancias grandes (RPD) 24
Ilustración 10. ARPD frente ACT para instancias grandes 25

1 INTRODUCCIÓN

El presente proyecto se enfoca en determinar el método que aporte la mejor solución a un problema de Programación de la Producción en un entorno de Distributed Flowshop con restricción de permutación, tiempos de transporte post-procesado y con objetivo de minimizar el Total Delivery Time (TDT).

En un entorno de Distributed Flowshop, se cuenta con un conjunto de fábricas idénticas, donde las máquinas y, por consiguiente, los tiempos de proceso de los trabajos en cada una de ellas son iguales también. El layout de las fábricas está basado en entornos Flowshop, es decir, talleres en los que cada trabajo debe pasar por una serie de estaciones de trabajo en un orden fijo y predefinido. Además, para la programación de la producción de este problema, es necesario tomar dos decisiones: la asignación de trabajos a las fábricas y la secuenciación de los trabajos dentro de cada una de las fábricas.

Este estudio se centra en un problema particular, ya que considera los tiempos de transporte necesarios después de que cada trabajo se ha fabricado en una fábrica determinada. Así, se introduce el concepto de *Delivery Time*, que es el tiempo total en el que un trabajo se entrega al cliente, sumando su tiempo de finalización y el tiempo de transporte desde la fábrica hasta el cliente.

El objetivo del estudio es el Total Delivery Time (TDT). Minimizar la suma de los tiempos de entrega de todos los trabajos es beneficioso para este tipo de problemas, ya que optimiza la eficiencia general y minimiza los tiempos de entrega acumulativos. Esto no solo mejora el cómputo global de los tiempos de entrega, sino que también organiza los trabajos de tal manera que la entrega individual de cada uno sea más rápida.

Los trabajos se transportan de manera individual una vez completados, asegurando que cada producto llegue a su destino final en el menor tiempo posible. Esta metodología mejora la eficiencia del proceso de manufactura y logística, y, además, responde a las necesidades de un mercado cada vez más demandante y personalizado.

Este trabajo se alinea con la filosofía de Toyota y su sistema de producción Just-in-time (JIT), analizado y comprendido a través del libro de (Ohno, 1988). Este sistema ha sido estudiado e introducido en muchas organizaciones, independientemente del tipo de industria, escala o fronteras nacionales. El Just-in-time significa que, en un proceso de flujo, las piezas necesarias para el ensamblaje llegan a la línea de ensamblaje en el momento exacto que se necesitan y solo en la cantidad necesaria. Una empresa capaz de establecer este sistema se acercará a un inventario de almacenamiento nulo. Aunque este sistema originó en la industria automotriz, es aplicable a cualquier otro tipo de organización. Como decía Taiichi Ohno, *"Lo único que hacemos es mirar la línea de tiempo desde el momento en el que un cliente nos hace un pedido hasta el momento en el que recibimos el dinero. Y estamos reduciendo esa línea de tiempo eliminando los desperdicios que no agregan valor"*. Este enfoque resalta de manera sencilla pero brillante la importancia de la mejora continua, aplicable a cualquier industria que implemente este método. Entre las diferentes empresas que implementan en la actualidad ese método se encuentra Apple, McDonald's, Dell, entre muchas otras.

El documento está estructurado de la siguiente forma:

- Capítulo 2, *Fundamentos teóricos.* Se introduce el concepto de Programación de la Producción, se presenta la notación empleada, se exploran los diferentes entornos existentes, así como las restricciones y funciones objetivo más comunes.
- Capítulo 3, *Descripción del problema.* Se describe de forma detallada el problema estudiado en este proyecto. Además, se explican las nuevas incorporaciones que se hacen al problema, tanto en términos de restricciones como de función objetivo, destacando su relevancia para el estudio.
- Capítulo 4, *Metodología.* Se presentan un total de 22 heurísticas basadas en artículos relevantes que abordan el problema global estudiado. Además, se describen una serie de métodos de aceleración cuyo objetivo es reducir los tiempos de cómputo de las heurísticas.
- Capítulo 5, *Análisis de resultados.* Se lleva a cabo la aplicación de las 22 heurísticas descritas en el capítulo anterior y se analizan exhaustivamente los resultados en base a dos índices de rendimiento, el ARPD (Average Relative Percentage Deviations), que mide la distancia respecto al mejor resultado

obtenido, y el ACT (Average CPU Time) que mide el tiempo promedio de cómputo de cada una de las heurísticas.

- Capítulo 6, *Conclusiones.* Finalmente, se presentan las conclusiones derivadas del estudio realizado, basadas en los resultados obtenidos en el capítulo anterior. Se discute el impacto de los hallazgos en el campo de la Programación de la Producción y se señalan posibles direcciones futuras para la investigación.

2 FUNDAMENTOS TEÓRICOS

Para la correcta comprensión del estudio llevado a cabo en este documento, es primordial entender los fundamentos teóricos en los que está basado. El libro de (Framinan et al., 2014) proporciona la mayor parte de la información utilizada en este capítulo. Comenzando con la Programación de la Producción, esta se basa en asignar los diversos recursos de una empresa a la fabricación de una gama de productos solicitados por los clientes.

Por otro lado, se encuentra la gestión de la producción, que es un proceso de gestión en el que se toman una gran cantidad de decisiones a lo largo del tiempo para garantizar la entrega de productos con la máxima calidad, el mínimo coste y tiempo de entrega. Estas decisiones van desde decisiones estratégicas, de alto impacto y a largo plazo, como decidir si una determinada planta fabricará o no un nuevo producto, hasta decisiones operativas, de corto plazo, de bajo nivel y de pequeño impacto, como qué producto será el próximo en fabricarse en cierta máquina del taller.

Dada la diferente naturaleza y el momento de estas decisiones, y los diferentes responsables de la toma de decisiones involucrados, la gestión de la producción ha adoptado una estructura jerárquica en la que se resuelven sucesivamente los diferentes problemas de decisión. En la parte superior de esta estructura se encuentran las decisiones estratégicas, seguidas por las decisiones tácticas (a medio plazo) y finalmente por las decisiones operativas.

Sin embargo, estas decisiones no son usualmente sencillas, y requieren de datos históricos o casuales para estimar cómo usar los recursos de manera óptima. Por lo tanto, se crea previamente un plan agregado de producción, en el que se estiman las familias o grupos de productos con uso similar de recursos y/o comportamiento respecto a la demanda que se pretende vender durante un periodo de tiempo determinado. Este plan agregado de producción no da ninguna indicación detallada de las ordenes de producción que deben enviarse al taller, ahí es donde entra la Programación de la Producción.

En la programación de la producción se toman una serie de decisiones, generalmente a corto plazo, las cuales tratan de hacer coincidir los trabajos (tareas) que se van a ejecutar con los recursos disponibles de la empresa. A continuación, definimos la notación de algunos datos básicos presentes en la programación de la producción:

- $N=\{1,..,n\}$ es el conjunto de trabajos o tareas a procesar. Se utilizan los índices $j, k \in N$ para referenciar a los trabajos.
- $M=\{1,..,m\}$ es el conjunto de máquinas que procesan los trabajos o tareas. Se utiliza el índice $i \in M$ para referenciar a las máquinas.
- R_j es el vector en el que se indica el orden (Ruta) en el que el trabajo j va a ser procesado.
- p_{ij} es el tiempo en el que la máquina $i \in M$ está ocupada procesando el trabajo $j \in N$. Si es independiente de la máquina se denota como p_j.

Además de esta notación básica, es fundamental contar con un sistema de clasificación estandarizado que permita describir y analizar de manera precisa los distintos problemas que pueden surgir. Un marco ampliamente reconocido y utilizado para este propósito es la clasificación de tres campos $\alpha|\beta|\gamma$, desarrollada por (Graham et al., 1979). Este sistema proporciona una forma estructurada y concisa de representar la complejidad de los problemas de programación de tareas, facilitando la comunicación entre investigadores y la aplicación de soluciones específicas. Lo podemos ver en la *Ilustración 1.* Clasificación de los modelos de programación de la producción.

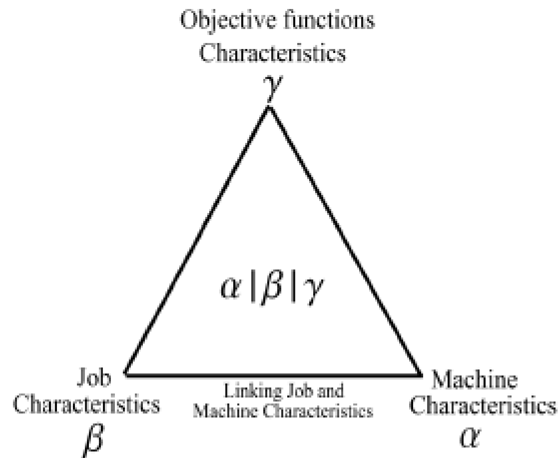

Ilustración 1. Clasificación de los modelos de programación de la producción (Framinan et al., 2014).

La notación viene dada por tres componentes esenciales:

- α: se utiliza para describir la configuración de las máquinas en el entorno de producción. Dos de las configuraciones más comunes son el entorno de máquina única y el entorno de máquinas paralelas.

  El caso más sencillo es el de una única máquina, se denota como $\alpha = 1$. En esta, cada trabajo debe pasar por la máquina realizando una única operación. Cuando tenemos *m* máquinas funcionando en paralelo se vuelve un entorno más complejo que el anterior, debido a la necesidad de asignar trabajos a múltiples máquinas y secuenciar dichos trabajos en cada una de ellas. Diferenciamos tres tipos:

  - Idénticas: se denotan como $\alpha = Pm$. Son máquinas paralelas idénticas entre sí, lo que quiere decir que sus tiempos de proceso p_j no dependen de la máquina.
  - Uniformes: se denotan como $\alpha = Qm$. Son máquinas paralelas que funcionan a diferentes velocidades v_i de tal manera que el tiempo de proceso de cada trabajo en cada máquina viene dado por $p_{ij} = \frac{p_j}{v_i}$.
  - No relacionadas: se denotan como $\alpha = Rm$. Son máquinas paralelas todas diferentes entre sí, este es el caso más general de máquinas paralelas, cuyo tiempo de proceso de cada trabajo depende de la máquina a la que se haya asignado p_{ij}.

  Por otro lado, tenemos el caso en el que cada máquina tiene un propósito diferente y en el que cada trabajo visita todas las máquinas ya que debe realizar una operación en cada una. Este es el entorno de los talleres. Dependiendo de la ruta de los trabajos R_j diferenciamos diferentes talleres:

  - Taller de flujo regular (Flowshop): se denota como $\alpha = Fm$. En este caso, todos los trabajos tienen la misma ruta $R_j = (1,2,..,m), \forall j \in N$.
  - Taller de trabajo (Job shop): se denota como $\alpha = Jm$. En este caso, cada trabajo tiene una ruta diferente, por tanto $R_j,\ \forall j \in N$ son un dato necesario.
  - Taller abierto (Open shop): se denota como $\alpha = Om$. Este es el caso más general y complejo de los talleres, en él no hay ruta determinada para ninguno de los trabajos.
  - Taller de flujo distribuido (Distributed Flowshop): se denota como $\alpha = DF$. En este caso, se dispone de un conjunto de fábricas idénticas donde cada una de ellas opera como un taller de flujo (Flowshop). Es decir, todos los trabajos tienen la misma ruta.

- β: incluye las características y restricciones específicas de los trabajos. Esto abarca tiempos de procesamiento, tiempos de llegada, fechas de vencimiento, restricciones de precedencia, y otros muchos aspectos que influencian el orden y la forma en que los trabajos deben ser programados.

  Adicionalmente, cuando el campo está vacío $\beta = \emptyset$, se asumen unas suposiciones generales: los trabajos están disponibles al principio del horizonte de programación, los trabajos no se pueden interrumpir, las máquinas están siempre disponibles, cada máquina puede hacer un trabajo a la vez y un trabajo puede ser realizado solo en una máquina, el buffer entre máquinas se supone infinito y el

tiempo de transporte es despreciable (Perez-Gonzalez et al., 2023)

Dentro de este campo encontramos muchas restricciones diferentes, las más comunes son las siguientes:

- $\beta = r_j$: *release dates*. Son las fechas de llegada (disponibilidad) de los trabajos o tareas a la planta.
- $\beta = \bar{d}_j$: *deadlines*. Son fechas de entrega de los trabajos de obligado cumplimiento. Cuando no son de obligado cumplimiento no aparecen en el campo β y se denota como d_j *(common due date)*.
- $\beta = s_{ij}$: *setup times*. Son tiempos de preparación de la máquina *i* antes de procesar el trabajo *j*. Cuando dependen de la secuencia se denota como $\beta = s_{ijk}$ y cuando es independiente de la máquina se elimina el subíndice *i*.

En los entornos taller encontramos otras restricciones como pueden ser:

- $\beta = prmu$. Es una restricción muy común en los talleres de flujo o flowshop, esta implica que la secuencia de trabajos es la misma para todas las máquinas.
- $\beta = no - idle$. Esta restricción no permite tiempos ociosos de las máquinas entre trabajos. Una vez que la máquina empieza a procesar el primer trabajo de su programa, no puede parar.
- $\beta = no - wait$. Esta restricción no permite que los trabajos puedan esperar entre máquinas o etapas.
- $\beta = tap$. Esta restricción, aplicada en entornos de talleres distribuidos, implica que se deben considerar los tiempos de transporte post-procesado desde cada una de las fábricas para cada uno de los trabajos. Esta se explica más en detalle en el *Capítulo 3*.

- γ: Indica el criterio de optimización, es decir, la métrica que se busca minimizar o maximizar. En programación de la producción, las funciones objetivo se clasifican en las categorías de coste, tiempo, calidad y flexibilidad, como se puede ver en la *Ilustración 2.*Ilustración 2. Categorías de los objetivos

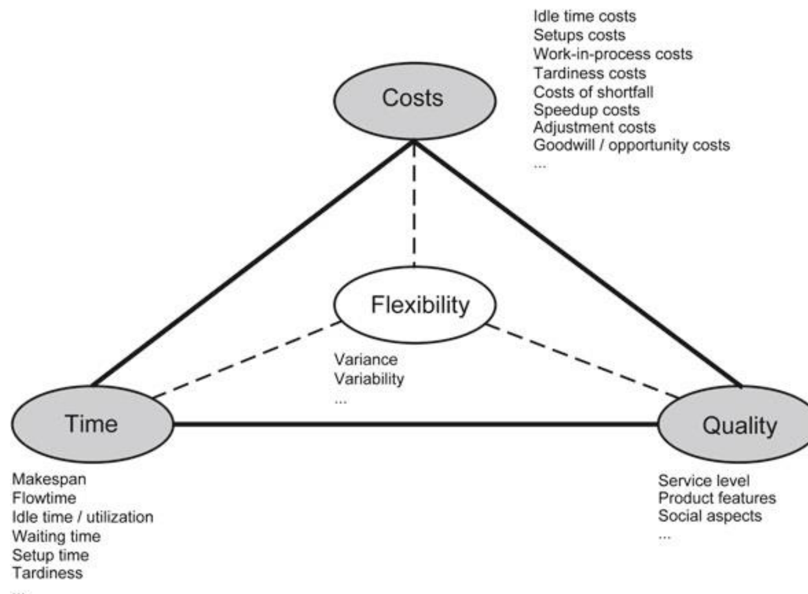

Ilustración 2. Categorías de los objetivos (Framinan et al., 2014).

Dentro de este campo existen algunas medidas de rendimiento, entre las cuales encontramos:

- C_j: Tiempo de finalización del trabajo *j (completion time)*.
- F_j: Tiempo de flujo del trabajo *j (flowtime)*. Es el tiempo que el trabajo está dentro del entorno. Viene dado por $F_j = C_j - r_j$

- L_j: Retraso del trabajo *j (lateness)*. Mide lo tarde que se termina un trabajo con respecto a su fecha de entrega, tal que $L_j = C_j - d_j$
- T_j: Tardanza del trabajo *j (tardiness)*. Si el trabajo se termina antes de su fecha de entrega, no pasa nada. $T_j = \max\{0, L_j\}$
- E_j: Adelanto del trabajo *j (earliness)*. Si el trabajo termina después de su fecha de entrega, no pasa nada. $E_j = \max\{0, -L_j\}$
- U_j: Trabajo tarde (*tardy job*). $U_j = \begin{cases} 1 & si\ T_j > 0 (C_j > d_j) \\ 0 & en\ caso\ contrario \end{cases}$
- DT_j: Delivery Time (Tiempo de entrega). Está directamente relacionado con la restricción *tap,* ya que representa el tiempo en el que el trabajo llega a su destino final, incluyendo el tiempo de transporte al tiempo de finalización. Este término y sus funciones objetivo asociadas serán exploradas en detalle en el *Descripción del* problema.

Mediante estas medidas de rendimiento, obtenemos las posibles funciones objetivo usando dos formatos: el formato de maximización y el formato de sumatoria.

- No relacionados con fechas de entrega:
  - Max-form: $f = \max_{1 \le j \le n} g(C_j)$
    - Makespan (Maximum Completion Time*):* $C_{max} = \max_{1 \le j \le n} C_j$
    - Maximum Flowtime: $\max F_j = \max_{1 \le j \le n} F_j$
    - Maximum Delivery Time: $\max DT_j = \max_{1 \le j \le n} DT_j$
  - Sum-form: $f = \sum_{j=1}^{n} g(C_j)$
    - Total Completion Time: $\sum C_j = \sum_{j=1}^{n} C_j$
    - Total Flow time: $\sum F_j = \sum_{j=1}^{n} F_j$
    - Total Delivery Time: $\sum DT_j = \sum_{j=1}^{n} DT_j$
- Relacionados con fechas de entrega:
  - Max-form:
    - Maximum Lateness: $\max L_j = \max_{1 \le j \le n} L_j$
    - Maximum Tardiness: $\max T_j = \max_{1 \le j \le n} T_j$
    - Maximum Earliness: $\max E_j = \max_{1 \le j \le n} E_j$
  - Sum-form:
    - Total Lateness: $\sum L_j = \sum_{j=1}^{n} L_j$
    - Total Tardiness: $\sum T_j = \sum_{j=1}^{n} T_j$
    - Total Earliness: $\sum E_j = \sum_{j=1}^{n} E_j$
    - Number of tardy jobs: $\sum U_j = \sum_{j=1}^{n} U_j$

El resultado final de este proceso es, idealmente, un programa o s*chedule* el cual indica las fechas de comienzo y fin de cada trabajo en cada una de las máquinas. Es muy importante no confundir con el termino secuencia, esta indica el orden en el que cada trabajo comienza a procesarse en cada máquina. En general, a cada secuencia le corresponden infinitos programas, depende de la regla de asignación que se esté utilizando y de la codificación de la secuencia.

Un programa que cumple con todas las restricciones y características del proceso productivo se conoce como programa admisible o *feasible schedule*. Además, cuando no es posible adelantar ninguna operación sin cambiar el orden en que alguna máquina procesa los trabajos y localmente (en cada máquina) cada trabajo está lo más a la izquierda posible para un orden dado, se conoce como un programa semiactivo o *semi-active schedule* (Perez-Gonzalez et al., 2023).

3 DESCRIPCIÓN DEL PROBLEMA

En este capítulo se explica de manera precisa y detallada el problema estudiado en este proyecto. En este caso, el problema es en un entorno Distributed Flowshop, que es un caso particular del Flowshop, explicado anteriormente en *Fundamentos teóricos*.

El caso estudiado es $DF|prmu, tap|\sum DT_j$ según la clasificación (Graham et al., 1979).

3.1. Distributed Permutation Flowshop Scheduling Problem (DPFSP)

El Distributed Permutation Flowshop Sheduling Problem, en adelante como DPFSP, se puede definir más formalmente de la siguiente manera: un conjunto N de n trabajos que deben ser procesados en un conjunto F de f fábricas, donde cada una dispone del mismo conjunto M de m máquinas, de igual forma que el comúnmente reconocido Permutation Flowshop Scheduling Problem (PFSP) (Naderi & Ruiz, 2010).

3.2. Restricciones

Es importante definir una serie de suposiciones generales, descritas con anterioridad en los *Fundamentos teóricos*, que también podemos ver en (Naderi & Ruiz, 2010). Todos los trabajos son independientes y están disponibles para procesarse desde el instante 0. Las máquinas están siempre disponibles. Cada máquina solo puede procesar un trabajo a la vez. Cada trabajo solo puede ser procesado en una máquina a la vez. Los trabajos no pueden ser interrumpidos una vez han comenzado a procesarse en una máquina. Se asume que no hay tiempos de setup. Se asumen buffers infinitos.

Además de estas suposiciones, en cuanto a tiempos de transporte, se asume que no hay tiempos de transporte entre máquinas, y tampoco hay posibilidad de cambiar un trabajo entre fábricas, es decir si un trabajo comienza su primera etapa en la fábrica *f,* debe realizar el resto de la ruta, R_j, en esa misma fábrica *f.*

Adicionalmente, se añade la restricción de permutación, explicada previamente en *Fundamentos teóricos*, la cual implica que para cada fábrica *f* del conjunto *F*, la secuencia de cada máquina *m* del conjunto *M,* es la misma.

Por otro lado, hay restricciones menos comunes que incluyen los tiempos de transporte. Este tema es muy importante en la gestión de la cadena de suministro, ya que el proceso de producción se ve muy afectado por la disponibilidad de materiales o por la capacidad de los buffers. Por lo tanto, en la literatura se consideran tres tipos de transporte (Perez-Gonzalez & Framinan, 2024):

- tbp, transporte antes del procesamiento. Se refiere al transporte de materias primas desde su origen hasta las fábricas. Esto se traduce en fechas de lanzamiento de los trabajos a programar, o en un tiempo de transporte dependiente de la fábrica que puede verse como fechas de lanzamiento dependiendo del trabajo y la fábrica.
- tdp, transporte durante el procesamiento. Se refiere al transporte de productos semiacabados entre fábricas, por lo que un trabajo que ha comenzado su procesamiento en una fábrica puede completarse en otra fábrica.
- tap, transporte después del procesamiento. Se refiere al transporte del producto terminado a su destino final. En este caso los trabajos se suelen agrupar en lotes para ser entregados de manera conjunta y por tanto se tiene un tiempo de transporte acorde al lote específico (Q. Li et al., 2021). O, por otro lado, se considera un problema de rutas de vehículos en los que se proporcionan los tiempos de transporte entre clientes (Hou et al., 2022).

En el problema estudiado en este proyecto, se añaden los tiempos de transporte después del procesamiento (tap). En este caso, la idea es que el tiempo de transporte depende exclusivamente del trabajo y de la fábrica en

la que se realice, puesto que se suponen lotes de una unidad. Es decir, una vez conocido el tiempo de finalización C_j del trabajo *j*, se hace llegar dicho trabajo al cliente desde la fábrica *f* donde se ha producido, con un tiempo de transporte TAP_{fj}.

3.3. Función objetivo

El estudio del DPFSP ha ganado atención en la literatura debido a su relevancia práctica y la complejidad añadida por la distribución de las instalaciones. Por tanto, a lo largo de los años se han ido analizando diferentes funciones objetivo para este problema.

(Naderi & Ruiz, 2010) fueron pioneros en el estudio del DPFSP, centrándose en la minimización del makespan. Esta función objetivo representa el tiempo total requerido para completar todas las tareas, lo cual es fundamental para mejorar la eficiencia en sistemas de producción distribuidos.

Sin embargo, en los últimos años, el problema de programación con el criterio de Total Flow Time (o el equivalente Total Completion Time cuando los tiempos de entrega r_j, son 0) ha comenzado a atraer más atención entre los investigadores. Esto se debe a que el entorno de producción actual es mucho más dinámico y está directamente relacionado con importantes medidas de rendimiento logístico de fabricación, como el nivel de inventario en proceso (Liu & Reeves, 2001).

Por esta misma razón, algunos investigadores han analizado el problema del Distributed Flowshop para minimizar el Total Flow Time (Fernandez-Viagas et al., 2018) y el Total Completion Time (Q. Y. Li et al., 2024). Este objetivo minimiza el tiempo de entrega de los trabajos de manera individual, estabiliza el uso de recursos y reduce el inventario en proceso. Estas ventajas se acentúan aún más en un entorno distribuido donde los desequilibrios entre los recursos y el trabajo en proceso pueden aumentar debido a la existencia de varias fábricas.
En este proyecto, se aborda un objetivo parecido al este, pero con tiempos de entrega o *Delivery times*.

Los *Delivery times,* introducidos en el capítulo anterior, se definen en

Ecuación 1. Estos determinan el instante de tiempo en el cual al cliente le llega su pedido, es decir, tiene en cuenta el tiempo de finalización del trabajo *j* y su tiempo de transporte desde la fábrica *f* donde se haya procesado, hasta el cliente, tal que:

$$DT_j = C_j + TAP_{fj}, \forall j \in N$$

Ecuación 1. Delivery Time

Aplicando el mismo criterio que en el Total Completion Time o en el Total Flow Time, se define el Total Delivery Time (TDT) en *Ecuación 2*:

$$Total\ Delivery\ Time\ (TDT) = \sum_{j=1}^{n} DT_j$$

Ecuación 2. Total Delivery Time

Por otro lado, tenemos la opción del Máximo Delivery Time, definido *en Ecuación 3*. Este es el equivalente al makespan, pero con el nuevo termino de los tiempos de entrega.

$$\max DT_j = \max_{1 \le j \le n} DT_j$$

Ecuación 3. Máximo Delivery Time

Ambas métricas, Máximo Delivery Time (DTmax) y Total Delivery time (TDT), son buenas opciones para este problema. Sin embargo, el TDT tiene un enfoque más individualizado en cada tarea, donde se trata de optimizar los recursos y reducir el tiempo de espera para todas y cada una de ellas. Por tanto, teniendo en cuenta el problema que se estudia en este proyecto, la función objetivo utilizada es el TDT.

El PFSP minimizando el Total Completion Time, $Fm|prmu|\sum C_j$, es NP-hard para $m \geq 2$ (Garey, Johnson, & Sethi, 1976). Dado que este problema es un caso particular del que se estudia en este proyecto, cuando el número de fábricas es 1 y los tiempos de transporte son 0, entonces $DF|prmu, tap|\sum DT_j$ también es NP-hard. Por consiguiente, este proyecto se centra en obtener soluciones aproximadas al problema planteado de manera que los tiempos de cómputo sean razonables.

4 METODOLOGÍA

En este capítulo se aborda la representación de soluciones al problema planteado basándonos en la heurística NEH. A partir de esta heurística base, se derivan un total de 22 variantes, cada una adaptada según la representación de la solución y la regla de asignación específicas. El objetivo común de estas 22 heurísticas es obtener las mejores soluciones para el problema estudiado, considerando cuidadosamente sus características y los objetivos de optimización buscados.

4.1. Heurísticas NEH

La heurística constructiva de Nawaz/Enscore/Ham (NEH) es una de las mejores heurísticas para minimizar el tiempo total (makespan) en problemas de flowshop con permutación. El enfoque NEH consta de dos pasos: (1) la generación de un orden inicial de trabajos con respecto a un valor indicado y (2) la inserción iterativa de trabajos en una secuencia parcial de acuerdo con el orden inicial del paso 1 (Nawaz et al., 1983). En este estudio se adapta esta heurística al problema del Distributed Flowshop y al objetivo del Total Delivery Time, de tal forma que se minimice lo máximo posible.

4.2. Representación de las soluciones

Se emplean dos representaciones de las soluciones, de igual forma que se hizo en (Fernandez-Viagas et al., 2018) estudiando el problema para la minimización de Total Flow Time.

Por un lado, está la representación R1, que consiste en una única secuencia de permutación de los trabajos. Esta representación por sí sola no proporciona información sobre la qué fábrica en la que está asignado cada trabajo. Para determinar este dato, necesita de una regla de asignación.

Por otro lado, está la representación R2, esta es una secuencia en forma de matriz, la cual contiene las secuencias completas por cada fábrica, es decir, al contrario que R1, esta representación sí que muestra por sí misma en qué fábrica está asignado cada trabajo.

Dependiendo de la representación que se esté utilizando, la manera de obtener las soluciones será de una manera completamente distinta.

4.3. Reglas de asignación

Hay un total de 11 reglas de asignación, 6 de ellas son las que podemos encontrar en (Fernandez-Viagas et al., 2018) pero adaptadas al problema planteado en este estudio. Las reglas son las siguientes:

- A1: Asignar el trabajo en la fábrica con menor makespan global (sin incluir el tiempo de transporte) después de asignar el trabajo.
- A2: Asignar el trabajo en la fábrica donde el makespan local (sin incluir el tiempo de transporte) sea menor, después de poner el trabajo (Earliest Completion Time).
- A3: Asignar el trabajo en la fábrica donde el makespan local (sin incluir el tiempo de transporte) sea menor, antes de poner el trabajo (First Available Machine).
- A4: Asignar el trabajo a la fábrica donde el Delivery Time sea menor después de poner el trabajo.
- A5: Asignar el trabajo a la fábrica donde el Total Delivery Time global (TDT) sea menor después de colocar el trabajo.

- A6: Asignar el trabajo a la fábrica con el menor DTmax local antes de poner el trabajo.
- A7: Asignar el trabajo a la fábrica con el menor Total Delivery Time (TDT) local antes de poner el trabajo.
- A8: Asignar el trabajo a la fábrica con el menor Total Delivery Time (TDT) local después de colocar el trabajo en la mejor posición.
- A9: Asignar el trabajo en la fábrica con el menor Total Delivery Time (TDT) global después de insertar el trabajo en la mejor posición.
- A10: Igual que A9, pero se excluye la fábrica con peor Total Delivery Time (TDT) a priori.
- A11: Igual que A10, pero se prueba en las F/2 fábricas con menor Total Delivery Time (TDT) a priori.

Las últimas dos reglas A10 y A11 presentan una notable ventaja en la reducción del tiempo de cómputo. No obstante, estas reglas también presentan un inconveniente significativo al excluir ciertas fábricas para disminuir el tiempo de cómputo. Al inicio de la aplicación de la regla, cuando aún no se ha asignado ningún trabajo a ninguna fábrica, se tiende a descartar sistemáticamente las primeras fábricas.

Este enfoque puede ser perjudicial, ya que, dado que los tiempos de transporte dependen de la fábrica seleccionada, se corre el riesgo de excluir desde el principio aquellas fábricas que podrían ofrecer tiempos de transporte más reducidos, únicamente con el fin de optimizar el tiempo de cómputo.

Para mitigar este problema, se desarrollaron las reglas A12 y A13, las cuales son equivalentes a las reglas A10 y A11, respectivamente. La modificación clave es que el descarte de fábricas comienza una vez alcanzado una cantidad mínima de fábricas con al menos un trabajo asignado. Por ejemplo, si se requiere descartar la mitad de cuatro fábricas (es decir, dos fábricas), no se descarta ninguna hasta que haya al menos dos fábricas con algún trabajo asignado.

Aunque esta estrategia podría incrementar ligeramente los tiempos de cómputo, también podría mejorar la calidad de las soluciones obtenidas. Sin embargo, tras realizar diversas pruebas, se observó que las reglas A12 y A13 ofrecían mejoras muy pequeñas y en casos muy puntuales. En la mayoría de los casos, los beneficios obtenidos no compensaban el aumento en el tiempo de cómputo requerido. Por esta razón ambas reglas fueron descartadas.

A continuación, vemos un ejemplo claro de aplicación de las diferentes representaciones de soluciones:

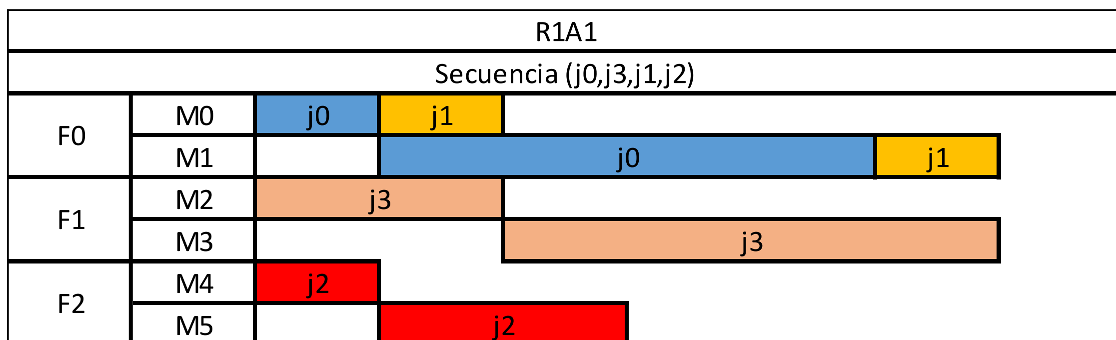

Ilustración 3. Representación R1 con regla de asignación A1

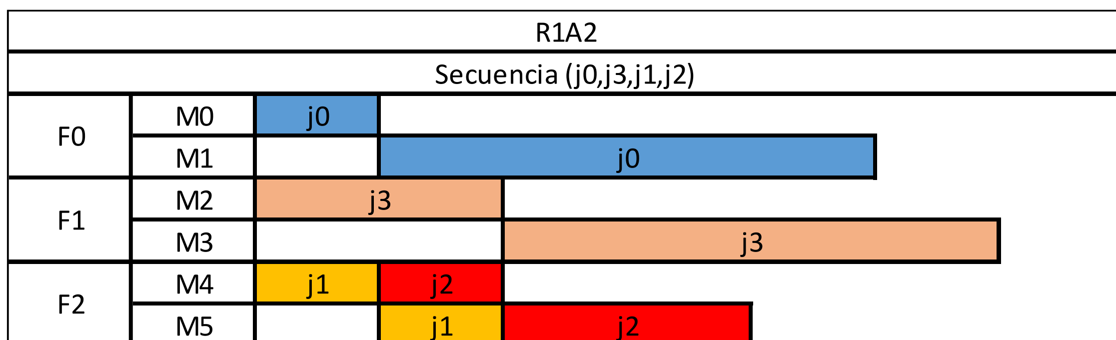

Ilustración 4. Representación R1 con regla de asignación A2

R2Ai				
Secuencia ((j0,j1),(j3),(j2))				
F0	M0	j0	j1	
	M1		j0	j1
F1	M2	j3		
	M3		j3	
F2	M4	j2		
	M5		j2	

Ilustración 5. Representación R2 con regla de asignación Ai

Como se muestra en la *Ilustración 3* e *Ilustración 4,* cuando se trata con R1, dependiendo de la regla de asignación se obtendrá un Diagrama de Gantt o Schedule diferente. Sin embargo, para R2 independientemente de la regla de asignación, dada una secuencia con este formato, la asignación la da la propia secuencia. Es importante no confundir esto con la obtención de dicha secuencia, para la cual sí será necesario aplicar las reglas de asignación, como se explicará en la sección posterior.

4.4. NEH con formato R1Ai

En el caso de las heurísticas NEH con la codificación de R1, las reglas de asignación influyen en la obtención de la función objetivo. Ya que, como se vio anteriormente, dependiendo de la regla de asignación empleada, se obtiene un *Schedule* diferente. En la *Ilustración 6* podemos ver el pseudocódigo de esta heurística.

Algorithm 1 NEH($\mathcal{R}_1, A_i$)

`// Initial Order`

$\Omega \leftarrow$ Jobs sorted in non-increasing sums of processing times ($\Omega = \{\omega_1, \ldots, \omega_n\}$);

$\Pi \leftarrow (\omega_1)$;

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

`// Construction of partial sequences`

$\Pi^{\omega kl} \leftarrow$ Partial sequence formed by inserting job ω_k in position l of partial sequence Π, $\forall l \in \{1, \ldots, k\}$;

`// Total delivery time evaluation`

Compute each partial sequence $\Pi^{\omega kl}$ by using assignment rule A_i to allocate jobs to factories at the end of that factory unless the assignment rule A_i indicates to find the best position, in which case all possible positions within each factory will be tested. Let l^* be the index l whose total completion time is minimal, i.e. $l^* \leftarrow \arg\min_l \{D_{\text{sum}}(\Pi^{\omega kl})\}$;

`// Sequence selection`

$\Pi \leftarrow \Pi^{\omega kl^*}$;

end for

Ilustración 6. NEH(R1,Ai) adaptada de (Fernandez-Viagas et al., 2018)

Inicialmente se parte de una secuencia de trabajos ordenada de manera descendiente según la suma de sus tiempos de proceso. Posteriormente, se asigna a la secuencia parcial Π (con formato R1), el primer trabajo de esa secuencia ordenada.

A continuación, se asigna el siguiente trabajo de la secuencia ordenada a la secuencia Π. Para ello, se construyen secuencias parciales en las que se asigna el trabajo en cada una de las posibles posiciones. La regla de asignación determinará en que fábrica y en que posición va cada trabajo de esas secuencia parciales, lo que permite evaluarlas y ver cuál de todas las *l* posiciones dentro de la secuencia Π obtiene los mejores resultados.

Este proceso se repite hasta asignar todos y cada uno de los trabajos de la secuencia de partida.

4.5. NEH con formato R2Ai

Al contrario que las heurísticas NEH con la codificación R1, cuando empleamos la codificación R2, las reglas de asignación influyen en la manera de obtener la secuencia, ya que, una vez obtenida esa secuencia en forma matricial, la obtención de la función objetivo es común para todas ellas. En la *Ilustración 7* podemos ver el pseudocódigo de esta heurística.

Algorithm 2 NEH($\mathcal{R}_2, A_i$)

```
// Initial Order
Ω ← Jobs sorted in non-increasing sums of processing times (Ω = {ω_1, ..., ω_n});
Π_f ← ∅, ∀f ∈ F;
for k = 1 to n do
// Assignment rule
    (l, f) ← A_i;
// Construction of partial sequences
    Π_f ← Sequence formed by inserting job ω_k in position l of partial sequence Π_f;
end for
Π ← (Π_1, ..., Π_f)
```

Ilustración 7. NEH(R2,Ai) adaptada de (Fernandez-Viagas et al., 2018)

En este caso, también se parte de la misma secuencia ordenada de mayor a menor suma de tiempos de proceso. Sin embargo, no se asigna directamente el primer trabajo de esta a la secuencia parcial Π, ya que ahora está en formato R2. Es decir, se debe aplicar una regla de asignación para determinar en qué fábrica y en qué posición se colocará el trabajo. Así, $\Pi = (\Pi_1, \dots, \Pi_f)$, donde Π_f representa las secuencias parciales de cada una de las fábricas.

Asignamos el primer trabajo de la secuencia inicial. Para ello, se crean secuencias parciales Π_f donde se inserta el trabajo en todas las posibles posiciones, o únicamente en la última posición de cada fabrica. Esto lo determinará la regla de asignación que se esté aplicando, la cual finalmente decidirá cuál de todas las secuencias parciales Π_f es la mejor, y, por tanto, cuál es la mejor fábrica *f* y la mejor posición *l* para asignar ese trabajo.

Este proceso se repite hasta asignar todos y cada uno de los trabajos de la secuencia de partida.

4.6. Métodos de aceleración

Uno de los problemas principales de estas heurísticas es el tiempo de cómputo de estas, ya que requieren insertar los trabajos en cada una de las posiciones posibles en busca del mejor resultado. Esto puede implicar un tiempo de cómputo excesivo en muchas ocasiones. Para mitigar este problema, se han empleado algunos métodos de aceleración. Es importante destacar que estos métodos se han utilizado únicamente en la codificación R1, ya que la R2 es lo suficientemente rápida como para no requerir dichas modificaciones.

El primer método se aplica a R1A1 y R1A2, que necesitan calcular el makespan insertando cada trabajo en la última posición de la fábrica seleccionada. Para evitar el cálculo de todos los Completion Times, C_j, se calcula únicamente el makespan del último trabajo colocado. De esta manera, se evita el tiempo de cómputo adicional que implicaría recalcular la secuencia al completo repetidamente.

El segundo método consiste en utilizar la memoización, una técnica de optimización que se usa principalmente para acelerar tiempos de cómputo. Esta técnica, descrita por (Norvig, 1991), se basa en crear una función que permite recordar automáticamente los resultados de cálculos previos.

En este contexto, la memoización almacena en memoria los valores del objetivo calculado según la regla de asignación de una secuencia dada. Así, cuando la heurística NEH realiza los cálculos de la función objetivo con las diferentes permutaciones de la secuencia, evita recalcular lo que ya ha sido calculado anteriormente.

Finalmente, para las reglas en las que no se especifica buscar la mejor posición del trabajo en la fábrica, se asigna siempre al final de la secuencia ya existente en esa fábrica. De tal manera que si tenemos una secuencia como ((j3,j1,j5), (j2)) si la regla de asignación determina que el trabajo j6 se asigna a la última fábrica, se inserta de la siguiente forma ((j3,j1,j5), (j2,j6)).

Por el contrario, cuando se especifica insertar en la mejor posición, se prueba en cada una de las posibles posiciones dentro de cada fábrica, de tal manera que nos quedamos con la fábrica y la posición dentro de la fábrica que mejor resultados da.

Este último enfoque, aplicado tanto para R1 como para R2, nos permite reducir tiempos de cómputo en las reglas de asignación que no especifiquen buscar la mejor posición posible.

5 ANÁLISIS DE RESULTADOS

En este capítulo se examinan los resultados obtenidos al aplicar las 22 heurísticas descritas en el capítulo anterior al problema planteado en este estudio. Las pruebas pertinentes se llevaron a cabo utilizando Python como lenguaje de programación, junto con la librería *schedule* (*Librería SCHEDULE – Grupo de Investigación Organización Industrial,* n.d.). Todo el código utilizado está disponible en el *Anexo A*: Código de programación.

5.1. Instancias e índices de rendimiento

Antes de comenzar con el análisis de los resultados obtenidos en este estudio, debemos definir las instancias utilizadas. Se han distinguido dos grupos de instancias: un grupo de instancias pequeñas formado por 34 combinaciones diferentes de los parámetros $n \in \{4,6,8,10,12,14,16\}, m \in \{2,3,4,5\}, F \in \{2,3,4\}$, y un grupo de instancias más grandes basadas en las instancias de (Taillard, 1993). En todas ellas, los tiempos de proceso, p_j, y los tiempos de transporte post-procesado, TAP_{fj}, están generados utilizando una distribución uniforme en el rango [1,99].

Para el análisis de ambos tamaños se define el índice RPD (Relative Percentage Deviations) en *Ecuación 4*, siendo TDT_i^h el Total Delivery Time de la instancia *i*, con $i \in [1, I]$, para la heurística *h,* con $h \in H$, siendo *H* el conjunto de heurísticas. Por otro lado, $Best_i^H$ representa el mejor Total Delivery Time entre todas las heurísticas del conjunto *H* para esa instancia *i*.

Adicionalmente, se define el CT_i^h (CPU Time) cómo el tiempo de cómputo, medido en segundos, de la heurística *h* en la instancia *i*. Este nos permite evaluar la escalabilidad y la eficiencia computacional del algoritmo, especialmente en problemas reales donde el tiempo de ejecución pueda ser un limitador importante.

$$RPD_i^h = \frac{TDT_i^h - Best_i^H}{Best_i^H} \times 100$$

Ecuación 4. Indice RPD

5.2. Análisis instancias pequeñas

Estas instancias se consideran como pruebas preliminares, que nos proporcionan una visión general del rendimiento y comportamiento de las heurísticas. Por tanto, se ha decidido no examinar todas las posibles combinaciones de los parámetros *n, m* y *F*. Esta información es útil para prever cómo funcionarán en instancias más grandes, que son las que se encuentran con mayor frecuencia en situaciones reales.

Por ende, en vez de mostrar los RPD individuales de cada método en cada instancia, se muestran los ARPD (Average Relative Percentage Deviations), definido en *Ecuación 5*. Por otro lado, en lugar de los tiempos de cómputo individuales (CT), se utilizará el promedio de estos (ACT: Average CPU Time).

$$ARPD = \frac{1}{I} \times \sum_{i=1}^{I} \frac{TDT_i - Best_i}{Best_i}$$

Ecuación 5. Índice ARPD

Tras haber definido los parámetros de análisis, procedemos a presentar y analizar de manera detallada los resultados obtenidos, los cuales se muestran en la *Tabla 1.*

NEH(.,.)	ARPD	ACT	NEH(.,.)	ARPD	ACT
$\mathcal{R}_1, \mathcal{A}_1$	11.132	0,0049	$\mathcal{R}_2, \mathcal{A}_1$	34.271	0,0004
$\mathcal{R}_1, \mathcal{A}_2$	10.721	0,0042	$\mathcal{R}_2, \mathcal{A}_2$	32.587	0,0006
$\mathcal{R}_1, \mathcal{A}_3$	11.596	0,0066	$\mathcal{R}_2, \mathcal{A}_3$	33.841	0,0003
$\mathcal{R}_1, \mathcal{A}_4$	2.792	0,0413	$\mathcal{R}_2, \mathcal{A}_4$	27.470	0,0010
$\mathcal{R}_1, \mathcal{A}_5$	2.792	0,0222	$\mathcal{R}_2, \mathcal{A}_5$	27.470	0,0008
$\mathcal{R}_1, \mathcal{A}_6$	11.696	0,0137	$\mathcal{R}_2, \mathcal{A}_6$	34.725	0,0003
$\mathcal{R}_1, \mathcal{A}_7$	12.879	0,0094	$\mathcal{R}_2, \mathcal{A}_7$	34.900	0,0002
$\mathcal{R}_1, \mathcal{A}_8$	1.326	0,0461	$\mathcal{R}_2, \mathcal{A}_8$	6.152	0,0024
$\mathcal{R}_1, \mathcal{A}_9$	0.358	0,0452	$\mathcal{R}_2, \mathcal{A}_9$	2.468	0,0025
$\mathcal{R}_1, \mathcal{A}_{10}$	0.966	0,0386	$\mathcal{R}_1, \mathcal{A}_{10}$	6.034	0,0018
$\mathcal{R}_1, \mathcal{A}_{11}$	1.872	0,0334	$\mathcal{R}_1, \mathcal{A}_{11}$	8.883	0,0013

Tabla 1. ARPD y ACT para instancias pequeñas

Los tiempos de cómputo promedio de cada heurística son notablemente reducidos, casi insignificantes. Esta característica hace que estas instancias de prueba sean ideales para obtener esos resultados preliminares de los que se hablaban en la introducción de esta sección. Se observa que la regla de asignación A9 muestra los mejores resultados para ambas representaciones de las soluciones, R1 y R2, siendo esta última ligeramente inferior a la primera. Además, se evidencia que, para la codificación R2, con excepción de las reglas A8, A9, A10 y A11, los demás resultados tienden a ser considerablemente peores, llegando hasta un 34.9% de diferencia con respecto al mejor de los resultados. En adición a esto último, las heurísticas R1A4 y R1A5 muestran resultados comparables a las mencionadas anteriormente, en contraste con las demás.

A fin de complementar el análisis presentado en la *Tabla 1,* se incluye a continuación un diagrama de caja y bigotes que ofrece una representación visual de los índices RPD de cada heurística evaluada. Este diagrama ofrece una perspectiva más clara y visual de las disparidades en el rendimiento entre las distintas heurísticas, lo cual no solo revela las divergencias más marcadas, sino también las semejanzas en el rendimiento entre ellas.

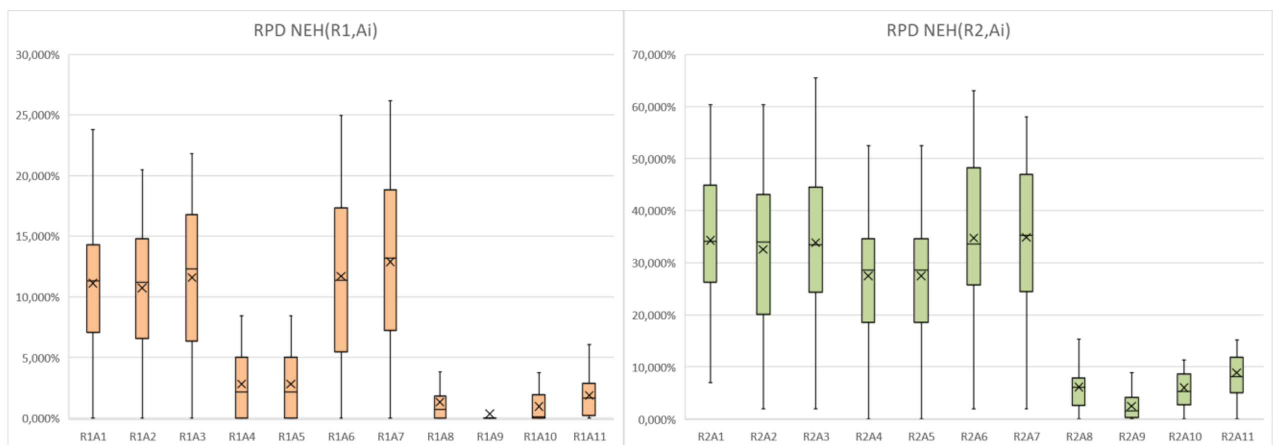

Ilustración 8. Diagrama de caja y bigote para instancias pequeñas (RPD)

5.3. Análisis instancias grandes

En esta sección estudiaremos el conjunto de instancias de mayor tamaño, lo cual nos permite explorar cómo las heurísticas propuestas responden y se adaptan a problemas con una mayor complejidad estructural y de datos. Este análisis es crucial para evaluar la viabilidad y efectividad de las heurísticas en escenarios prácticos y realistas.

En este contexto, el objetivo es analizar cómo las diferentes heurísticas gestionan la optimización en instancias grandes, comparando tanto la calidad de las soluciones como los tiempos de cómputo. Comenzaremos analizando las primeras 70 instancias, que abarcan combinaciones entre *n, m* y *F* tal que: $n \in \{20, 50, 100\}, m \in \{5, 10, 20\}$ y $F \in \{2, 3, 4, 5, 6, 7\}$.

Esto resulta en 70 instancias para cada tamaño de fábrica, sumando un total de 420 instancias. A continuación, se muestran los resultados del índice ARPD para cada uno de los métodos en función de los parámetros *n* y *m* *(ver Tabla 2* y *Tabla 3)* y en función del parámetro *F (ver Tabla 4* y *Tabla 5)*. De igual forma, se muestran los tiempos de cómputo (ACT) en la *Tabla 6, Tabla 7, Tabla 8* y *Tabla 9*.

m x n	$\mathcal{R}_1, \mathcal{A}_1$	$\mathcal{R}_1, \mathcal{A}_2$	$\mathcal{R}_1, \mathcal{A}_3$	$\mathcal{R}_1, \mathcal{A}_4$	$\mathcal{R}_1, \mathcal{A}_5$	$\mathcal{R}_1, \mathcal{A}_6$	$\mathcal{R}_1, \mathcal{A}_7$	$\mathcal{R}_1, \mathcal{A}_8$	$\mathcal{R}_1, \mathcal{A}_9$	$\mathcal{R}_1, \mathcal{A}_{10}$	$\mathcal{R}_1, \mathcal{A}_{11}$
5 x 20	14,380	16,993	19,408	4,508	4,508	19,329	22,332	1,774	0,194	0,518	1,476
5 x 50	24,755	27,651	32,229	13,548	13,548	31,044	33,321	2,372	0,145	0,640	1,728
5 x 100	31,144	32,810	36,599	24,306	24,306	36,384	37,476	2,619	0,044	0,696	1,832
10 x 20	10,409	13,210	15,664	4,186	4,186	15,019	15,882	1,409	0,114	0,350	0,975
10 x 50	18,765	21,154	23,938	11,766	11,766	23,805	24,438	2,036	0,093	0,456	1,354
20 x 20	8,088	9,658	11,025	3,613	3,613	10,330	10,856	1,122	0,097	0,364	0,920
20 x 50	14,010	15,160	19,111	9,279	9,279	18,134	19,049	1,804	0,075	0,492	1,214

Tabla 2. ARPD de las NEH(R1,Ai) agrupada por *n* y *m*

m x n	$\mathcal{R}_2, \mathcal{A}_1$	$\mathcal{R}_2, \mathcal{A}_2$	$\mathcal{R}_2, \mathcal{A}_3$	$\mathcal{R}_2, \mathcal{A}_4$	$\mathcal{R}_2, \mathcal{A}_5$	$\mathcal{R}_2, \mathcal{A}_6$	$\mathcal{R}_2, \mathcal{A}_7$	$\mathcal{R}_2, \mathcal{A}_8$	$\mathcal{R}_2, \mathcal{A}_9$	$\mathcal{R}_2, \mathcal{A}_{10}$	$\mathcal{R}_2, \mathcal{A}_{11}$
5 x 20	36,807	36,487	38,766	32,397	32,397	39,113	39,358	5,044	2,092	3,535	5,775
5 x 50	45,366	44,790	48,492	42,284	42,284	48,611	48,373	6,041	2,850	3,840	6,218
5 x 100	46,976	45,821	49,151	44,251	44,251	49,430	49,345	6,224	3,225	4,056	6,025
10 x 20	27,123	27,316	29,284	25,146	25,146	28,822	29,203	3,559	2,081	2,743	4,514
10 x 50	33,363	33,247	35,702	31,734	31,734	36,093	36,002	4,858	2,406	3,082	4,708
20 x 20	18,441	18,619	19,539	17,478	17,478	19,831	19,932	2,695	1,849	2,324	3,122
20 x 50	26,018	25,732	27,585	24,979	24,979	27,571	27,619	4,202	2,436	2,970	3,899

Tabla 3. ARPD de las NEH(R2,Ai) agrupada por *n* y *m*

F	$\mathcal{R}_1, \mathcal{A}_1$	$\mathcal{R}_1, \mathcal{A}_2$	$\mathcal{R}_1, \mathcal{A}_3$	$\mathcal{R}_1, \mathcal{A}_4$	$\mathcal{R}_1, \mathcal{A}_5$	$\mathcal{R}_1, \mathcal{A}_6$	$\mathcal{R}_1, \mathcal{A}_7$	$\mathcal{R}_1, \mathcal{A}_8$	$\mathcal{R}_1, \mathcal{A}_9$	$\mathcal{R}_1, \mathcal{A}_{10}$	$\mathcal{R}_1, \mathcal{A}_{11}$
2	14,677	14,677	17,905	11,409	11,409	16,983	15,550	1,149	0,018	1,239	1,239
3	17,245	18,789	21,183	11,257	11,257	21,280	21,559	1,575	0,060	0,590	2,178
4	18,883	19,839	23,691	10,546	10,546	22,618	24,544	1,882	0,082	0,475	1,106
5	18,269	20,864	23,981	9,753	9,753	23,370	26,205	2,130	0,153	0,279	1,502
6	17,515	21,460	24,368	9,515	9,515	24,043	26,147	2,238	0,145	0,221	0,914
7	17,597	21,488	24,279	8,553	8,553	23,744	26,013	2,286	0,197	0,211	1,203

Tabla 4. ARPD de las NEH(R1,Ai) agrupada por F

F	$\mathcal{R}_2,\mathcal{A}_1$	$\mathcal{R}_2,\mathcal{A}_2$	$\mathcal{R}_2,\mathcal{A}_3$	$\mathcal{R}_2,\mathcal{A}_4$	$\mathcal{R}_2,\mathcal{A}_5$	$\mathcal{R}_2,\mathcal{A}_6$	$\mathcal{R}_2,\mathcal{A}_7$	$\mathcal{R}_2,\mathcal{A}_8$	$\mathcal{R}_2,\mathcal{A}_9$	$\mathcal{R}_2,\mathcal{A}_{10}$	$\mathcal{R}_2,\mathcal{A}_{11}$
2	35,752	35,752	36,828	34,967	34,967	36,839	36,901	4,472	3,158	5,508	5,508
3	34,595	34,311	36,146	33,320	33,320	36,547	36,718	4,693	2,564	3,684	6,456
4	33,857	33,450	35,857	31,803	31,803	35,802	36,092	4,773	2,369	2,873	4,438
5	33,375	32,539	35,446	30,219	30,219	35,532	35,478	4,635	2,213	2,557	4,871
6	31,895	31,820	34,649	28,907	28,907	34,881	34,595	4,651	2,055	2,416	3,917
7	31,177	30,995	34,090	27,871	27,871	34,232	34,358	4,739	2,161	2,291	4,177

Tabla 5. ARPD de las NEH(R2,Ai) agrupada por F

m x n	$\mathcal{R}_1,\mathcal{A}_1$	$\mathcal{R}_1,\mathcal{A}_2$	$\mathcal{R}_1,\mathcal{A}_3$	$\mathcal{R}_1,\mathcal{A}_4$	$\mathcal{R}_1,\mathcal{A}_5$	$\mathcal{R}_1,\mathcal{A}_6$	$\mathcal{R}_1,\mathcal{A}_7$	$\mathcal{R}_1,\mathcal{A}_8$	$\mathcal{R}_1,\mathcal{A}_9$	$\mathcal{R}_1,\mathcal{A}_{10}$	$\mathcal{R}_1,\mathcal{A}_{11}$
5 x 20	0,0462	0,0366	0,0612	0,5862	0,2372	0,1275	0,0867	0,5346	0,5295	0,4831	0,3876
5 x 50	0,6490	0,5024	1,5262	15,8575	5,4327	3,3723	2,3139	27,9235	24,2126	22,6843	20,0540
5 x 100	4,9863	3,8329	20,0476	220,0970	69,9365	46,3601	31,9662	748,9315	580,7534	569,0447	525,6353
10 x 20	0,0673	0,0579	0,0962	0,9170	0,3543	0,1947	0,1315	0,8303	0,8361	0,7118	0,5743
10 x 50	0,9518	0,8071	2,4901	26,3946	8,8033	5,5006	3,7345	47,2780	40,5776	38,4857	33,5657
20 x 20	0,1096	0,1002	0,1627	1,6009	0,6105	0,3323	0,2224	1,4578	1,5133	1,2892	0,9858
20 x 50	1,5647	1,4183	4,3762	47,8485	16,0753	9,8590	6,6297	88,3713	72,6272	68,1563	59,5285

Tabla 6. ACT de las NEH(R1,Ai) agrupada por n y m

m x n	$\mathcal{R}_2,\mathcal{A}_1$	$\mathcal{R}_2,\mathcal{A}_2$	$\mathcal{R}_2,\mathcal{A}_3$	$\mathcal{R}_2,\mathcal{A}_4$	$\mathcal{R}_2,\mathcal{A}_5$	$\mathcal{R}_2,\mathcal{A}_6$	$\mathcal{R}_2,\mathcal{A}_7$	$\mathcal{R}_2,\mathcal{A}_8$	$\mathcal{R}_2,\mathcal{A}_9$	$\mathcal{R}_2,\mathcal{A}_{10}$	$\mathcal{R}_2,\mathcal{A}_{11}$
5 x 20	0,0029	0,0029	0,0007	0,0050	0,0049	0,0010	0,0011	0,0169	0,0170	0,0132	0,0079
5 x 50	0,0137	0,0136	0,0029	0,0233	0,0234	0,0049	0,0048	0,1793	0,1794	0,1339	0,0797
5 x 100	0,0473	0,0472	0,0102	0,0842	0,0840	0,0174	0,0176	1,2568	1,2553	0,9193	0,5448
10 x 20	0,0047	0,0047	0,0009	0,0077	0,0078	0,0016	0,0016	0,0275	0,0274	0,0210	0,0124
10 x 50	0,0231	0,0230	0,0048	0,0390	0,0392	0,0079	0,0079	0,3061	0,3063	0,2257	0,1331
20 x 20	0,0084	0,0084	0,0018	0,0135	0,0135	0,0027	0,0026	0,0489	0,0488	0,0370	0,0218
20 x 50	0,0416	0,0415	0,0088	0,0710	0,0715	0,0144	0,0143	0,5673	0,5676	0,4127	0,2429

Tabla 7. ACT de las NEH(R2,Ai) agrupada por n y m

F	$\mathcal{R}_1,\mathcal{A}_1$	$\mathcal{R}_1,\mathcal{A}_2$	$\mathcal{R}_1,\mathcal{A}_3$	$\mathcal{R}_1,\mathcal{A}_4$	$\mathcal{R}_1,\mathcal{A}_5$	$\mathcal{R}_1,\mathcal{A}_6$	$\mathcal{R}_1,\mathcal{A}_7$	$\mathcal{R}_1,\mathcal{A}_8$	$\mathcal{R}_1,\mathcal{A}_9$	$\mathcal{R}_1,\mathcal{A}_{10}$	$\mathcal{R}_1,\mathcal{A}_{11}$
2	0,5776	0,5083	3,7089	14,3588	7,4707	7,1384	4,9364	152,3094	100,4881	99,5518	99,6783
3	0,7960	0,6778	3,8530	24,0314	10,0895	8,0077	5,5176	137,3944	99,6115	100,0907	82,7429
4	1,0331	0,8556	4,0453	35,6380	12,9238	8,9219	6,1314	129,2979	100,0967	100,9901	93,5858
5	1,3104	1,0530	4,2055	49,1072	15,8666	9,8766	6,7565	125,1426	102,4711	96,8276	87,0715
6	1,5880	1,2483	4,3505	64,1755	18,6529	10,7366	7,3411	121,2802	105,1789	99,9736	93,2476
7	1,8733	1,4474	4,4886	81,2335	21,9534	11,6730	7,9612	119,1416	110,1964	103,2992	92,8721

Tabla 8. ACT de las NEH(R1,Ai) agrupada por F

F	$\mathcal{R}_2, \mathcal{A}_1$	$\mathcal{R}_2, \mathcal{A}_2$	$\mathcal{R}_2, \mathcal{A}_3$	$\mathcal{R}_2, \mathcal{A}_4$	$\mathcal{R}_2, \mathcal{A}_5$	$\mathcal{R}_2, \mathcal{A}_6$	$\mathcal{R}_2, \mathcal{A}_7$	$\mathcal{R}_2, \mathcal{A}_8$	$\mathcal{R}_2, \mathcal{A}_9$	$\mathcal{R}_2, \mathcal{A}_{10}$	$\mathcal{R}_2, \mathcal{A}_{11}$
2	0,0081	0,0081	0,0039	0,0109	0,0110	0,0054	0,0053	0,2361	0,2362	0,1198	0,1201
3	0,0126	0,0125	0,0041	0,0184	0,0184	0,0061	0,0060	0,2759	0,2758	0,1817	0,0936
4	0,0173	0,0173	0,0042	0,0276	0,0275	0,0067	0,0068	0,3188	0,3185	0,2329	0,1565
5	0,0225	0,0223	0,0044	0,0382	0,0383	0,0075	0,0076	0,3630	0,3632	0,2803	0,1422
6	0,0277	0,0277	0,0045	0,0501	0,0503	0,0082	0,0082	0,4085	0,4080	0,3247	0,1956
7	0,0331	0,0332	0,0046	0,0636	0,0638	0,0090	0,0089	0,4574	0,4569	0,3715	0,1856

Tabla 9. ACT de las NEH(R2,Ai) agrupada por F

Se observa que las heurísticas que obtienen valores más pequeños del índice ARPD coinciden con los resultados obtenidos en la sección anterior, donde se analizaban las instancias más pequeñas. Es decir, las reglas A8, A9, A10 y A11 para ambas codificaciones, R1 y R2, siguen siendo las mejores para instancias más grandes, manteniéndose siempre por debajo del 10% del ARPD. Además, la R1A4 y R1A5 son las que más se acercan a estas cuatro últimas reglas, al igual que en las instancias pequeñas.

Para complementar el análisis, a continuación, se incluye un diagrama de caja y bigotes que ofrece una representación visual de los índices RPD de cada heurística evaluada, tal como se realizó en la sección anterior.

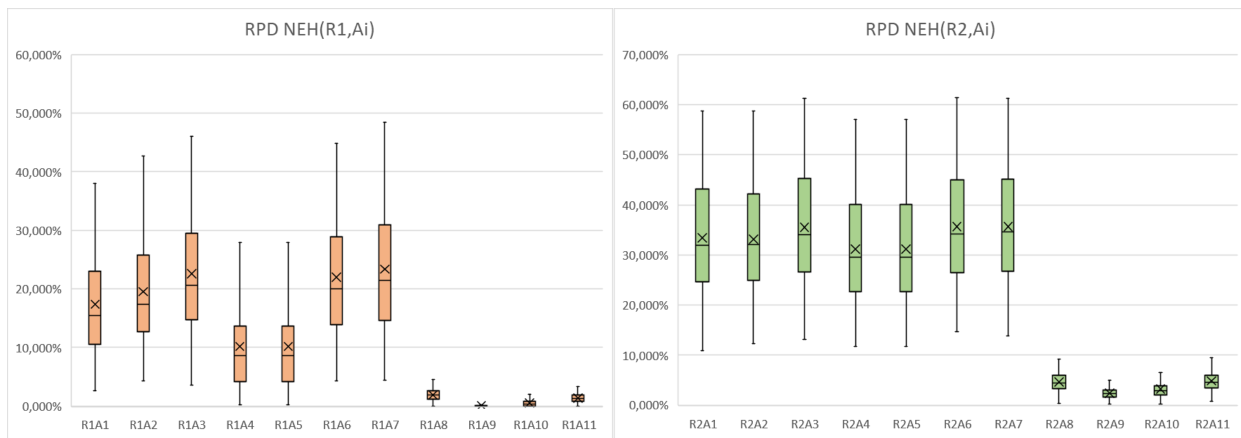

Ilustración 9. Diagrama de caja y bigote para instancias grandes (RPD)

La heurística R1A9 es la que ofrece mejores resultados en cuanto a ARPD. La regla de asignación utilizada, A9, coincide con la ganadora, A4, de (Fernandez-Viagas et al., 2018), pero adaptada al problema estudiado.

Sin embargo, esta heurística presenta unos tiempos de cómputo elevados para instancias más grandes. Los tiempos promedio de cómputo (ACT) superan los 500 segundos para $m = 5$ y $n = 100$. Aunque este tiempo puede parecer razonable, es importante compararlo con los tiempos de cómputo para el tamaño previo ($m = 20$ y $n = 50$), donde los ACT eran alrededor de 60 y 70 segundos. Esta escalada significativa en los tiempo de cómputo provoca que, para el resto de instancias de (Taillard, 1993) que quedan por analizar, estas heurísticas se vuelvan muy ineficientes en términos de tiempos de cómputo.

Para visualizar esto de manera más clara y concisa, se presenta la *Tabla 10* en una gráfica, donde se muestra el ARPD frente al ACT para cada una de las heurísticas aplicadas.

NEH(.,.)	ARPD	ACT	NEH(.,.)	ARPD	ACT
$\mathcal{R}_1, \mathcal{A}_1$	17,364	1,1964	$\mathcal{R}_2, \mathcal{A}_1$	33,442	0,0202
$\mathcal{R}_1, \mathcal{A}_2$	19,520	0,9651	$\mathcal{R}_2, \mathcal{A}_2$	33,144	0,0202
$\mathcal{R}_1, \mathcal{A}_3$	22,568	4,1086	$\mathcal{R}_2, \mathcal{A}_3$	35,503	0,0043
$\mathcal{R}_1, \mathcal{A}_4$	10,172	44,7574	$\mathcal{R}_2, \mathcal{A}_4$	31,181	0,0348
$\mathcal{R}_1, \mathcal{A}_5$	10,172	14,4928	$\mathcal{R}_2, \mathcal{A}_5$	31,181	0,0349
$\mathcal{R}_1, \mathcal{A}_6$	22,006	9,3924	$\mathcal{R}_2, \mathcal{A}_6$	35,639	0,0071
$\mathcal{R}_1, \mathcal{A}_7$	23,336	6,4407	$\mathcal{R}_2, \mathcal{A}_7$	35,690	0,0071
$\mathcal{R}_1, \mathcal{A}_8$	1,877	130,7610	$\mathcal{R}_2, \mathcal{A}_8$	4,661	0,3433
$\mathcal{R}_1, \mathcal{A}_9$	0,109	103,0071	$\mathcal{R}_2, \mathcal{A}_9$	2,420	0,3431
$\mathcal{R}_1, \mathcal{A}_{10}$	0,502	100,1222	$\mathcal{R}_1, \mathcal{A}_{10}$	3,222	0,2518
$\mathcal{R}_1, \mathcal{A}_{11}$	1,357	91,5330	$\mathcal{R}_1, \mathcal{A}_{11}$	4,894	0,1489

Tabla 10. ARPD y ACT para instancias grandes

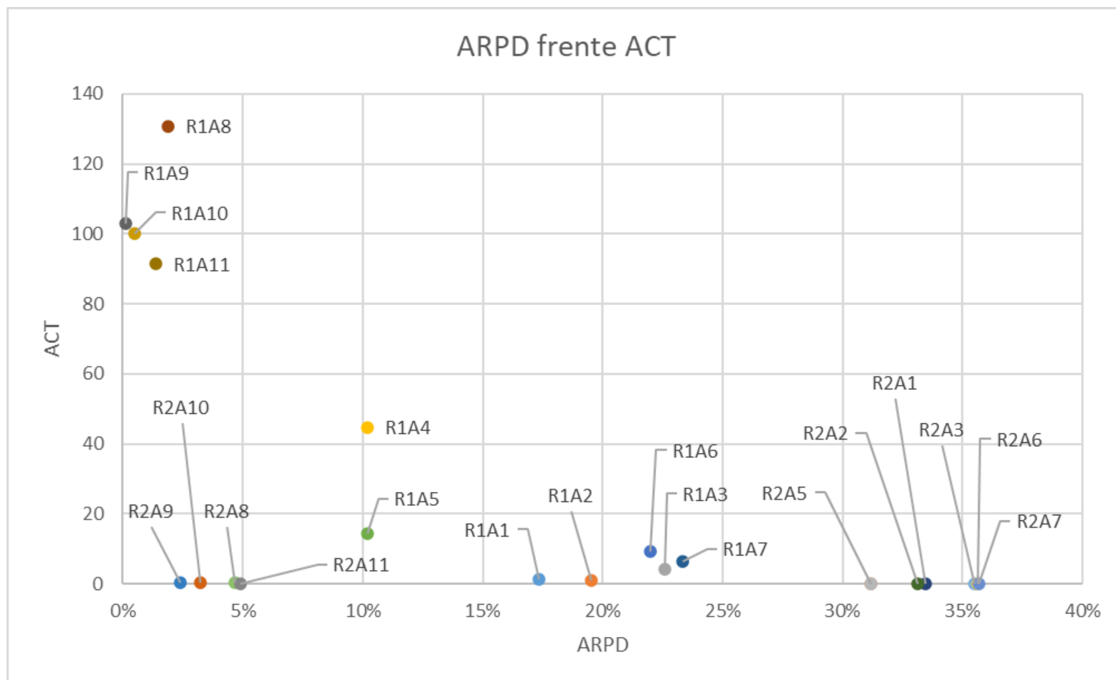

Ilustración 10. ARPD frente ACT para instancias grandes

La *Ilustración 10* confirma la información previamente discutida. Dado que los tiempos de cómputo escalan considerablemente para la codificación R1 en las instancias restantes, se analizarán estas instancias exclusivamente en la codificación R2. Este análisis servirá principalmente para ver los tiempos de cómputo de estas heurísticas en instancias muy grandes, ya que, al no analizar las heurísticas con la codificación R1, los índices ARPD nos servirán únicamente para confirmar que las reglas A8, A9, A10 y A11 siguen siendo las mejores con respecto al resto en las instancias más grandes de (Taillard, 1993).

Los resultados del análisis de las ultimas 50 instancias se ven reflejados de igual forma que se representaron las primeras 70, según el índice ARPD agrupadas por los parámetros *m* y *n*, en la *Tabla 11* y agrupadas por el parámetro *F* en la *Tabla 12*. Los tiempos de cómputo, que realmente es lo que más interesa de este análisis vienen reflejados en la *Tabla 13* y *Tabla 14*.

m x n	$\mathcal{R}_2, \mathcal{A}_1$	$\mathcal{R}_2, \mathcal{A}_2$	$\mathcal{R}_2, \mathcal{A}_3$	$\mathcal{R}_2, \mathcal{A}_4$	$\mathcal{R}_2, \mathcal{A}_5$	$\mathcal{R}_2, \mathcal{A}_6$	$\mathcal{R}_2, \mathcal{A}_7$	$\mathcal{R}_2, \mathcal{A}_8$	$\mathcal{R}_2, \mathcal{A}_9$	$\mathcal{R}_2, \mathcal{A}_{10}$	$\mathcal{R}_2, \mathcal{A}_{11}$
10 x 100	33,684	33,397	36,228	32,591	32,591	35,900	35,875	2,518	0,100	0,721	2,181
10 x 200	34,603	33,747	35,935	33,136	33,136	35,866	35,755	2,667	0,079	0,745	1,946
20 x 100	22,761	22,606	24,574	22,283	22,283	24,521	24,674	1,994	0,036	0,578	1,495
20 x 200	26,623	26,258	27,906	25,898	25,898	27,909	28,065	2,166	0,061	0,586	1,568
20 x 500	27,270	26,616	27,855	26,428	26,428	27,692	27,836	2,005	0,025	0,442	1,326

Tabla 11. ARPD de las NEH(R2,Ai) en las últimas 50 instancias agrupada por n y m

F	$\mathcal{R}_2, \mathcal{A}_1$	$\mathcal{R}_2, \mathcal{A}_2$	$\mathcal{R}_2, \mathcal{A}_3$	$\mathcal{R}_2, \mathcal{A}_4$	$\mathcal{R}_2, \mathcal{A}_5$	$\mathcal{R}_2, \mathcal{A}_6$	$\mathcal{R}_2, \mathcal{A}_7$	$\mathcal{R}_2, \mathcal{A}_8$	$\mathcal{R}_2, \mathcal{A}_9$	$\mathcal{R}_2, \mathcal{A}_{10}$	$\mathcal{R}_2, \mathcal{A}_{11}$
2	28,155	28,155	28,973	27,782	27,782	28,735	28,743	1,352	0,007	1,714	1,714
3	28,882	28,876	30,000	28,363	28,363	29,897	29,978	1,917	0,008	0,690	2,658
4	29,172	28,688	30,587	28,552	28,552	30,608	30,525	2,145	0,053	0,395	1,216
5	29,286	28,703	30,931	28,222	28,222	30,956	30,854	2,675	0,070	0,386	1,812
6	29,161	28,441	31,230	27,793	27,793	30,885	31,216	2,679	0,116	0,229	1,233
7	29,272	28,286	31,276	27,692	27,692	31,185	31,332	2,851	0,107	0,274	1,586

Tabla 12. ARPD de las NEH(R2,Ai) en las últimas 50 instancias agrupada por F

m x n	$\mathcal{R}_2, \mathcal{A}_1$	$\mathcal{R}_2, \mathcal{A}_2$	$\mathcal{R}_2, \mathcal{A}_3$	$\mathcal{R}_2, \mathcal{A}_4$	$\mathcal{R}_2, \mathcal{A}_5$	$\mathcal{R}_2, \mathcal{A}_6$	$\mathcal{R}_2, \mathcal{A}_7$	$\mathcal{R}_2, \mathcal{A}_8$	$\mathcal{R}_2, \mathcal{A}_9$	$\mathcal{R}_2, \mathcal{A}_{10}$	$\mathcal{R}_2, \mathcal{A}_{11}$
10 x 100	0,0819	0,0823	0,0177	0,1454	0,1446	0,0299	0,0297	2,0837	2,0839	1,5910	0,9378
10 x 200	0,1492	0,1504	0,0324	0,2689	0,2669	0,0545	0,0547	3,8686	3,8620	2,9518	1,7332
20 x 100	0,3052	0,3048	0,0675	0,5615	0,5592	0,1160	0,1150	15,7491	15,7712	11,9579	7,0305
20 x 200	0,5597	0,5585	0,1223	1,0447	1,0417	0,2142	0,2130	29,3660	29,3728	22,3556	13,1081
20 x 500	3,4372	3,4470	0,7560	6,6707	6,6860	1,3561	1,3692	464,4038	464,0118	351,8633	207,0111

Tabla 13. ACT de las NEH(R2,Ai) en las últimas 50 instancias agrupada por n y m

F	$\mathcal{R}_2, \mathcal{A}_1$	$\mathcal{R}_2, \mathcal{A}_2$	$\mathcal{R}_2, \mathcal{A}_3$	$\mathcal{R}_2, \mathcal{A}_4$	$\mathcal{R}_2, \mathcal{A}_5$	$\mathcal{R}_2, \mathcal{A}_6$	$\mathcal{R}_2, \mathcal{A}_7$	$\mathcal{R}_2, \mathcal{A}_8$	$\mathcal{R}_2, \mathcal{A}_9$	$\mathcal{R}_2, \mathcal{A}_{10}$	$\mathcal{R}_2, \mathcal{A}_{11}$
2	0,3898	0,3895	0,1935	0,5050	0,5190	0,2485	0,2471	71,4637	71,6768	36,0599	36,1624
3	0,6131	0,6017	0,1963	0,8757	0,8936	0,2897	0,2877	83,3045	83,9474	56,2000	28,2051
4	0,8057	0,8098	0,1964	1,3547	1,3626	0,3369	0,3398	97,6952	97,8517	73,4230	48,8951
5	0,9947	1,0116	0,2024	1,9063	1,8988	0,3771	0,3818	109,7320	109,1996	87,1711	43,8817
6	1,2126	1,2211	0,2029	2,5347	2,5085	0,4149	0,4117	122,1026	121,1130	100,8782	60,6951
7	1,4240	1,4178	0,2035	3,2531	3,2555	0,4579	0,4698	134,2674	134,3335	115,1313	57,9456

Tabla 14. ACT de las NEH(R2,Ai) en las últimas 50 instancias agrupada por F

Este análisis final nos permite ver como la heurística R2A9 obtiene el mejor resultado en cuanto al índice ARPD de todas las heurísticas que emplean la codificación R2. Además, al considerar los tiempos de cómputo, incluso para las instancias más grandes de (Taillard, 1993), el ACT es menor que el de las heurísticas R1A8, R1A9, R1A10 y R1A11 en la instancia más grande de las 70 anteriores.

Dado que no se ha realizado el análisis de las R1Ai para estas últimas instancias, la comparación con las R2Ai puede considerarse algo injusta. Sin embargo, atendiendo a lo dicho anteriormente en *Fundamentos teóricos,* las decisiones tomadas en la Programación de la Producción son mayoritariamente a corto plazo, por lo tanto, se busca tanto buenas soluciones como tiempos de cómputo reducido, y se ha demostrado que la codificación R2, especialmente, la heurística R2A9, ha proporcionado buenas soluciones y tiempos de cómputo razonables.

No obstante, es importante considerar también la heurística R1A9, que ha sido la clara ganadora en cuanto a calidad de las soluciones hasta el estudio realizado, aunque ha sido inferior en términos de tiempo de cómputo

en comparación con la heurística R2A9.

Independientemente de la codificación empleada, la regla de asignación A9 ha aportado las mejores soluciones a lo largo de todas las pruebas realizadas. Por tanto, la decisión de qué heurística elegir depende únicamente del usuario y de sus preferencias en cuanto a la calidad de las soluciones y el tiempo de cómputo.

6 CONCLUSIONES

En el transcurso de este proyecto se ha estudiado el entorno Distributed Flowshop, compuesto por un conjunto de fábricas idénticas que operan como un Flowshop de permutación y teniendo en cuenta los tiempos de transporte post-procesado de cada uno de los trabajos. Cada fábrica dispone de un conjunto idéntico de máquinas dispuestas en serie, por las cuales los trabajos pasan en el mismo orden. Este entorno ha sido objeto de estudio por numerosos investigadores debido a la complejidad y relevancia del problema en la actualidad. Sin embargo, este proyecto ha introducido un enfoque innovador con la inclusión de conceptos como el Delivery Time y el Total Delivery Time. Además, se ha aplicado un enfoque totalmente diferente a los tradicionales en cuanto a los tiempos de transporte post-procesado, ya que, se ha considerado que un trabajo se entrega inmediatamente después de finalizar su procesamiento, sin formar lotes de trabajos. Esta consideración ha añadido una capa adicional de complejidad y realismo al problema, destacando la importancia de gestionar eficientemente los tiempos de transporte en la organización de la producción.

Para abordar el problema, se ha adaptado la heurística NEH, diferenciando dos representaciones de las soluciones, propuestas por (Fernandez-Viagas et al., 2018): R1, que consiste en una única secuencia de permutación de los trabajos y R2, que es una secuencia en forma de matriz, la cual contiene las secuencias completas por cada fábrica. Esto ha dado lugar a 22 heurísticas distintas con el objetivo de encontrar el mejor método de resolución. El rendimiento de estas heurísticas se ha evaluado mediante el ARPD (Average Relative Percentage Deviation), que ha medido la desviación respecto a la mejor solución obtenida, y el ACT (Average CPU Time), que ha medido los tiempos de cómputo.

Estas heurísticas, programadas mediante lenguaje de programación Python, se han aplicado inicialmente a un conjunto de instancias pequeñas, lo que ha proporcionado una visión del comportamiento de cada una de ellas y nos ha servido como guía para las instancias de mayor tamaño. Posteriormente, se ha resuelto un conjunto de instancias grandes, basadas en las instancias de (Taillard, 1993), dónde se ha podido observar que, hasta un tamaño de 100 trabajos y 5 máquinas, la heurística R1A9 ha sido superior al resto en términos de ARPD, pero con unos tiempos de cómputo muy elevados en comparación a las demás heurísticas.

Por esta misma razón, el análisis final se ha centrado en las heurísticas codificadas mediante R2, cuyo tiempo de cómputo ha sido bastante más razonable para instancias mucho mayores. Finalmente, se ha llegado a la conclusión de que la mejor solución en este caso dependerá del usuario y sus preferencias, R1A9 ofrece soluciones superiores al resto con tiempos de cómputo elevados, mientras que R2A9 proporciona soluciones ligeramente inferiores con tiempos de cómputo mucho más reducidos. Independientemente de la representación de la solución elegida, la regla de asignación que ha aportado mejores soluciones ha sido la A9. Esta regla consiste en asignar los trabajos a la fábrica cuyo valor del Total Delivery Time (TDT) global sea menor después de insertar el trabajo en la mejor posición posible. Además, esta regla de asignación coincide con la ganadora del estudio de (Fernandez-Viagas et al., 2018) en la cual se asignaba el trabajo a la fábrica con el menor Total Flowtime global después de insertar el trabajo en la mejor posición.

Como se ha mencionado anteriormente, este estudio sigue abierto y los métodos propuestos pueden ser mejorados. Además, debido a los altos tiempos de cómputo para la codificación R1 en las últimas 50 instancias de (Taillard, 1993), sería necesaria una alta capacidad computacional para poder realizar estos experimentos. Por tanto, para comparar con certeza plena ambas heurísticas, R1A9 y R2A9, se debe terminar este análisis en su totalidad. Las conclusiones obtenidas en este proyecto se basan únicamente en lo estudiado hasta ahora, y futuras investigaciones podrán ampliar y refinar estos hallazgos si disponen de la capacidad computacional necesaria para terminar los análisis.

BIBLIOGRAFÍA

Garey, M. R., Johnson, D. S., & Sethi, R. (1976). *COMPLEXITY OF FLOWSHOP AND JOBSHOP SCHEDULING.* Mathematics of Operations Research, 1(2), 117-129. https://doi.org/10.1287/moor.1.2.117

Librería SCHEDULE – Grupo de Investigación Organización Industrial. (n.d.). Retrieved April 21, 2024, from http://grupo.us.es/oindustrial/investigacion/software-y-librerias/libreria-schedule/

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

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

Gonzalez, P., Torres, P. F., & Manuel, J. (2023). *Programación y Control de la Producción 4º GRADO EN INGENIERIA DE ORGANIZACIÓN INDUSTRIAL*.

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

Hou, Y., Fu, Y., Gao, K., Zhang, H., & Sadollah, A. (2022). Modelling and optimization of integrated distributed flow shop scheduling and distribution problems with time windows. *Expert Systems with Applications*, *187*. https://doi.org/10.1016/j.eswa.2021.115827

Li, Q., Li, J., Zhang, X., & Zhang, B. (2021). A wale optimization algorithm for distributed flow shop with batch delivery. *Soft Computing*, *25*(21), 13181–13194. https://doi.org/10.1007/s00500-021-06099-0

Li, Q. Y., Pan, Q. K., Sang, H. Y., Jing, X. L., Framiñán, J. M., & Li, W. M. (2024). Self-Adaptive Population-Based Iterated Greedy Algorithm for Distributed Permutation Flowshop Scheduling Problem with Part of Jobs Subject to a Common Deadline Constraint. *Expert Systems with Applications*, *248*. https://doi.org/10.1016/j.eswa.2024.123278

Liu, J., & Reeves, C. R. (2001). *Theory and Methodology Constructive and composite heuristic solutions to the P// P C i scheduling problem*. www.elsevier.com/locate/dsw

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

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

Norvig, P. (1991). *Technical Correspondence Techniques for Automatic Memoization with Applications to Context-Free Parsing*.

Ohno, T. (1988). *Toyota Production System Beyond Large-Scale Production*.

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

Taillard, E. (1993). Benchmarks for basic scheduling problems. In *European Journal of Operational Research* (Vol. 64).

ANEXO A: CÓDIGO DE PROGRAMACIÓN

En el presente Anexo se muestra el código completo utilizado para la ejecución de las diferentes heurísticas explicadas en este documento.

- Archivo ClaseModel.py.

```
from memo import check_memo_dt, update_memo_dt, check_memo_ct, update_memo_ct
from scheptk.scheptk import FlowShop
from scheptk.scheptk import Model
from scheptk.util import read_tag, print_tag, sorted_index_desc
import sys # to stop the execution (function exit())

class DistributedFlowShop(Model):

    def __init__(self, filename):

        # Leemos la instancia FlowShop
        self.flowshopinst=FlowShop(filename)

        # initializing additional data (not basic)
        self.machines = 0

        # starting reading
        print("----- Reading Distributed FlowShop instance data from file " + filename + " -------")

        # jobs (mandatory data)
        self.jobs = read_tag(filename, "JOBS")

        # if jobs = -1 the program cannot continue
        if (self.jobs==-1):
            print("No jobs specified. The program cannot continue.")
            sys.exit()
        else:
            print_tag("JOBS", self.jobs)

        # machines (another mandatory data)
        self.machines = read_tag(filename, "MACHINES")
```

```
# if machines = -1 the program cannot continue
if(self.machines ==-1):
   print("No machines specified. The program cannot continue.")
   sys.exit()
else:
   print_tag("MACHINES", self.machines)

# factories (another mandatory data)
self.factories = read_tag(filename, "FACTORIES")

if (self.factories==-1):
   print("No factories specified. The program cannot continue.")
   sys.exit()
else:
   print_tag("FACTORIES", self.factories)

# processing times (mandatory data, machines in rows, jobs in cols)
self.pt = read_tag(filename, "PT")

if(self.pt==-1):
   print("No processing times specified. The program cannot continue.")
   sys.exit()
else:
   if(len(self.pt) != self.machines ):
      print("Number of processing times does not match the number of machines (MACHINES={}, length of PT={}). The program cannot continue".format(self.machines, len(self.pt)))
      sys.exit()
   else:
      for i in range(self.machines):
         if(len(self.pt[i]) != self.jobs):
            print("Number of processing times does not match the number of jobs for machine {} (JOBS={}, length of col={}). The program cannot continue".format(i, self.jobs, len(self.pt[i])))
            sys.exit()
      print_tag("PT", self.pt)

# tap times (mandatory data)
self.tap = read_tag(filename,"TAP")

if (self.tap==-1):
```

```
            print("No tap times specified. The program cannot continue.")
            sys.exit()
        else:
            if(len(self.tap) != self.factories):
                print("Number of tap times does not match the number of factories. The program cannot continue.")
                sys.exit()
            else:
                for i in range(self.factories):
                    if(len(self.tap[i]) != self.jobs):
                        print("Number of tap times does not match the number of jobs for machine {} (JOBS={}, length of col={}). The program cannot continue.".format(i, self.jobs, len(self.tap[i])))
                        sys.exit()
                print_tag("TAP", self.tap)

        print("----- end of Distributed FlowShop instance data from file " + filename + " -------")


    def ct(self,solution):

        # Obtenemos una lista completa de los trabajos en orden de secuencia
        seq_compl= [trabajo for maquina in solution for trabajo in maquina]

        # Breve comprobación de la secuencia
        if(len(set(seq_compl))!=len(seq_compl)):
            print("Trabajos repetidos en la secuencia")
            sys.exit()

        # Inicializamos los completion times a 0
        completion_time = [[0 for j in range(len(seq_compl))] for i in range(self.machines*self.factories)]

        # Calculamos los índices de los trabajos en seq_compl
        trabajo_indices = {trabajo: index for index, trabajo in enumerate(seq_compl)}

        factory=1
        maq_inicial=0
        factory_order=[]
        # Recorremos la secuencia con sublistas
        for maquina in solution:
```

```
        # Comprobamos si fabrica trabaja
        if(len(maquina)!=0):
          for trabajo in maquina:
            trabajo_index = trabajo_indices[trabajo]
            factory_order.append(factory-1)
            # Colocamos el primer trabajo de la fábrica en la primera máquina
            if trabajo==maquina[0]:
              completion_time[maq_inicial][trabajo_index] = self.pt[0][trabajo]
              # Colocamos el trabajo en el resto de máquinas
              for i in range(maq_inicial+1,self.machines*factory):
                completion_time[i][trabajo_index] = completion_time[i-1][trabajo_index] + self.pt[i-maq_inicial][trabajo]
            # Colocamos el resto de trabajos
            else:
              completion_time[maq_inicial][trabajo_index] = completion_time[maq_inicial][trabajo_index-1] + self.pt[0][trabajo]
              for i in range(maq_inicial+1,self.machines*factory):
                completion_time[i][trabajo_index] = max(completion_time[i][trabajo_index-1],completion_time[i-1][trabajo_index]) + self.pt[i-maq_inicial][trabajo]

        maq_inicial=self.machines*factory
        factory+=1

      return completion_time, seq_compl, factory_order

    def ctR1A1(self,solution):
      # Breve comprobación de la secuencia
      if(len(set(solution))!=len(solution)):
        print("Trabajos repetidos en la secuencia")
        sys.exit()

      secuencia=[[] for i in range(self.factories)]
      ct_f=[[0 for j in range(self.machines)] for i in range(self.factories)]
      for i in range(len(solution)):
        job=solution[i]
        makespan=[]
        # Creamos secuencias parciales en cada una de las posiciones
        ct_f_parciales=[]
        for f in range(self.factories):
          # Calculamos el completion time del último trabajo colocado
```

```
            ct_f_aux=[list(fila) for fila in ct_f]
            ct_f_aux[f][0]=ct_f_aux[f][0]+self.pt[0][job]
            for m in range(1,self.machines):
                ct_f_aux[f][m]=max(ct_f_aux[f][m-1],ct_f_aux[f][m])+self.pt[m][job]
            ct_f_parciales.append(ct_f_aux)
            makespan.append(max([ct_f_parciales[-1][f][self.machines-1] for f in range(self.factories)]))
        minimo=min(makespan)
        f_seleccionada=makespan.index(minimo)
        secuencia[f_seleccionada].append(job)
        ct_f=ct_f_parciales[f_seleccionada]
    completion_time, solution, factory_order = self.ct(secuencia)
    return completion_time, solution, factory_order

def ctR1A2(self,solution):

    # Breve comprobación de la secuencia
    if(len(set(solution))!=len(solution)):
        print("Trabajos repetidos en la secuencia")
        sys.exit()

    secuencia=[[] for i in range(self.factories)]
    ct_f=[[0 for j in range(self.machines)] for i in range(self.factories)]
    for i in range(len(solution)):
        job=solution[i]
        makespan=[]
        # Creamos secuencias parciales en cada una de las posiciones
        ct_f_parciales=[]
        for f in range(self.factories):
          # Calculamos el completion time del último trabajo colocado
            ct_f_aux=[list(fila) for fila in ct_f]
            ct_f_aux[f][0]=ct_f_aux[f][0]+self.pt[0][job]
            for m in range(1,self.machines):
                ct_f_aux[f][m]=max(ct_f_aux[f][m-1],ct_f_aux[f][m])+self.pt[m][job]
            ct_f_parciales.append(ct_f_aux)
            makespan.append(ct_f_parciales[-1][f][self.machines-1])
        f_seleccionada=makespan.index(min(makespan))
        secuencia[f_seleccionada].append(job)
        ct_f=ct_f_parciales[f_seleccionada]
```

```
        completion_time, solution, factory_order = self.ct(secuencia)
        return completion_time, solution, factory_order

    def ctR1A3(self,solution):

        # Breve comprobación de la secuencia
        if(len(set(solution))!=len(solution)):
            print("Trabajos repetidos en la secuencia")
            sys.exit()

        secuencia=[[] for i in range(self.factories)]
        for i in range(len(solution)):
            job=solution[i]
            # Comprobamos si está calculado
            valor=check_memo_ct(secuencia)
            if(valor==-1):
                ct_global, ct_local = self.Cmax(secuencia, "ct")
                A3=ct_local
                update_memo_ct(secuencia, ct_global, ct_local)
            else:
                A3=valor[:-1]
            minimo=min(A3)
            f_seleccionada=A3.index(minimo)
            secuencia[f_seleccionada].append(job)
        completion_time, solution, factory_order = self.ct(secuencia)
        return completion_time, solution, factory_order

    def ctR1A4(self,solution):

        # Breve comprobación de la secuencia
        if(len(set(solution))!=len(solution)):
            print("Trabajos repetidos en la secuencia")
            sys.exit()

        secuencia=[[] for i in range(self.factories)]
        for i in range(len(solution)):
            job=solution[i]
            A4=[]
            # Creamos secuencias parciales en cada una de las posiciones
```

```
        for f in range(self.factories):
            secuencia[f].append(job)
            # Obtengo la posición del trabajo en orden de secuencia
            aplanada = [trabajo for sublista in secuencia for trabajo in sublista]
            posicion=aplanada.index(job)
            DT=self.DTj(secuencia, "ct")[0][posicion]
            A4.append(DT)
            secuencia[f].pop(-1)
        f_seleccionada=A4.index(min(A4))
        secuencia[f_seleccionada].append(job)
    completion_time, solution, factory_order = self.ct(secuencia)
    return completion_time, solution, factory_order

def ctR1A5(self, solution):

    # Breve comprobación de la secuencia
    if(len(set(solution))!=len(solution)):
        print("Trabajos repetidos en la secuencia")
        sys.exit()

    secuencia=[[] for i in range(self.factories)]
    for i in range(len(solution)):
        job=solution[i]
        A5=[]
        # Creamos secuencias parciales en cada una de las posiciones
        sec_parciales=[]
        for fabrica,sec in enumerate(secuencia):
            permutacion=sec[:]
            permutacion.append(job)
            sec_aux=secuencia[:]
            sec_aux[fabrica]=permutacion[:]
            sec_parciales.append(sec_aux)
            # Comprobamos si está calculado
            valor=check_memo_dt(sec_parciales[-1])
            if(valor==-1):
                dt_global, dt_f = self.sumDTj(sec_parciales[-1], "ct")
                A5.append(dt_global)
                update_memo_dt(sec_parciales[-1], dt_global, dt_f)
```

```
            else:
                A5.append(valor[-1])
        f_seleccionada=A5.index(min(A5))
        secuencia[f_seleccionada].append(job)
    completion_time, solution, factory_order = self.ct(secuencia)
    return completion_time, solution, factory_order

def ctR1A6(self, solution):

    # Breve comprobación de la secuencia
    if(len(set(solution))!=len(solution)):
        print("Trabajos repetidos en la secuencia")
        sys.exit()

    secuencia=[[] for i in range(self.factories)]
    for i in range(len(solution)):
        job=solution[i]
        A6=self.DTjmax(secuencia, "ct")[1]
        minimo=min(A6)
        f_seleccionada=A6.index(minimo)
        secuencia[f_seleccionada].append(job)
    completion_time, solution, factory_order = self.ct(secuencia)
    return completion_time, solution, factory_order

def ctR1A7(self, solution):

    # Breve comprobación de la secuencia
    if(len(set(solution))!=len(solution)):
        print("Trabajos repetidos en la secuencia")
        sys.exit()

    secuencia=[[] for i in range(self.factories)]
    for i in range(len(solution)):
        job=solution[i]
        valor=check_memo_dt(secuencia)
        if(valor==-1):
            dt_global, dt_f = self.sumDTj(secuencia, "ct")
            A7=dt_f
            update_memo_dt(secuencia, dt_global, dt_f)
```

```
        else:
            A7=valor[:-1]
        minimo=min(A7)
        f_seleccionada=A7.index(minimo)
        secuencia[f_seleccionada].append(job)
    completion_time, solution, factory_order = self.ct(secuencia)
    return completion_time, solution, factory_order

def ctR1A8(self, solution):
    # Breve comprobación de la secuencia
    if(len(set(solution))!=len(solution)):
        print("Trabajos repetidos en la secuencia")
        sys.exit()

    secuencia=[[] for i in range(self.factories)]
    for i in range(len(solution)):
        job=solution[i]
        sec_parciales=[]
        A8=[]
        # Creamos secuencias parciales en cada una de las posiciones
        for fabrica,sec in enumerate(secuencia):
            for j in range(len(sec)+1):
                if len(sec)==0:
                    permutacion=[]
                else:
                    permutacion=sec[:]
                permutacion.insert(j,job)
                sec_aux=secuencia[:]
                sec_aux[fabrica]=permutacion[:]
                sec_parciales.append(sec_aux)
                valor=check_memo_dt(sec_parciales[-1])
                if(valor==-1):
                    dt_global, dt_f = self.sumDTj(sec_parciales[-1], "ct")
                    A8.append(dt_f[fabrica])
                    update_memo_dt(sec_parciales[-1], dt_global, dt_f)
                else:
                    A8.append(valor[fabrica])
        minimo=min(A8)
```

```
        secuencia=sec_parciales[A8.index(minimo)]
    completion_time, solution, factory_order = self.ct(secuencia)
    return completion_time, solution, factory_order

def ctR1A9(self, solution):
    # Breve comprobación de la secuencia
    if(len(set(solution))!=len(solution)):
        print("Trabajos repetidos en la secuencia")
        sys.exit()

    secuencia=[[] for i in range(self.factories)]
    for i in range(len(solution)):
        job=solution[i]
        sec_parciales=[]
        A9=[]
        # Creamos secuencias parciales en cada una de las posiciones
        for fabrica,sec in enumerate(secuencia):
            for j in range(len(sec)+1):
                if len(sec)==0:
                    permutacion=[]
                else:
                    permutacion=sec[:]
                permutacion.insert(j,job)
                sec_aux=secuencia[:]
                sec_aux[fabrica]=permutacion[:]
                sec_parciales.append(sec_aux)
                valor=check_memo_dt(sec_parciales[-1])
                if(valor==-1):
                    dt_global, dt_f = self.sumDTj(sec_parciales[-1], "ct")
                    A9.append(dt_global)
                    update_memo_dt(sec_parciales[-1], dt_global, dt_f)
                else:
                    A9.append(valor[-1])
        minimo=min(A9)
        secuencia=sec_parciales[A9.index(minimo)]
    completion_time, solution, factory_order = self.ct(secuencia)
    return completion_time, solution, factory_order

def ctR1A10(self, solution):
```

```
# Breve comprobación de la secuencia
if(len(set(solution))!=len(solution)):
    print("Trabajos repetidos en la secuencia")
    sys.exit()

secuencia=[[] for i in range(self.factories)]
for i in range(len(solution)):
    job=solution[i]
    criterio=check_memo_dt(secuencia)
    if(criterio==-1):
        dt_global, dt_f = self.sumDTj(secuencia, "ct")
        criterioDescarte=dt_f
        update_memo_dt(secuencia, dt_global, dt_f)
    else:
        criterioDescarte=criterio[:-1]
    f_descartada=criterioDescarte.index(max(criterioDescarte))
    sec_parciales=[]
    A10=[]
    # Creamos secuencias parciales en cada una de las posiciones
    for fabrica,sec in enumerate(secuencia):
        if fabrica==f_descartada:
            continue # Paso a la siguiente fábrica
        for j in range(len(sec)+1):
            if len(sec)==0:
                permutacion=[]
            else:
                permutacion=sec[:]
            permutacion.insert(j,job)
            sec_aux=secuencia[:]
            sec_aux[fabrica]=permutacion[:]
            sec_parciales.append(sec_aux)
            valor=check_memo_dt(sec_parciales[-1])
            if(valor==-1):
                dt_global, dt_f = self.sumDTj(sec_parciales[-1], "ct")
                A10.append(dt_global)
                update_memo_dt(sec_parciales[-1], dt_global, dt_f)
            else:
```

```
            A10.append(valor[-1])
        minimo=min(A10)
        secuencia=sec_parciales[A10.index(minimo)]
    completion_time, solution, factory_order = self.ct(secuencia)
    return completion_time, solution, factory_order

def ctR1A11(self, solution):

    # Breve comprobación de la secuencia
    if(len(set(solution))!=len(solution)):
        print("Trabajos repetidos en la secuencia")
        sys.exit()

    n_descartadas=self.factories-int(self.factories/2) # cantidad de f descartadas

    secuencia=[[] for i in range(self.factories)]
    for i in range(len(solution)):
        job=solution[i]

        # Determinamos las fábricas con peor sumDTj para descartarlas
        criterio=check_memo_dt(secuencia)
        if(criterio==-1):
            dt_global, dt_f = self.sumDTj(secuencia, "ct")
            criterioDescarte=dt_f
            update_memo_dt(secuencia, dt_global, dt_f)
        else:
            criterioDescarte=criterio[:-1]
        f_orden=sorted_index_desc(criterioDescarte) # f ordenadas de mayor a menor sumDTj
        f_descartadas=[f_orden[i] for i in range(n_descartadas)]
        sec_parciales=[]
        A11=[]
        # Creamos secuencias parciales en cada una de las posiciones
        for fabrica,sec in enumerate(secuencia):
            if(any(f==fabrica for f in f_descartadas)):
                continue # Paso a la siguiente fábrica
            for j in range(len(sec)+1):
                if len(sec)==0:
                    permutacion=[]
                else:
```

```
                permutacion=sec[:]
            permutacion.insert(j,job)
            sec_aux=secuencia[:]
            sec_aux[fabrica]=permutacion[:]
            sec_parciales.append(sec_aux)
            valor=check_memo_dt(sec_parciales[-1])
            if(valor==-1):
                dt_global, dt_f = self.sumDTj(sec_parciales[-1], "ct")
                A11.append(dt_global)
                update_memo_dt(sec_parciales[-1], dt_global, dt_f)
            else:
                A11.append(valor[-1])
        minimo=min(A11)
        secuencia=sec_parciales[A11.index(minimo)]
    completion_time, solution, factory_order = self.ct(secuencia)
    return completion_time, solution, factory_order

def Cj(self, solution, ct_func):
    ct, job_order, factory_order = eval("self."+ ct_func + "(solution)")
    Cj = [max([ct[i][j] for i in range(len(ct))]) for j in range(len(job_order))]
    return Cj, factory_order, job_order

def DTj(self, solution, ct_func):
    Cj, factory_order, job_order=self.Cj(solution, ct_func)
    DT=[0 for i in range(len(Cj))]
    TAP=self.tap
    DT_f=[[] for i in range(self.factories)]

    for i,factory in enumerate(factory_order):
        job=job_order[i]
        DT[i]=Cj[i]+TAP[factory][job]
        DT_f[factory].append(DT[i])

    # Comprobación de fábricas vacías
    if(len(DT)==0):
        DT.append(0)
    for sublista in DT_f:
        if len(sublista)==0:
```

```
            sublista.append(0)
        return DT, DT_f

    def DTjmax(self, solution, ct_func):
        DT, DT_f = self.DTj(solution, ct_func)
        maxDTjGlobal=max(DT)
        maxDTj_F=[max(sublista) for sublista in DT_f]
        return maxDTjGlobal, maxDTj_F

    def Cmax(self, solution, ct_func):
        instancia=self.flowshopinst
        Cmax_f=[0 for i in range(self.factories)]
        for f in range(self.factories):
            if(len(solution[f])!=0):
                Cmax_f[f]=instancia.Cmax(solution[f])
        return max(Cmax_f), Cmax_f

    def sumDTj(self, solution, ct_func):
        DT, DT_f = self.DTj(solution, ct_func)
        sumDTjGlobal= sum(DT)
        sumDTj_F=[sum(sublista) for sublista in DT_f]
        return sumDTjGlobal, sumDTj_F
```

- Archivo DispatchingRules.py

```
from scheptk.util import sorted_index_desc
from memo import resetmemo
class DispatchingRules:

    # Algoritmo NEH que sirve para todas las R1Ai
    def NEHR1Ai(instancia, ct_func):
        # Ordenamos de mayor a menor pt
        pt_total=[]
        for i in range(instancia.jobs):
            suma=sum(sublista[i] for sublista in instancia.pt)
            pt_total.append(suma)
        orden=sorted_index_desc(pt_total)
        # Asignamos el primero de la lista a la secuencia
        sec=[]
        sec.append(orden[0])
        # Comenzamos bucle para asignar el resto de trabajos
```

```
    for i in range(1,len(orden)):
        job=orden[i]
        sec_parciales=[]
        objetivo=[]
        # Creamos secuencias parciales en todas las posiciones
        for j in range(len(sec)+1):
            permutacion=sec[:]
            permutacion.insert(j,job)
            sec_parciales.append(permutacion)
            # Criterio de asignación
            objetivo.append(instancia.sumDTj(sec_parciales[-1], ct_func)[0])
        minimo=min(objetivo)
        sec=sec_parciales[objetivo.index(minimo)]
    # Reiniciamos el diccionario de memoización
    resetmemo()
    return sec

def NEHR2A1(instancia, ct_func):
    # Ordenamos de mayor a menor pt
    pt_total=[]
    for i in range(instancia.jobs):
        suma=sum(sublista[i] for sublista in instancia.pt)
        pt_total.append(suma)
    orden=sorted_index_desc(pt_total)

    secuencia=[[] for i in range(instancia.factories)]
    for i in range(len(orden)):
        job=orden[i]
        sec_parciales=[]
        objetivo=[]
        # Creamos secuencias parciales en todas las posiciones
        for fabrica,sec in enumerate(secuencia):
            permutacion=sec[:]
            permutacion.append(job)
            sec_aux=secuencia[:]
            sec_aux[fabrica]=permutacion[:]
            sec_parciales.append(sec_aux)
            # Criterio de asignación
```

```
        objetivo.append(instancia.Cmax(sec_parciales[-1], ct_func)[0])
      minimo=min(objetivo)
      secuencia=sec_parciales[objetivo.index(minimo)]
    return secuencia

def NEHR2A2(instancia, ct_func):
    # Ordenamos de mayor a menor pt
    pt_total=[]
    for i in range(instancia.jobs):
      suma=sum(sublista[i] for sublista in instancia.pt)
      pt_total.append(suma)
    orden=sorted_index_desc(pt_total)

    secuencia=[[] for i in range(instancia.factories)]
    for i in range(len(orden)):
      job=orden[i]
      sec_parciales=[]
      objetivo=[]
      # Creamos secuencias parciales en todas las posiciones
      for fabrica,sec in enumerate(secuencia):
        permutacion=sec[:]
        permutacion.append(job)
        sec_aux=secuencia[:]
        sec_aux[fabrica]=permutacion[:]
        sec_parciales.append(sec_aux)
        # Criterio de asignación
        objetivo.append(instancia.Cmax(sec_parciales[-1], ct_func)[1][fabrica])
      minimo=min(objetivo)
      secuencia=sec_parciales[objetivo.index(minimo)]
    return secuencia

def NEHR2A3(instancia, ct_func):
    # Ordenamos de mayor a menor pt
    pt_total=[]
    for i in range(instancia.jobs):
      suma=sum(sublista[i] for sublista in instancia.pt)
      pt_total.append(suma)
    orden=sorted_index_desc(pt_total)
```

```
    secuencia=[[] for i in range(instancia.factories)]
    for i in range(len(orden)):
        job=orden[i]
        # Criterio de asignación a la fábrica
        objetivo=instancia.Cmax(secuencia, ct_func)[1]
        minimo=min(objetivo)
        f_seleccionada=objetivo.index(minimo)
        secuencia[f_seleccionada].append(job)
    return secuencia

def NEHR2A4(instancia, ct_func):
    # Ordenamos de mayor a menor pt
    pt_total=[]
    for i in range(instancia.jobs):
        suma=sum(sublista[i] for sublista in instancia.pt)
        pt_total.append(suma)
    orden=sorted_index_desc(pt_total)

    secuencia=[[] for i in range(instancia.factories)]
    for i in range(len(orden)):
        job=orden[i]
        sec_parciales=[]
        objetivo=[]
        # Creamos secuencias parciales en todas las posiciones
        for fabrica,sec in enumerate(secuencia):
            permutacion=sec[:]
            permutacion.append(job)
            sec_aux=secuencia[:]
            sec_aux[fabrica]=permutacion[:]
            sec_parciales.append(sec_aux)
            aplanada = [trabajo for sublista in sec_parciales[-1] for trabajo in sublista]
            posicion=aplanada.index(job)
            # Criterio de asignación
            objetivo.append(instancia.DTj(sec_parciales[-1], ct_func)[0][posicion])
        minimo=min(objetivo)
        secuencia=sec_parciales[objetivo.index(minimo)]
    return secuencia
```

```
def NEHR2A5(instancia, ct_func):
    # Ordenamos de mayor a menor pt
    pt_total=[]
    for i in range(instancia.jobs):
        suma=sum(sublista[i] for sublista in instancia.pt)
        pt_total.append(suma)
    orden=sorted_index_desc(pt_total)

    secuencia=[[] for i in range(instancia.factories)]
    for i in range(len(orden)):
        job=orden[i]
        sec_parciales=[]
        objetivo=[]
        # Creamos secuencias parciales en todas las posiciones
        for fabrica,sec in enumerate(secuencia):
            permutacion=sec[:]
            permutacion.append(job)
            sec_aux=secuencia[:]
            sec_aux[fabrica]=permutacion[:]
            sec_parciales.append(sec_aux)
            # Criterio de asignación
            objetivo.append(instancia.sumDTj(sec_parciales[-1],ct_func)[0])
        minimo=min(objetivo)
        secuencia=sec_parciales[objetivo.index(minimo)]
    return secuencia

def NEHR2A6(instancia, ct_func):
    # Ordenamos de mayor a menor pt
    pt_total=[]
    for i in range(instancia.jobs):
        suma=sum(sublista[i] for sublista in instancia.pt)
        pt_total.append(suma)
    orden=sorted_index_desc(pt_total)

    secuencia=[[] for i in range(instancia.factories)]
    for i in range(len(orden)):
        job=orden[i]
        A6=instancia.DTjmax(secuencia, "ct")[1]
        minimo=min(A6)
```

```
        f_seleccionada=A6.index(minimo)
        secuencia[f_seleccionada].append(job)
    return secuencia

def NEHR2A7(instancia, ct_func):
    # Ordenamos de mayor a menor pt
    pt_total=[]
    for i in range(instancia.jobs):
        suma=sum(sublista[i] for sublista in instancia.pt)
        pt_total.append(suma)
    orden=sorted_index_desc(pt_total)

    secuencia=[[] for i in range(instancia.factories)]
    for i in range(len(orden)):
        job=orden[i]
        A7=instancia.sumDTj(secuencia, "ct")[1]
        minimo=min(A7)
        f_seleccionada=A7.index(minimo)
        secuencia[f_seleccionada].append(job)
    return secuencia

def NEHR2A8(instancia, ct_func):
    # Ordenamos de mayor a menor pt
    pt_total=[]
    for i in range(instancia.jobs):
        suma=sum(sublista[i] for sublista in instancia.pt)
        pt_total.append(suma)
    orden=sorted_index_desc(pt_total)

    secuencia=[[] for i in range(instancia.factories)]
    for i in range(len(orden)):
        job=orden[i]
        sec_parciales=[]
        A8=[]
        # Creamos secuencias parciales en todas las posiciones
        for fabrica,sec in enumerate(secuencia):
            for j in range(len(sec)+1):
                if len(sec)==0:
```

```
                permutacion=[]
            else:
                permutacion=sec[:]
            permutacion.insert(j,job)
            sec_aux=secuencia[:]
            sec_aux[fabrica]=permutacion[:]
            sec_parciales.append(sec_aux)
            # Criterio de asignación
            A8.append(instancia.sumDTj(sec_parciales[-1], "ct")[1][fabrica])
        minimo=min(A8)
        secuencia=sec_parciales[A8.index(minimo)]
    return secuencia

def NEHR2A9(instancia, ct_func):
    # Ordenamos de mayor a menor pt
    pt_total=[]
    for i in range(instancia.jobs):
        suma=sum(sublista[i] for sublista in instancia.pt)
        pt_total.append(suma)
    orden=sorted_index_desc(pt_total)

    secuencia=[[] for i in range(instancia.factories)]
    for i in range(len(orden)):
        job=orden[i]
        sec_parciales=[]
        A9=[]
        # Creamos secuencias parciales en todas las posiciones
        for fabrica,sec in enumerate(secuencia):
            for j in range(len(sec)+1):
                if len(sec)==0:
                    permutacion=[]
                else:
                    permutacion=sec[:]
                permutacion.insert(j,job)
                sec_aux=secuencia[:]
                sec_aux[fabrica]=permutacion[:]
                sec_parciales.append(sec_aux)
                # Criterio de asignación
                A9.append(instancia.sumDTj(sec_parciales[-1], "ct")[0])
```

```
        minimo=min(A9)
        secuencia=sec_parciales[A9.index(minimo)]
    return secuencia

def NEHR2A10(instancia, ct_func):
    # Ordenamos de mayor a menor pt
    pt_total=[]
    for i in range(instancia.jobs):
        suma=sum(sublista[i] for sublista in instancia.pt)
        pt_total.append(suma)
    orden=sorted_index_desc(pt_total)

    secuencia=[[] for i in range(instancia.factories)]
    for i in range(len(orden)):

        # Determinamos la fábrica con peor sumDTj para descartarla
        criterioDescarte=instancia.sumDTj(secuencia, "ct")[1]
        f_descartada=criterioDescarte.index(max(criterioDescarte))

        job=orden[i]
        sec_parciales=[]
        A10=[]
        # Creamos secuencias parciales en todas las posiciones
        for fabrica,sec in enumerate(secuencia):
            if fabrica==f_descartada:
                continue # Paso a la siguiente fábrica
            for j in range(len(sec)+1):
                if len(sec)==0:
                    permutacion=[]
                else:
                    permutacion=sec[:]
                permutacion.insert(j,job)
                sec_aux=secuencia[:]
                sec_aux[fabrica]=permutacion[:]
                sec_parciales.append(sec_aux)
                # Criterio de asignación
                A10.append(instancia.sumDTj(sec_parciales[-1], "ct")[0])
        minimo=min(A10)
```

```
        secuencia=sec_parciales[A10.index(minimo)]
    return secuencia

def NEHR2A11(instancia, ct_func):
    # Ordenamos de mayor a menor pt
    pt_total=[]
    for i in range(instancia.jobs):
        suma=sum(sublista[i] for sublista in instancia.pt)
        pt_total.append(suma)
    orden=sorted_index_desc(pt_total)

    n_descartadas=instancia.factories-int(instancia.factories/2) # cantidad de f descartadas

    secuencia=[[] for i in range(instancia.factories)]
    for i in range(len(orden)):

        # Determinamos la fábrica con peor sumDTj para descartarla
        criterioDescarte=instancia.sumDTj(secuencia, "ct")[1]
        f_orden=sorted_index_desc(criterioDescarte) # f ordenadas de mayor a menor sumDTj
        f_descartadas=[f_orden[i] for i in range(n_descartadas)]

        job=orden[i]
        sec_parciales=[]
        A11=[]
        # Creamos secuencias parciales en todas las posiciones
        for fabrica,sec in enumerate(secuencia):
            if(any(f==fabrica for f in f_descartadas)):
                continue # Paso a la siguiente fábrica
            for j in range(len(sec)+1):
                if len(sec)==0:
                    permutacion=[]
                else:
                    permutacion=sec[:]
                permutacion.insert(j,job)
                sec_aux=secuencia[:]
                sec_aux[fabrica]=permutacion[:]
                sec_parciales.append(sec_aux)
                # Criterio de asignación
                A11.append(instancia.sumDTj(sec_parciales[-1], "ct")[0])
```

```
            minimo=min(A11)
            secuencia=sec_parciales[A11.index(minimo)]
        return secuencia
```

- Archivo memo.py

```
memo = {}
memo_ct ={}



def check_memo_dt(secuencia):
    clave=str(secuencia)
    # Devuelve el valor cuando ya fue calculada y -1 en cc
    if clave in memo:
        return memo[clave]
    else:
        return -1

def update_memo_dt(secuencia, valorglobal, valorlocal):
    clave=str(secuencia)
    valorlocal.append(valorglobal)
    memo[clave]=valorlocal

def check_memo_ct(secuencia):
    clave=str(secuencia)
    # Devuelve el valor cuando ya fue calculada y -1 en cc
    if clave in memo_ct:
        return memo_ct[clave]
    else:
        return -1

def update_memo_ct(secuencia, valorglobal, valorlocal):
    clave=str(secuencia)
    valorlocal.append(valorglobal)
    memo_ct[clave]=valorlocal

def resetmemo():
    global memo
```

```
global memo_ct
memo={}
memo_ct={}
```