Trabajo Fin de Grado

Grado en Ingeniería de Tecnologías Industriales

Análisis del problema Distributed FlowShop con permutación y objetivo de transportes

Autor: Rafael García García

Tutor: Paz Pérez González

Dpto. de Organización Industrial y Gestión de Empresas I

Escuela Técnica Superior de Ingeniería

Sevilla, 2024

Trabajo Fin de Grado
Ingeniería en Ingeniería de Tecnologías Industriales

Análisis del problema Distributed FlowShop con permutación y objetivo de transportes

Autor:
Rafael García García

Tutor:
Paz Pérez Gonzaéz

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

Proyecto Fin de Carrera: Análisis del problema Distributed FlowShop con permutación y objetivo de transportes

Autor: Rafael García García

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

Agradecimientos

A mis padres, a mi familia y a mis amigos por haberme acompañado a lo largo de estos años y siempre apoyarme.

Rafael García García

Sevilla, 2023

Resumen

En este trabajo de fin de grado se estudia el entorno de *Distributed Flowshop.* Este entorno está formado por un conjunto de fábricas, siendo a su vez, cada una de ellas un entorno *Flowshop*, que se caracteriza por estar constituido por máquinas en serie y donde todos los trabajos siguen la misma ruta. El objetivo de este estudio es el de minimizar el *total weighted completion time*, así como reducir los costes de transporte. Para ello a lo largo del documento se presentarán los conceptos básicos necesarios para abordar el problema, se adaptarán métodos de resolución ya existentes al problema planteado y se hará un estudio detallado del rendimiento de estos algoritmos para el objetivo que se plantea. Tras el análisis de estos métodos de resolución se llega a la conclusión de que el algoritmo genético es el que proporciona mejores resultados.

Índice

ÍNDICE DE TABLAS

ÍNDICE DE FIGURAS

1 INTRODUCCIÓN Y OBJETIVOS DEL ESTUDIO

Según (Fu et al., 2022) en los últimos años la globalización y el aumento de la frecuencia de envíos ha provocado que las empresas empiecen a considerar el transporte como una parte importante de su cadena de suministro. El transporte permite la conexión entre proveedores y clientes y en un entorno empresarial cada vez más competitivo la velocidad de entrega juega un factor importante a la hora de mejorar la satisfacción media de los clientes y la respuesta frente a emergencias. Con una buena gestión de la cadena de suministro que se centre en la unión de los procesos de producción y distribución conseguiremos además una reducción en los costes operativos, como son el combustible, seguros o el mantenimiento de vehículos entre otros.

1.1 Objetivo

El objetivo de este estudio es analizar, de manera aproximada a través de metaheurísticas, el problema del entorno *Distributed Flow Shop* con restricción de permutación y con objetivo de minimizar los costes de transporte y el *total weighted completion time*.

El entorno Distributed Flow Shop se caracteriza por estar formado por un conjunto de fábricas siendo cada una de ellas un entorno de tipo taller con máquinas en serie y donde cada una de ellas tiene un propósito diferente. Además, el problema propuesto cuenta con la restricción de permutación, es decir, que la secuencia que siguen los trabajos a través de las máquinas es la misma para todas ellas.

Para resolver el problema se plantean en primer lugar cuatro reglas de asignación que serán evaluadas a través de cinco reglas de despacho básicas. Tras esto se propondrán tres metaheurísticas que serán comparadas entre si para analizar cuál de ellas proporciona mejores resultados.

1.2 Estructura

El presente documento está compuesto por seis capítulos:

- En el primer capítulo se hace una introducción del problema que se tratará, se hace una descripción del objetivo del estudio y se plantea la estructura que seguirá el documento
- En el capítulo segundo se hace una introducción a la programación de operaciones donde se definen los conceptos básicos de esta, la forma de representar las soluciones, los diferentes métodos de resolución que existen para un modelo y los distintos entornos, restricciones y objetivos que se pueden encontrar en el ámbito de la programación de operaciones.
- En el capítulo tres se describe el problema en función a los conceptos previamente descritos en el capítulo anterior. Se detalla el entorno de trabajo, las restricciones que tendrá el problema y de presenta el objetivo del problema.
- En el capítulo cuatro son descritos los diferentes métodos de resolución planteados para resolver el problema.
- Posteriormente, en el capítulo 5, se presentan los resultados obtenidos y se analiza que método de los propuestos es el mejor.
- Finalmente, el capítulo 6 se exponen las conclusiones a las que se han llegado tras el estudio del problema.

2 INTRODUCCIÓN A LA PROGRAMACIÓN DE OPERACIONES

La programación de operaciones es un proceso mediante el cual se determinan el orden y los tiempos en que se realizarán las diferentes tareas o actividades necesarias para completar un proyecto o proceso específico. Este proceso se utiliza comúnmente en la gestión de proyectos y la gestión de la cadena de suministro y juega un papel importante en empresas dedicadas al transporte y distribución, así como en industrias dedicadas a sistemas de fabricación (Pinedo, 2012).

La programación de la producción no puede ser vista como un proceso independiente, sino que se encuentra integrada en un conjunto de decisiones de gestión que se denomina comúnmente gestión de la producción (Framinan et al., 2014). Un sistema muy conocido es el de planificación de las necesidades materiales (MRP). Una vez generada la programación es necesario que todos los materiales y recursos estén disponibles en el momento requerido. Tanto el sistema de programación de la producción como el sistema MRP deben colaborar en la determinación de las fechas de preparación de todas las tareas de manera conjunta (Pinedo, 2012).

Aunque las decisiones de programación pueden variar de una empresa a otra, todas ellas comparten una serie de puntos en común, estos son (Framinan et al., 2014):

- Son decisiones complejas, puesto que implica desarrollar planes detallados para asignar tareas a los recursos a lo largo del tiempo.
- Las decisiones de programación son decisiones que se toman a corto plazo y deben ser tomadas de manera recurrente.
- La programación determina plazos de entrega del producto y los costes que este implica, por lo que a pesar de ser una decisión a corto plazo, esta juega un papel importante en el nivel de servicio de una empresa.

2.1 Conceptos básicos

A continuación, se definen algunos conceptos básicos relacionados con la programación de operaciones junto a la notación normalmente utilizada:

- Trabajos: N= {1, 2 , …, n}. Son los productos que deben pasar por las operaciones de las máquinas de las fábricas. Los trabajos tienen el subíndice j, j ∈ N (Framinan et al., 2014).
- Máquinas: M= {1, 2, …, m}. Son los recursos productivos capaces de realizar operaciones de transformación y/o transporte de material. Las máquinas tienen el subíndice i, i ∈ M.
- Ruta: la ruta indica el orden en el que debe de procesarse el trabajo, es decir por que máquinas debe de pasar. Se denota como R_j.

  Es un vector de m componentes, formado por la m máquinas, habiendo un total de n rutas, correspondientes a los números de trabajos.
- Tiempo de proceso: es el tiempo que la máquina i ∈ M tarda en procesar el trabajo j ∈ N. La notación usada es p_{ij}.

  En el caso de que este sea independiente de la máquina se usa p_j.
- Fecha de llegada: es el instante de tiempo en el cual el trabajo puede empezar a ser procesado. Para las fechas de llegada usamos r_j.
- Fecha de entrega: es el momento en el que el trabajo *j* debe estar finalizado. Se denota mediante d_j.

- Pesos: es un factor de prioridad del trabajo, w_j.

2.2 Programa

Un programa o *Schedule* es una asignación en una escala temporal concreta de las máquinas de una empresa para la fabricación de un conjunto de trabajos. Un programa establece el comienzo y el final de cada operación a realizar por la máquina, no obstante, como los tiempos de proceso suelen ser conocidos, basta con obtener el tiempo de comienzo o finalización de la tarea. Sin embargo, cuando nos encontramos con operaciones interrumpibles esta información puede no ser suficiente y necesitaremos saber en que momento esta interrupción se llevará a cabo y cuando se retomará la operación. Cuando un programa cumple con todas las restricciones del proceso productivo hablamos de programa admisible. El objetivo principal de la programación de la producción es encontrar un programa admisible y el procedimiento para obtenerlos se denomina algoritmo. Cuando en un programa no es posible adelantar ninguna operación sin que interfiera en el orden en que se procesan los trabajos hablamos de programa semi-activo.

La forma más habitual de representar los programas es a través del diagrama de Gantt, donde en el eje horizontal se representa el momento de comienzo y final de todos los trabajos a realizar y en el eje vertical se representa la ruta en cada máquina (Framinan et al., 2014).

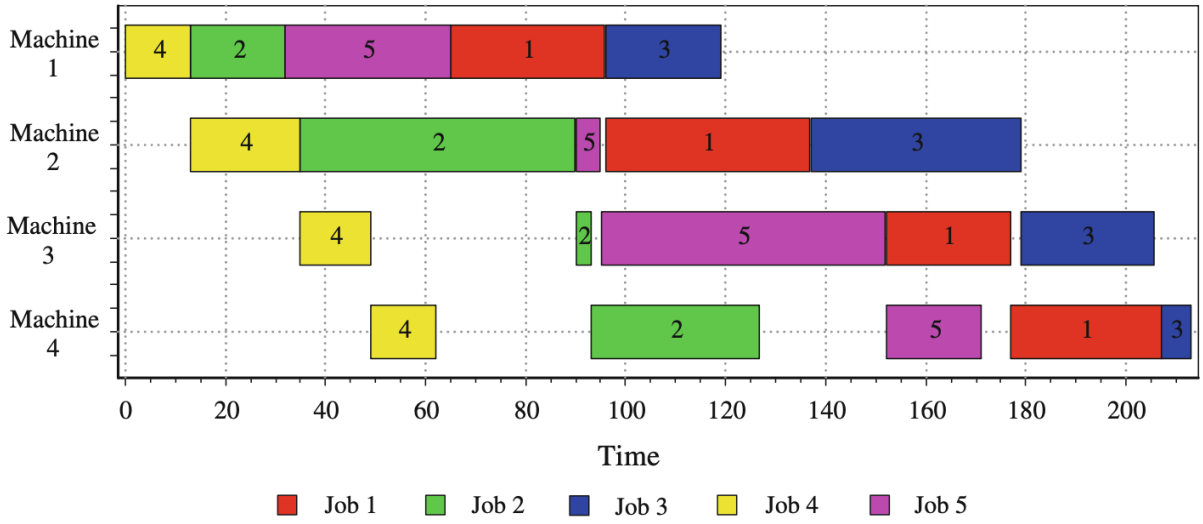

Ilustración 1. Diagrama de Gantt de un Flowshop (Framinan et al., 2014).

2.3 Métodos de programación

Un método de programación es un procedimiento formal que puede aplicarse a cualquier instancia de un modelo de programación con el fin de obtener un programa factible y que obtenga buenos resultados del valor de la función objetivo. Puesto que este procedimiento puede debe formal, puede describirse usando un número finitos de pasos. El nombre técnico de este procedimiento se conoce como algoritmo (Framinan et al., 2014). Estos algoritmos se pueden clasificar en dos grandes grupos, algoritmos exactos y algoritmos aproximados.

- Algoritmos exactos: proporcionan la solución óptima para el problema. Dentro de los algoritmos exactos existen dos tipos:
    - Algoritmos constructivos exactos: son polinomiales, proporcionan el óptimos para problemas específicos. Dentro de este tipo se encuentran algunas reglas de despacho, el algoritmo de Johnson, de Lawler y de Moore.
    - Algoritmos enumerativos: son no polinomiales, es decir, dan el óptimo para instancias pequeñas. Ejemplos de algoritmos enumerativos son la programación matemática, la enumeración completa y Branch and Bound.

- Algoritmos aproximados: estos algoritmos no garantizan la optimalidad.
    - Heurísticas constructivas: las heurísticas constructivas generan una solución una secuencia factible desde cero. En primer lugar, se ordenan los trabajos, después se elige un trabajo que todavía no ha sido programado y se inserta en los posibles huecos de la secuencia y por último, se guarda la mejor secuencia obtenida como punto de partida para seguir construyendo la secuencia.
    - Heurísticas de mejora: estas heurísticas tratan de obtener una solución mejor a partir de una secuencia de partida. Un ejemplo de estos algoritmos es la búsqueda local (*local search*).
    - Metaheurísticas: estas tratan de ampliar el espacio de búsqueda permitiendo el empeoramiento temporal de las soluciones que se considerarán para la siguiente búsqueda (Framinan et al., 2014).

2.4 Modelos: entorno, restricciones y objetivos

Los modelos de programación de la producción se definen a través de tres conceptos fundamentales, propuestos por Graham, Lawler, Lenstra y Rinnooy Kan, que son entorno, restricciones y objetivos. La notación usada para ello es $\alpha \mid \beta \mid \gamma$ (Graham et al., 1979).

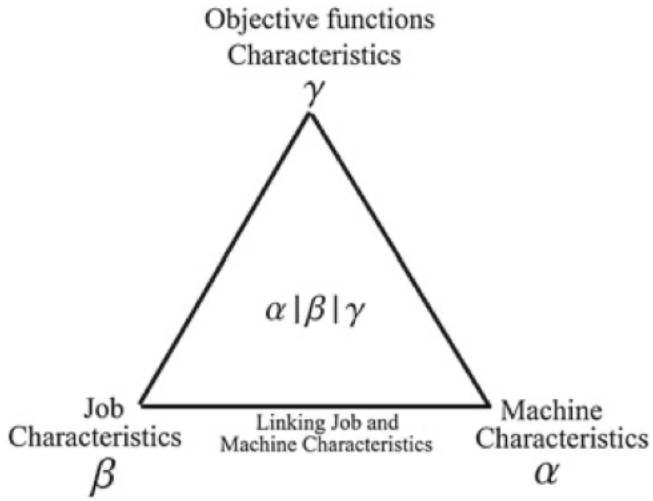

Ilustración 2. Clasificación de los modelos de producción (Framinan et al., 2014)

- Entorno (α): el entorno o *layout* indica el número de máquinas en la planta y la disposición de esta. Un buen *layout* ayudará a la empresa a incrementar la producción, disminuir el manejo de materiales y reducir costes. A continuación, se definen los diferentes entornos junto con la notación utilizada:
    - Una máquina o *single machine* ($\alpha = 1$): el entorno está formado por una sola máquina por la que pasan todos los trabajos, por lo que no existe una ruta.
    - Máquinas en paralelo o *parallel machines*: es un entorno formado por m máquinas. Cada trabajo tiene una sola operación y se procesa en una de las máquinas, normalmente todas las máquinas pueden procesar el trabajo. Las máquinas paralelas pueden ser a su vez divididas en tres categorías.
        - *Identical Parallel Machines* ($\alpha = P_m$): todas las máquinas son iguales, es decir, el tiempo de proceso p_j no depende de la máquina
        - *Uniform Parallel Machines* ($\alpha = Q_m$): en este caso las velocidades de las máquinas, v_i, son diferentes, lo que da lugar a tiempos de procesos diferentes, $p_{ij} = \frac{p_j}{vi}$. Un valor de v_i mayor a uno quiere decir que esa máquina es más rápida mientras que valores de v_i menores a uno representan máquinas más lentas.

        - *Unrelated Parallel Machines* ($\alpha = R_m$): se trata del caso más general, las máquinas son diferentes y el tiempo de proceso del trabajo depende de la máquina en la que es procesado, p_{ij}.
    - Taller de flujo regular *o Flow Shop* ($\alpha = F_m$)*:* se trata de un entorno formado por m máquinas, cada una de ellas con una función diferente. En este tipo de entornos todos los trabajos tienen que pasar por todas las máquinas, siguiendo la misma ruta $R_j = (1,2,\ldots,m)$. Aquí p_{ij} denota el tiempo que tarda cada máquina i en procesar el trabajo j. Un caso particular de este entorno es el *Permutation Flow Shop,* donde la secuencia es la misma para todas las máquinas.
    - Taller o *Job Shop* ($\alpha = J_m$)*:* aquí cada trabajo tiene una ruta diferente. Este tipo de entornos es común cuando la diferenciación entre productos es grande y cada trabajo requiere un tratamiento diferente.
    - Taller abierto u *Open Shop* ($\alpha = O_m$)*:* se trata del caso más general, en este entorno no hay una ruta especificada por lo que cuando se da el programa a su vez se construye una ruta.

- Restricciones (β): las restricciones nos condicionan la forma de procesar y decidir el orden de los trabajos. Aunque no siempre están presentes, hay que tener en cuentas algunas suposiciones generales sobre estas: los trabajos siempre están disponibles al principio del horizonte de programación, siempre y cuando no se tengan restricciones relacionadas con fechas de llegada, los trabajos no se pueden interrumpir, las máquinas siempre están disponibles, el *buffer* entre máquinas se supone infinito y el tiempo de transporte es despreciable. Algunas de las restricciones más comunes son:
    - Interrupciones: el trabajo puede interrumpirse una vez que ha comenzado a procesarse, esto se puede deber a llegado un nuevo trabajo que requiere ser procesado urgentemente, por cancelaciones de clientes o por avería de la máquina. Los tipos de interrupción son:
        - Non-resumable (β = *pmtn-non-resumable*): el trabajo hecho se pierde y se tiene que iniciar de nuevo.
        - Semi-resumable (β = *pmtn-semi-resumable*): sólo se pierde una parte del trabajo realizado.
        - Resumable (β = *pmtn -resumable*): el trabajo realizado se conserva y retoma por donde se había dejado tras la interrupción.
    - Fechas de llegada o *release date* ($\beta = r_j$): indica el instante de tiempo en el que el trabajo estará disponible para procesar.
    - Fechas de entrega de los trabajos: hay dos tipos de restricciones relacionadas con las fechas de entrega
        - *Deadlines* ($\beta = \overline{d_j}$): indica la fecha en la que, como muy tarde, el trabajo, obligatoriamente debe de ser entregado, para así no acarrear penalización.
        - *Common due date* ($\beta = \mathrm{d_j} = \mathrm{d}$): se trata de un número común a todos los trabajos y la entrega en este instante de tiempo no es obligatoria, a diferencia del caso anterior.
    - Tiempos de *setup*: es el tiempo que se requiere para preparar la máquina o el trabajo antes de ser procesado. Los tiempos de *setup* pueden anticipatorios o no anticipatorios, independientes de la secuencia ($\beta = \mathrm{s_{ij}}$)*,* dependientes de la secuencia ($\beta = \mathrm{s_{ijk}}$) o independientes de la secuencia, en este caso se elimina el subíndice i.
    - Procesado por lotes: el procesado por lotes puede resultar interesante cuando varios trabajos requieren el mismo *setup* y estos pueden ser procesados por lotes, las máquinas procesan a la vez lotes de hasta *b* trabajos. Por lo general, existen dos tipos de procesamiento por lotes:
        - Lotes en paralelo o *parallel batching* ($\beta = p - batch(b)$): el tiempo de procesamiento del lote se corresponde con el máximo de los tiempos de proceso de los trabajos que están en dicho lote.

        - Lotes en serie o *serial batching* ($\beta = s - batch(b)$): el tiempo de proceso es la suma de los tiempos de proceso de los trabajos en el lote.
    - Precedencia ($\beta = prec$): un trabajo no puede empezar a procesarse hasta que no acaben los trabajos que lo preceden.

  En los entornos tipo taller nos podemos encontrar las siguientes restricciones:

    - De permutación ($\beta = prmu$): donde la secuencia es la misma para todos los trabajos.
    - Tiempos ociosos no permitidos ($\beta = no - idle$): cuando la máquina empieza a procesar trabajos esta no puede parar.
    - Los trabajos no pueden esperar ($\beta = no - wait$): cuando un trabajo comienza a ser procesado este no puede esperar entre máquinas.
    - Almacenes de una máquina o buffer ($\beta = b_i$): es la cantidad de trabajos que pueden esperar en el almacén de una máquina.

- Objetivos (γ): es la función a optimizar. En programación de la producción las funciones objetivos suelen clasificarse según coste, tiempo, calidad y flexibilidad (Framinan et al., 2014).

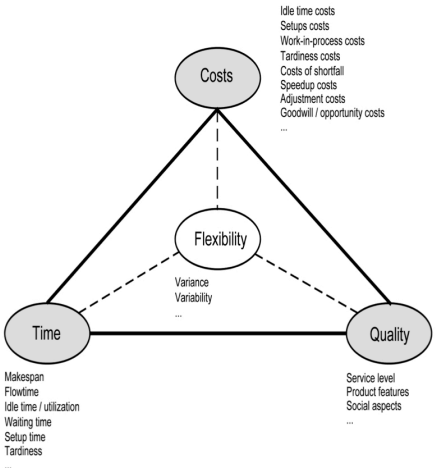

Ilustración 3 Clasificación de los objetivos (Framinan et al., 2014)

Algunos de los objetivos más comunes en programación de operaciones son los siguientes: el tiempo de finalización o *makespan* ($\gamma = C_{max}$), que es el momento en el que se terminan de procesar todos los trabajos, el *total completion time* ($\gamma = \sum_{j=1}^{n} C_j$), que es la suma de todos los tiempos de finalización de los trabajos, el *maximun flowtime* ($\gamma = máx\ F_j$), que hace referencia al tiempo que el trabajo se encuentra en el sistema siendo procesado o el *total flowtime* ($\gamma = \sum_{j=1}^{n} F_j$), que es la suma de los *flowtimes*. También hay objetivos relacionados con las fechas de entrega, como son: el *maximun lateness* ($\gamma = máx\ L_j$), que indica como de tarde se ha entregado el trabajo, el *total lateness* ($\gamma = \sum_{j=1}^{n} L_j$), que es la suma de los *lateness* de todos los trabajos, estos dos últimos objetivos pueden tomar valores negativos pues son el resultado de restar el tiempo de finalización y la fecha de entrega del trabajo, el *maximun tardiness* ($\gamma = máx\ T_j$) y el *total tardiness* ($\gamma = \sum_{j=1}^{n} T_j$), que indican lo mismo que los dos últimos objetivos solo que estos no pueden ser valores negativos, el *earliness* ($\gamma = máx\ E_j$), y el *total earliness* ($\gamma = \sum_{j=1}^{n} E_j$), que indican con cuanta antelación se entregan los trabajos y por último el *number of tardy job* que indica cuantos trabajos han sido entregados con retraso ($\gamma = \sum_{j=1}^{n} U_j$).

3 DESCRIPCIÓN DEL PROBLEMA

En este documento el entorno a estudiar se trata un *Distributed Permutation Flowshop*, que siguiendo la notación $\alpha|\beta|\gamma$, vista anteriormente, se define como $DF_m|prmu|F_i(\sum_{j=1}^{n} w_j C_j, \sum_{j=1}^{n}\sum_{f=1}^{f} w'_f x_{jf})$.

La programación distribuida, según (Toptal & Sabuncuoglu, 2010), se puede definir como un enfoque donde un problema es dividido en partes más pequeñas y donde estas partes se coordinan entre sí, utilizando la información más reciente del sistema, para alcanzar el objetivo global del problema. Un entorno distribuido generalmente responde de manera más efectiva ante cambios imprevistos en la producción, o eventos que no son controlables, como pueden ser averías en las máquinas, le llegada de trabajos con alta preferencia o cancelación de pedidos a última hora. A esta ventaja competitiva se le une el desarrollo de la globalización de la economía, por lo que en los últimos años cada vez son más las empresas que optan por un entorno distribuido para ahorrar costes de fabricación y entrega (Fernandez-Viagas et al., 2018). En comparación con los entornos tradicionales donde sólo hay fábrica, en los entornos distribuidos además de programar los trabajos en un conjunto de máquinas, entra en juego otra decisión, asignar los trabajos a la fábrica adecuada (Naderi & Ruiz, 2010).

A continuación, se describe más detalladamente el problema objeto de estudio, definiendo así el entorno, las restricciones y el objetivo de este.

- Entorno ($\alpha = DF_m$): el entorno *Distributed Flowshop* hay un conjunto *N* de *n* trabajos, que tienen que ser procesados en un conjunto *F* de *f* fábricas. Cada una de estas fábricas se trata de un *Flowshop* (descrito en el capítulo anterior) formado por un número *m* de máquinas, que es constante entre fábricas.

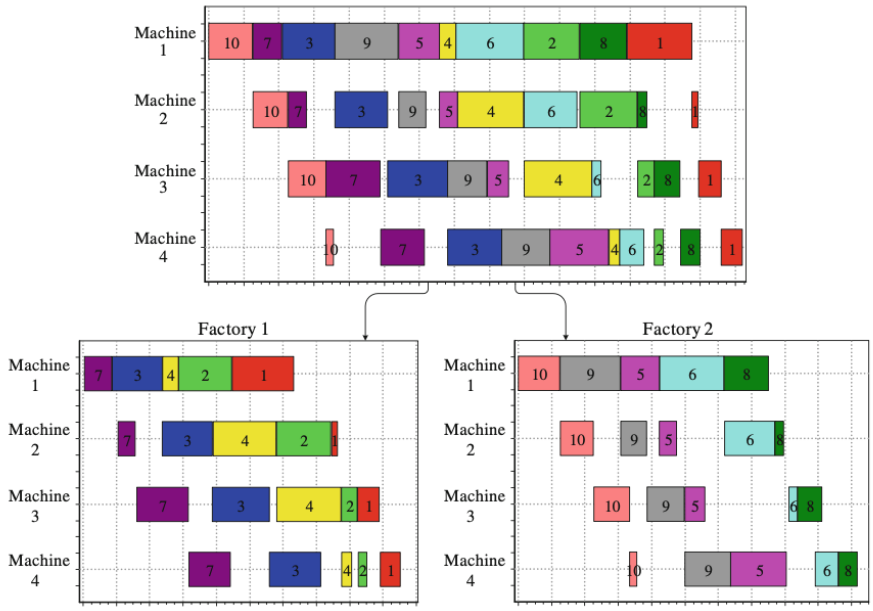

Ilustración 4. Ejemplo de Distributed Permutation Flowshop (Framinan et al., 2014).

En la *Ilustración 4* se puede observar como un entorno de *Permutation Flowshop* con diez trabajos y cuatro máquinas es dividido en un *Distributed Permutation Flowshop* con dos fábricas, cada una con cuatro máquinas.

- Restricciones ($\beta = prmu$): el problema a tratar tiene la restricción de permutación, es decir que todas las máquinas tienen la misma secuencia, por lo que el orden en el que se ejecutan los trabajos en cada máquina en el mismo para todas.

  Además de esta restricción hay que tener en cuenta otras restricciones generales: todos los trabajos están disponibles en el instante cero, así como las máquinas. No se disponen de fechas de entrega. Las fábricas son idénticas entre sí y capaces de procesar todos los trabajos. El tiempo de proceso del trabajo *j* en la máquina *i*, que definiremos como p_{ij}, es el mismo para todas las fábricas del conjunto. Una vez que un trabajo empieza a ser procesado en una fábrica, este debe ser acabado en esta, sin posibilidad de interrupción y de ser continuado en una fábrica diferente. El tiempo de transporte entre máquinas se considera despreciable y el buffer entre máquinas se considera infinito.

- Objetivo ($\gamma = F_i(\sum_{j=1}^{n} w_j C_j, \sum_{j=1}^{n}\sum_{f=1}^{f} w'_f x_{jf})$): el objetivo del problema consistirá en minimizar el *total weighted completion time* más la suma de los costes de transporte. Este objetivo tomará un valor normalizado entre 0 y 2. La variable w'_f representa los costes de transporte y habrá tantos como fábricas tenga el entorno que se esté estudiando. La variable x_{jf} de la función objetivo tomará un valor de 0 o 1 y solo se activará si el trabajo *j* está siendo procesado en la fábrica *f*. Por tanto, la función objetivo se puede escribir como:

$$F_i(\sum_{j=1}^{n} w_j C_j, \sum_{j=1}^{n}\sum_{f=1}^{f} w'_f x_{jf}) = \frac{\sum_{j=1}^{n} w_j C_j}{\sum_{j=1}^{n} w_j \cdot C_{máx}} + \frac{\sum_{j=1}^{n}\sum_{f=1}^{f} w\prime_f x_{jf} - f\cdot n\cdot min(w\prime_f)}{f\cdot n\cdot max(w\prime_f) - f\cdot n\cdot min(w\prime_f)}$$

3.1 Cálculo del objetivo

A continuación, se muestra mediante un ejemplo sencillo el cálculo de la función objetivo para un entorno con $f = 3$ fábricas, $m = 4$ máquinas y $j = 7$ trabajos, donde los costes de transporte de llevar los trabajos desde el centro de distribución a cada una de las tres fábricas son $w_f = [3,2,1]$.

		Trabajos						
	p_{ij}	0	1	2	3	4	5	6
Máquinas	0	5	3	4	2	3	1	1
	1	6	2	3	4	4	2	2
	2	2	2	6	5	8	7	3
	3	5	4	1	2	3	4	4
	w_j	1	2	3	1	2	1	4
	$\sum_{j=0}^{6}\sum_{i=0}^{3} p_{ij}$	18	11	14	13	18	14	10

Tabla 1. Tiempos de procesos del ejemplo propuesto

Para la asignación de trabajos a las fábricas se usará la regla de asignación ECT (*Earliest Completion Time*), por lo que los trabajos serán asignados a aquella fábrica cuyo tiempo de terminación tras asignar dicho trabajo sea menor. Se tomará como ejemplo la siguiente secuencia: S = [4,0,3,5,2,6,1].

Como puede observarse en *Ilustración 5*, al comienzo de la programación como todas las fábricas están libres, asignaremos el trabajo 4 a la fábrica 0, el trabajo 0 a la fábrica 1 y el trabajo 3 a la fábrica 2. Tras asignar estos tres primeros trabajos, el siguiente trabajo es el 5, si este es procesado en la fábrica 0, el *completion time* es 26 u.t, si va a la fábrica 1 este es de 24 u.t, mientras que si va a la fábrica 2 este acabará de procesarse en el instante 22. Viendo estos tiempos de finalización el trabajo 5, será procesado en la fábrica 2 puesto que es la fábrica que finaliza antes con el trabajo. Este proceso se usa a lo largo de toda la programación, obteniendo el diagrama de Gantt que se observa en la *Ilustración 5*.

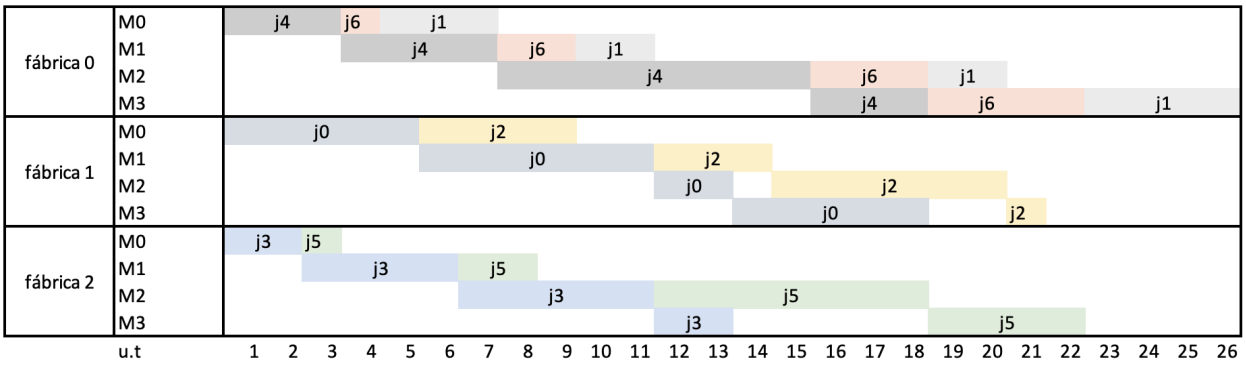

Ilustración 5. Ejemplo Distributed Flowshop *según regla de asignación ECT.*

Una vez hemos asignado todos los trabajos vemos que los *completion times* son [18,18,13,22,21,22,26]. Con estos datos podemos calcular la función objetivo de la siguiente forma:

$$FO = \frac{292}{14 \cdot 26} + \frac{15-7}{21-7} = 1{,}374$$

4 MÉTODOS DE RESOLUCIÓN

En este apartado se exponen las distintas reglas de asignación, reglas de despacho, heurísticas constructivas y metaheurísticas elegidas para resolver el problema.

4.1 Reglas de asignación

Para resolver este problema en primer lugar analizaremos qué regla de asignación a fábricas elegir. En este problema se plantean cuatro reglas de asignación que se describen a continuación:

- ECT (*earliest completion time*): consiste en asignar el trabajo *j* a la fábrica *f* cuyo tiempo de terminación es menor (*Ilustración 5*). En caso de empate el trabajo será asignado a la fábrica de menor índice.
- FAM (*first available machine*): esta regla de asignación consiste en asignar los trabajos a la primera fábrica que esté disponible. Esta puede ser utilizada para tomar decisiones rápidas y eficientes sobre como asignar trabajos a una fábrica. La regla de desempate usada en esta regla de asignación será la misma descrita anteriormente.
- ECT_minWF: esta regla de asignación sigue el mismo procedimiento que ECT solo que en el caso en el que un determinado trabajo tenga el mismo tiempo de finalización en varias fábricas, este irá a la fábrica cuyo coste de transporte sea menor.
- FAM_minWF: en este caso la forma de actuar es la misma que en la regla de asignación anterior, pero siguiendo la regla de asignación FAM.

4.2 Reglas de despacho o *priority rules*

Para evaluar que regla de asignación es mejor usaremos reglas de despacho o *priority rules.* Una regla de despacho se refiere a un conjunto de criterios utilizados para tomar decisiones sobre el orden en el que se ejecutarán las tareas. En este problema se plantean cinco:

- SPT (*shortest processing time*): consiste en ordenar los trabajos de menor a mayor tiempo de proceso. Para calcular estos tiempos de proceso sumaremos el tiempo que cada trabajo pasa en cada una de las máquinas. Aplicando esto a nuestro ejemplo de *Tabla 1* nuestra secuencia es: S_{SPT} = [6,1,3,2,5,0,4]. En caso de empate, se elige el trabajo con menor índice.
- LPT (*longest processing time*): consiste en ordenar los trabajos de mayor a menor tiempo de proceso. La forma de obtener estos tiempos de proceso es igual a la seguida en SPT. S_{LPT} = [0,4,2,5,3,1,6]. Si se produce un empate, el trabajo a insertar será el de menor índice.
- WSPT (weighted shortest processing time): el trabajo con mayor valor del cociente entre el peso y el tiempo de proceso se procesa primero. S_{WSPT} = [6,2,1,4,3,5,0]. La regla de desempate usada aquí será la misma que en SPT y LPT.
- SPT_INT: esta regla de despacho parte de la secuencia obtenida al aplicar SPT al problema. Una vez tenemos la secuencia iremos intercalando el primer trabajo de esta, seguido del último, estos serán los trabajos con menor y mayor tiempo de proceso respectivamente que aún quedan sin asignar, así hasta recorrer toda la secuencia. Siguiendo estos pasos la secuencia que obtenemos para nuestro ejemplo es: S_{SPT_INT} = [6,4,1,0,3,5,2].
- LPT_INT: esta regla de despacho sigue el mismo procedimiento que la descrita anteriormente, pero esta vez la secuencia de partida es lo obtenida mediante LPT. S_{SPT_INT} = [0,6,4,1,2,3,5].

Para estas dos últimas reglas de despacho en caso de empate se inserta el trabajo que primero se encuentre al recorrer la secuencia correspondiente, S_{SPT} o S_{LPT}, ya sea en sentido ascendente o descendente.

4.3 Metaheurística Black Widow Optimization (BWO)

Esta metaheurística para el problema de *Distributed Flowshop* fue propuesta por (Fu et al., 2022) y es una adaptación del algoritmo planteado por(Hayyolalam & Pourhaji Kazem, 2020). El BWO consta de tres fases: procreación, canibalismo y mutación. En primer lugar, la fase de procreación se encarga de buscar globalmente en el espacio de soluciones, mientras que la fase de canibalismo se enfoca en realizar búsquedas locales en regiones prometedoras identificadas en la fase anterior, por último, la fase de mutación se dedica a incrementar la diversidad de la población.

Antes de comenzar con la descripción del algoritmo conviene describir algunos parámetros que son esenciales para obtener los resultados al ejecutar el algoritmo. Mientras mejor se ajuste la cantidad de parámetros mayor será la probabilidad de escapar de óptimos locales y explorar el espacio de búsqueda de forma global. En este algoritmo se proponen tres:

- PP (*procreation rate*): este parámetro determina cuántos individuos participarán en la fase de procreación.
- CR (*cannibalism rate*): el ratio de canibalismo ayudará a descartar aquellos peores individuos de la población.
- PM (*mutation rate*): el ratio de mutación tiene como objetivo aumentar la diversidad de la población, este servirá para escoger individuos de la población inicial.

4.3.1 Inicialización de la población

Para mantener la diversidad de la población se generan aleatoriamente una serie de individuos, al tamaño de la población lo llamaremos *PS.*

4.3.2 Fase de procreación

En la fase de procreación usamos el parámetro PP para determinar cuántos individuos participarán en esta fase.

La fase de procreación se puede separar a su vez en dos, una donde se seleccionan pares de padres y otra donde a partir de estos padres, se crearán los hijos:

- Elección de padres:
    1) En primer lugar, se seleccionan aleatoriamente *nr* individuos de la población inicial (*nr* = PS·PP)
    2) Una vez se tengan estos individuos, elegimos aleatoriamente cuatro, de estos cuatro individuos el que tenga menor valor de la función objetivo será elegido como padre 1.
    3) Repetimos el paso 2 para obtener el padre 2.
    4) Si la pareja de padres obtenida es igual, repetimos el paso 3. En caso contrario vamos al paso 5.
    5) Se comparan los valores de las funciones objetivos de ambos padres, el que tenga menor valor de la función objetivo será elegido hembra y el otro como macho.
    6) Repetir los pasos del 1 al 5 hasta obtener *nr* pares de padres.

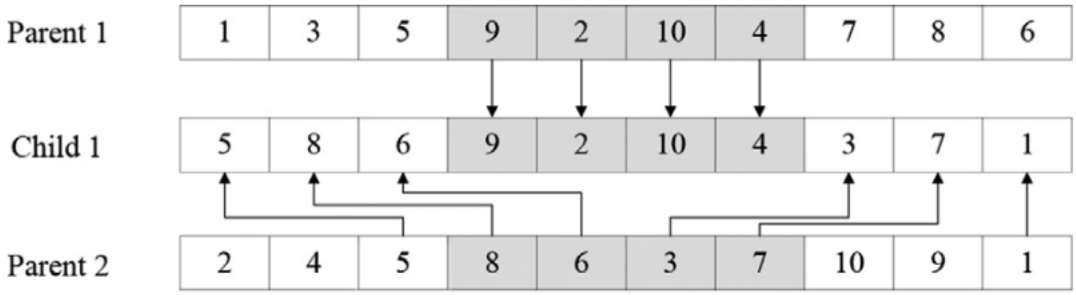

Ilustración 6. Creación de hijos (Fu et al., 2022)

- Creación de hijos:
    1) En primer lugar, se seleccionan dos puntos en cada padre para dividirlo.
    2) Tras esto la parte gris del padre 1, que se observa en *Ilustración 6*, se inserta directamente en el hijo 1.
    3) Ahora se recorre el padre 2 y aquellos elementos que no se encuentren aún en el hijo 1, se insertan en este, en el orden en el que aparecen en el padre 2.
    4) Repetir el mismo proceso para crear el hijo 2, intercambiando los roles del padre 1 y el padre 2.
    5) Ejecutar los pasos del 1 al 4 un total de cuatro veces por cada par de padres. Repitiendo esto obtenemos $8 \cdot nr$ nuevos individuos.

4.3.3 Fase de canibalismo

Para esta parte del algoritmo se elige de entre los tres tipos posibles de canibalismo, que son, el canibalismo sexual, el canibalismo entre hermanos, y el canibalismo especial, el canibalismo entre hermanos. En este, de los hijos que se generaron en la fase anterior solamente sobrevivirán un número determinado, $nc = (8 \cdot nr) \cdot (1-CR)$.

4.3.4 Fase de mutación

En el BWO las fases de procreación y canibalismo tienen como objetivo encontrar zonas prometedoras en el espacio de soluciones, mientras que la fase de mutación, donde se aplicará la técnica del recocido simulado, SA, se centra en encontrar mejores soluciones (Fu et al., 2022). En esta fase se explorarán un total de *nm* individuos, $nm = PS \cdot PM$.

El recocido simulado es una técnica relativamente reciente introducida por (Kirkpatrick et al., 1983), esta imita el proceso de recocido del acero que consiste en calentar hasta una temperatura T, y luego enfriar lentamente. Se puede hacer una analogía donde los movimientos de los átomos en este caso se refieren al movimiento de las soluciones. Inicialmente los átomos (soluciones), tienen una gran energía lo que permite grandes desplazamientos desde la zona inicial y a medida que la temperatura va decayendo estos saltos o movimientos son más limitados.

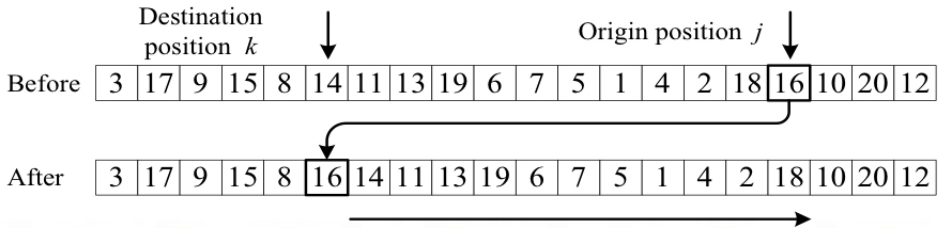

Ilustración 7. Creación de vecindades por el método de inserción (Framinan et al., 2014)

La metaheurística del recocido simulado se describe a continuación:

- En primer lugar, se establece una temperatura final, T_f=10^{-9}, y una temperatura inicial, $T = \frac{-(O_w - O_b)}{Ln(0.1)}$, donde O_w se corresponde con el peor valor de la función objetivo de la población de *nm* individuos que se está explorando, mientras que O_b se corresponde con el mejor valor encontrado en la población.
- Para cada individuo de la población (π), mientras no se cumplan las condiciones de paradas, generamos vecinos (π') por el método de inserción (*Ilustración 7*).
- Si el valor de la función objetivo del nuevo individuo es mejor, $obj' < obj*$, este se almacena y se toma como nuevo valor a comparar para las siguientes iteraciones, si es peor, se puede aceptar con una cierta probabilidad, criterio de aceptación de Metropolis (Metropolis et al., 1953). Se genera un número aleatorio *r* entre 0 y 1. Si $r \leq e^{-(obj' - obj^*)/T}$, el vecino se acepta.
- Por último, se actualiza el valor de T, T = 0,99·T.

A través de la experimentación se observa que, si la temperatura inicial es establecida a través de la fórmula antes mencionada, este valor es bajo, lo que puede provocar que no se exploren regiones del espacio que pueden ser prometedoras. Ante esta situación se propone una temperatura de partida fija de 50º. Además de lo mencionado se propone también una modificación del algoritmo donde se cree una población nueva en cada iteración que se haga. El análisis de los resultados obtenidos se llevará a cabo en la sección 5.3.2.

```
Input instance data, PS, PP, CR, PM
Output π_b, obj_b
Begin
    while Stopping criterion is not satisfied do
        IND:=PS initial sequences;
        pares_padres:=nr pairs of parents selected from IND;
        hijos:=8·nr offspring obtained from pares_padres;
        supervivientes:=nc best individuals selected from hijos;
        pob:=nm individuals selected from IND;
        for j = 1 to pob_size do
            Calculate initial temperature T:=-(Ow-Ob)/np.log(0.1);
            Set final temperature Tf:=1e-9 ;
            Calculate initial solution π_1:=pob_j;
            Set best known solution π*:= π_1;
            Calculate initial objective function value obj_1:=Obj(π_1);
            Calculate best objective function value so far obj*:=Obj(π*);
            Set current solution π_c:= π*;
            while Stopping criterion is not satisfied or T>Tf do
                π: = π_c;
                foreach neighbour π' of π in pob do
                    Calculate objective function value of neighbour obj':=Obj(π');
                    if obj'< obj* then
                        π* = π';
                        π_c:= π';
                        obj* = obj';
                    else
                        Accept worse solution probabilistically
                        if random ≤ e^(-(obj'-obj*)/T)
                            π_c:= π';
                pob_j:= π*;
                T:= r*T;
        population:=supervivientes + pob;
        π_b:= best individual from population;
        Obj_b:=Obj(π_b);
    return best solution π_b, best objective function value obj_b
```

Ilustración 8. Pseudocódigo Black Widow Optimization

4.4 Metaheurística Iterated Greedy con localsearch (IG)

Esta variante del *Iterated Greedy* fue propuesta por (Dubois-Lacoste et al., 2017) y a diferencia de otros algoritmos de *Iterated Greedy*, este incluye una búsqueda local, como el que se puede ver en *Ilustración 9*, tanto en la fase de destrucción como en la de construcción. La búsqueda local en las soluciones parciales resulta de interés puesto que se exploran regiones del espacio de soluciones distintas al de la solución completa, el tiempo computacional es menor, puesto que el tamaño de las soluciones es menor que cuando se trabaja con soluciones completas y además se obtiene una reestructuración de las soluciones que puede ser difícil de obtener de otro modo (Dubois-Lacoste et al., 2017).

Input: instance data, Π
Output: Π_b, obj_b, *improving*
begin
  $obj := Obj(\Pi)$;
  $\Pi_b = \Pi$;
  $obj_b = obj$;
  improving:=**False**;
  foreach *neighbour* Π' *of* Π **do**
    $obj' := Obj(\Pi')$;
    if $obj' \triangleleft obj$ **then**
      $\Pi_b = \Pi'$;
      $obj_b = obj'$;
      improving:=**True**;
  return Π_b, obj_b, *improving*

Ilustración 9. Método de búsqueda local First Improvement (Perez-Gonzalez et al., 2022)

Previo a la descripción del algoritmo, se definirán dos parámetros, T y δ, y que son claves para el buen funcionamiento de la metaheurística:

- $T = Tp \cdot \frac{\sum_{i=1}^{m}\sum_{j=1}^{n} pij}{n \cdot m \cdot 10}$, este valor establece la temperatura como una décima parte del tiempo medio de procesamiento de todas las operaciones (Osman & Potts, 1989). Para el valor Tp, se usará el mejor valor encontrado por (Dubois-Lacoste et al., 2017) es decir, Tp = 0.8, además del valor clásico que suele tomar este parámetro que es Tp=0.4 (Ruiz & Stützle, 2007).
- El parámetro δ indica el número de trabajos que serán borrados de la secuencia de partida y tomará un valor de $\delta = 2$ (Dubois-Lacoste et al., 2017) y δ = 4 (Ruiz & Stützle, 2007). Además de estos valores, también se propone que el número de trabajos a borrar sea un número aleatorio en cada iteración.

Esta metaheurística, a parte, de los dos parámetros previamente mencionados, también toma como dato de entrada una secuencia inicial, esta será obtenida a través de la heurística constructiva NEH (Nawaz et al., 1983), usada para problemas del tipo: $F_m|prmu|C_{max}$. En primer lugar, se ordenan los trabajos de mayor a menor tiempo de proceso, siguiendo la regla de despacho LPT. El primer elemento de esta ordenación se selecciona para crear una secuencia parcial de longitud uno, tras esto se insertan el resto de los trabajos en aquella posición donde el valor de la función objetivo de la secuencia parcial sea menor.

Input instance data
Output π
begin

$P_j = \sum_{i=1}^{m} p_{ij}$;
$\pi = \emptyset$, $J = \{1,\ldots,n\}$ verifying $P_1 \geq P_2 \geq \ldots P_n$;
$\pi[1]$=J[1];
for i=2 to n **do**
    π:= Insert job i in the position of π yielding the lowest objective function value;
return π

Ilustración 10. Pseudocódigo de la heurística constructiva NEH (Perez-Gonzalez et al., 2022)

Una vez se obtiene la secuencia inicial empieza la fase de destrucción, se borra de esta un número de trabajos, δ, obteniendo así dos secuencias, una secuencia, π^D, de longitud n-δ, donde han quedado los trabajos que no han sido borrados y otra, π^R, formado por los trabajos borrados. Es primero a esta secuencia π^R, a la que se le aplicará un mecanismo de búsqueda local como el de la *Ilustración 9*. Tras esto, da comienzo la fase de construcción, una vez se obtiene la nueva secuencia π^R, esta se recorre insertando cada uno de sus elementos en la posición de π^D que genere menor valor de la función objetivo. Cuando hemos recorrido toda la secuencia π^R, la longitud de π^D es n, es en este momento cuando se vuelve a aplicar un mecanismo de búsqueda local a la solución completa. Si esta nueva secuencia mejora en valor a la función objetivo de la secuencia de partida obtenida por NEH, este se almacenará y se usará en las siguientes iteraciones como valor a comparar. Si por el contrario no se consigue una mejora, la secuencia puede ser aceptada con una cierta probabilidad, criterio de aceptación de Metropolis (Metropolis et al., 1953).

Input instance data, parameters (T,δ)
Output π_b, obj_b
begin
    $\pi := (\pi_1, \ldots, \pi_n)$ initial sequence
    $obj := Obj(\pi)$
    $\pi_b := \pi$
    $obj_b := obj$;
    while Stopping criterion is **not** satisfied **do**
        $\pi' := \pi$
        $\pi_R := \emptyset$, $\pi_R = \pi'$;
        for $k = 1$ *to* δ **do**
            π_j:=Randomly selected job from π';
            π^D:=Remove job π_j;
            $\pi_R := \pi_R \cup \pi_j$;
        π_R=:LocalSearch(π_R);
        for $k = 1$ *to* δ **do**
            $\pi^{'}$:=Insert π_k^R in the best position of π^D;
            $\pi^D := \pi^{'}$;
        π^D:= LocalSearch(π^D);
        $obj' := Obj(\pi')$;
        if $obj' < obj$ **then**
            $\pi := \pi'$;
            obj = obj′;
            if $obj' < obj_b$ **then**
                $\pi_b := \pi$;
                $obj_b := obj$;
        else
            Accept worse solution probabilistically
            if random $\leq$ $e^{\frac{-(obj' - obj^*)}{T}}$
                $\pi := \pi'$;
    return π_b, obj_b

Ilustración 11. Pseudocódigo Iterated Greedy (Adaptado de (Dubois-Lacoste et al., 2017))

4.5 Metaheurística Algortimo Genético (GA)

El algoritmo genético al igual que el BWO descrito anteriormente, está inspirado en la naturaleza, y tratan de imitar el comportamiento de las especies a través de la evolución genética, donde las mejores sobreviven y las peores se extinguen.

Este algoritmo parte de una población inicial, gran parte de esta se genera aleatoriamente para aumentar la diversidad, pero habrá tres de ellas que se obtengan por el método de búsqueda local *first improvement*, generando vecinos, a partir de la secuencia obtenida por la heurística NEH (Nawaz et al., 1983), por los métodos de *adjacent swap*, *general swap* e *insertion*. Este último ya ha sido comentado en el apartado *4.3.4 Fase de mutación*, mientras que para los dos primeros se muestra en la *Ilustración 12* el procedimiento a seguir.

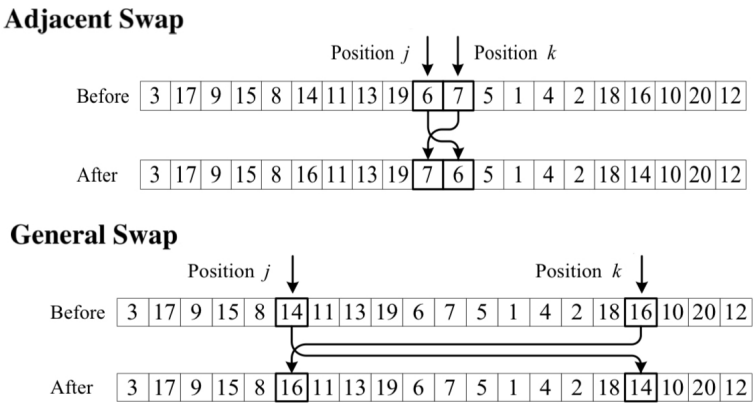

Ilustración 12. Creación de vecindades por los métodos adjacent y general swap (Perez-Gonzalez et al., 2022)

Una vez la población ha sido creada, es el momento de elegir dos padres de forma aleatoria. A partir de estos dos padres se generará un primer hijo del mismo modo del que se generaron en el algoritmo BWO (*Ilustración 6*). Partiendo de este primer hijo se crea un segundo por intercambio de pares, se seleccionan aleatoriamente dos posiciones y se intercambian los trabajos que las ocupan. Una vez se tiene el segundo hijo se evalúa el valor de la función objetivo y si es mejor que el de la secuencia de partida se hace una búsqueda local del individuo y este se guarda para compararlo en las siguientes iteraciones.

```
Input instance data, P_size
Output Population, π_b, fitness_b
begin
    Population:= P_size initial sequences;
    π_b:=Population_1;
    fitness_b:=Fitness of π_b;
    for j=1 to P_size do
        fitness_j:=Fitness of the sequence Population_j;
        if fitness_j<fitness_b then
            π_b:=Population_j
            fitness_b:=fitness_j
    while Stopping criterion is not satisfied do
        Parent_1:= sequence selected from Population;
        Parent_2:= sequence selected from Population;
        Child_1:= Cross Parent_1 and Parent_2;
        Child':= Mutate Child_1;
        fitness:=Fitness of Child';
        if fitness<fitness_b then
            π_b:= LocalSearch(Child');
            fitness_b:=obj(Child');
    return Population, π_b, fitness_b
```

Ilustración 13. Pseudocódigo GA con local search (Adaptado de (Perez-Gonzalez et al., 2022))

5 RESULTADOS OBTENIDOS

En este apartado se detalla los programas utilizados para la interpretación de resultados, la resolución de los algoritmos, las instancias empleadas en el problema y los indicadores usados para la interpretación del problema.

5.1 Herramientas utilizadas

Para codificar los algoritmos propuestos se hace uso de *Python*, este se trata de un lenguaje de programación de alto nivel diseñado para ser comprensible por los humanos, e interpretado, es decir, el código se ejecuta línea a línea en lugar de ser compilado antes de la ejecución (Sulbarán, 2023). *Python* cuenta con una amplia serie de librerías, entre ellas se encuentra la librería *scheptk* (*Librería SCHEPTK – Grupo de Investigación Organización Industrial*, n.d), que será usada durante el estudio.

Para la interpretación de los resultados se hará uso de *Microsoft Excel*, que permite realizar operaciones con facilidad y crear tablas y gráficos de forma rápida y sencilla.

5.2 Instancias e indicadores para la evaluación

Para la evaluación de los algoritmos se usa un total de 102 instancias de (Naderi & Ruiz, 2010). Estas instancias toman valores de $f \in [2,3,4,5,6,7]$, $n \in [10,12,14,16,20,50,100]$ y $m \in [2,3,4,5,10,20]$. Los tiempos de proceso de estas instancias comprenden valores entre 1 y 99, y han sido generados siguiendo una distribución uniforme (Taillard, 1993). Los pesos de los trabajos, *w*, y los costes de transporte, *wf*, han sido añadidos y toman valores entre 1 y 99, y 10 y 99 respectivamente. Cabe recordar también que en el problema que se plantea, los trabajos se encuentran disponibles en el instante cero y que no hay fechas de entregas, por lo que estas tomarán valores de cero en el momento de la creación de las instancias.

Para la evaluación de los resultados obtenidos el indicador elegido es el *average relative percentage deviation* (ARPD) que es el valor promedio de las desviaciones porcentuales relativas, conocida en sus siglas en inglés por RPD. El RPD muestra la variación de cada instancia con respecto al mejor valor obtenido para esa instancia. Valores cercanos a cero indican que el valor de la función objetivo para esa instancia está próximo al mejor valor de la función objetivo obtenido al evaluar dicha instancia, mientras que valores cercanos a 100 indican que estamos lejos de la mejor solución obtenida. El RPD y el ARPD se describen como siguen:

$$\mathrm{RPD} = \frac{\mathrm{valorFO} - \mathrm{ValorMinFO}^*}{\mathrm{ValorMinFO}^*} \cdot 100 \qquad \mathrm{ARPD} = \frac{1}{\mathrm{k}} \sum_{\mathrm{i}=1}^{\mathrm{k}} \mathrm{RPD_i}$$

5.3 Experimentación

5.3.1 Resultados regla de asignación

Como se comentó en la sección 4.1 Reglas de asignación, en primer lugar, se analiza que regla de asignación (RA) de las cuatro propuestas es mejor.

Valores ARPD por regla de despacho					
ECT	SPT	0,1575	**ECT_minWF**	SPT	0,1417
	LPT	0,3468		LPT	0,3567
	WSPT	0,1029		WSPT	0,0984
	SPT_INT	0,2451		SPT_INT	0,2228
	LPT_INT	0,2517		LPT_INT	0,2783
FAM	SPT	0,1491	**FAM_minWF**	SPT	0,1364
	LPT	0,3348		LPT	0,3371
	WSPT	0,0975		WSPT	0,0848
	SPT_INT	0,1055		SPT_INT	0,2240
	LPT_INT	0,1928		LPT_INT	0,2758

Tabla 2. Resultados ARPD por regla de despacho

Ilustración 14. ARPD por regla de despacho

Observando este gráfico se puede ver que el mejor valor del ARPD se obtiene con la cuarta regla de asignación *(FAM_minWF)* y la regla de despacho *weighted shortest processing time* (WSPT). Esta primera conclusión puede resultar imprecisa puesto que no se tiene en cuenta el resto de los valores obtenidos para esa mima regla de asignación, pudiendo ignorar así comportamientos menos eficaces en la obtención de resultados con otras reglas de despacho. Es por ello por lo que se calcula a continuación la media de los valores de ARPD obtenidos por las reglas de asignación.

Valores ARPD por regla de asignación	
ECT	0,2208
FAM	**0,1759**
ECT_minWF	0,2196
FAM_minWF	0,2116

Tabla 3. Valores ARPD por regla de asignación

Ilustración 15. Valores ARPD por regla de asignación

Como se puede ver en la *Tabla 3*, el mejor valor de la función objetivo se obtiene con la regla de asignación *first avaliable machine* (FAM), y desde ahora en adelante será la regla de asignación que se usará a la hora de ejecutar con las metaheurísticas propuestas.

5.3.2 Resultados de las metaheurísticas

Antes de empezar a comparar los tres algoritmos propuestos, se procederá a analizar los resultados arrojados por la metaheurística BWO en dos situaciones distintas, en la primera se generará una población que se mantendrá fija a lo largo del tiempo de ejecución del algoritmo, mientras que en la segunda se generará una población distinta en cada nueva iteración. En cada caso el algoritmo es ejecutado dos veces, la primera vez se usa la fórmula propuesta por (Fu et al., 2022), $T = \frac{-(O_w - O_b)}{Ln(0.1)}$, y la segunda se usa una temperatura fija T=50º.

Valores ARPD para BWO	Población fija		Población en cada iteración	
	$T = \frac{-(O_w - O_b)}{Ln(0.1)}$	T=50º	$T = \frac{-(O_w - O_b)}{Ln(0.1)}$	T=50º
	BWO	BWO1	BWO2	BWO3
ARPD	0,0274	0,0218	0,0257	**0,0108**
Media ARPD	0,0246		**0,0182**	

Tabla 4. Valores ARPD para BWO

Como se observa, los mejores valores para el algoritmo de viuda negra se obtienen cuando se genera una población nueva en cada iteración y cuando la temperatura inicial es de 50º.

A continuación, en la *Tabla 5* se muestran los resultados del algoritmo *iterated greedy* en dos casos distintos, el primero de ellos cuando *Tp* toma el valor de *Tp*=0.8 y el segundo cuando *Tp*=0.4. En cada caso, a su vez se diferencian dos situaciones, una cuando el número de trabajos a borrar es *d*=2 y otro cuando este es aleatorio. Para el caso de *Tp*=0.4, como ya se comentó anteriormente, también se evaluará el comportamiento del algoritmo cuando el número de trabajos a borrar toma un valor aleatorio. Como se observa, para el caso de estudio de este documento, se obtienen mejores valores para Tp=0.4 y *d*=2.

Valores ARPD para IG	*Tp*=0.8		*Tp*=0.4		
	d=2	*d*=random	*d*=2	*d*=random	*d=4*
	IG	IG1	IG2	IG3	IG4
ARPD	0,0185	0,0297	**0,0163**	0,0291	0,0235
Media ARPD	0,0241		**0,0230**		

Tabla 5. Resultados ARPD para IG

A continuación, se muestran los resultados obtenidos por cada metaheurística.

Valores ARPD por metaheurística	
BWO3	0,0227
IG2	0,0151
GA	**0,0128**

Tabla 6. Resultados ARPD por metaheurística

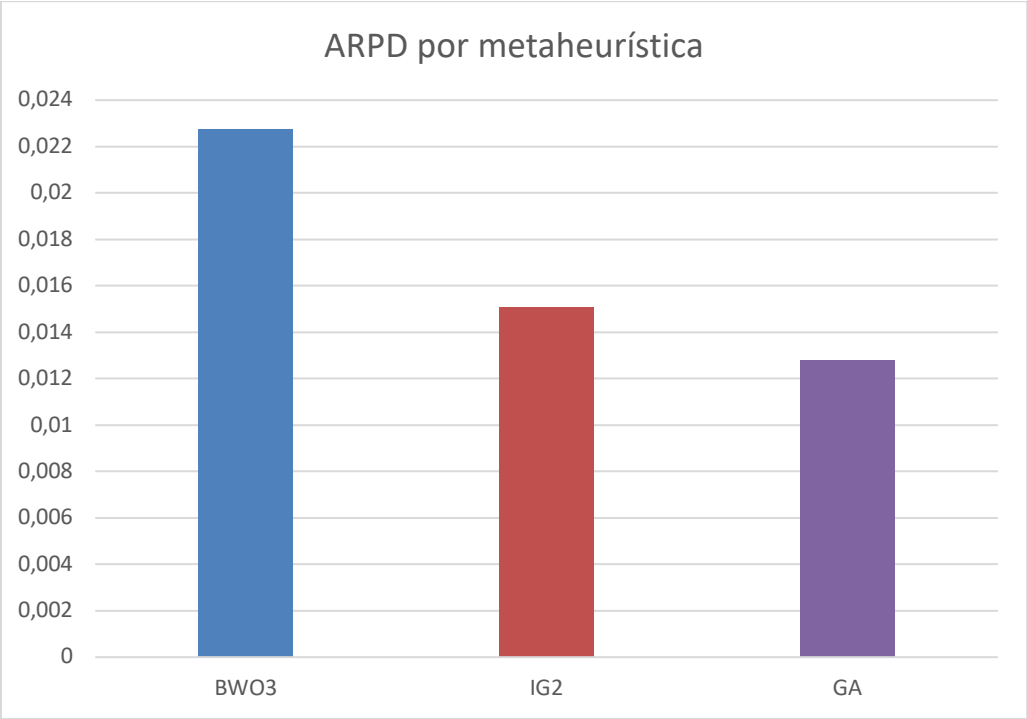

Ilustración 16. Resultados ARPD por metaheurística

En la *Tabla 6* se observa que el mejor valor promedio ARPD para las instancias evaluadas es la metaheurística algoritmo genético (GA).

Analizando más detalladamente los resultados obtenidos, en la Tabla 7 se analizan los valores promedios del RPD en función del número de fábricas. Como se puede ver, cuando $f = 2$ la metaheurística *IG* es la que obtiene mejores resultados, mientras que cuando el número de fábricas es $f = 3,4,5,6,7$ es el algoritmo genético quien proporciona mejores resultados. Observando los valores medios se observa que es de nuevo el algoritmo genético quien arroja mejores resultados, mejorando significativamente con respecto a los otros dos algoritmos que se proponen cuando va aumentando el número de fábricas.

En la *Tabla 8* se observa que cuando el número de trabajos es, $j = 10,12,16$ el algoritmo de *IG* es el mejor de los estudiados, mientras que cuando $j = 14,20,50,100$ el algoritmo genético obtiene resultados mejores. Si se estudian los valores medios obtenidos tras la experimentación, se puede ver que el *Iterated Greedy* es la metaheurística que da mejores resultados.

Por último, analizando ahora los resultados en función del número de máquinas, en la *Tabla 9* se puede ver que cuando $m = 2$, *BWO* aporta los mejores resultados, cuando $m = 3,4$ es *IG* quien los arroja, y cuando el número de máquinas es $m = 5,10,20$ los mejores resultados se obtienen con *GA*. De nuevo, viendo los valores medios, es la metaheurística *IG* quien proporciona mejor resultados, aunque se observa que es el algoritmo genético quien tiene un mejor comportamiento conforme se incrementa el número de máquinas.

VALORES ARPD POR FÁBRICAS			
FÁBRICAS	BWO3	IG2	GA
2	0,0212	**0,0116**	0,0216
3	0,0226	0,0165	**0,0112**
4	0,0291	0,0178	**0,0152**
5	0,0181	0,0118	**0,0075**
6	0,0148	0,0144	**0,0026**
7	0,0222	0,0169	**0,001**
Media ARPD	0,0213	0,0148	**0,0099**

Tabla 7. Resultados ARPD por fábricas

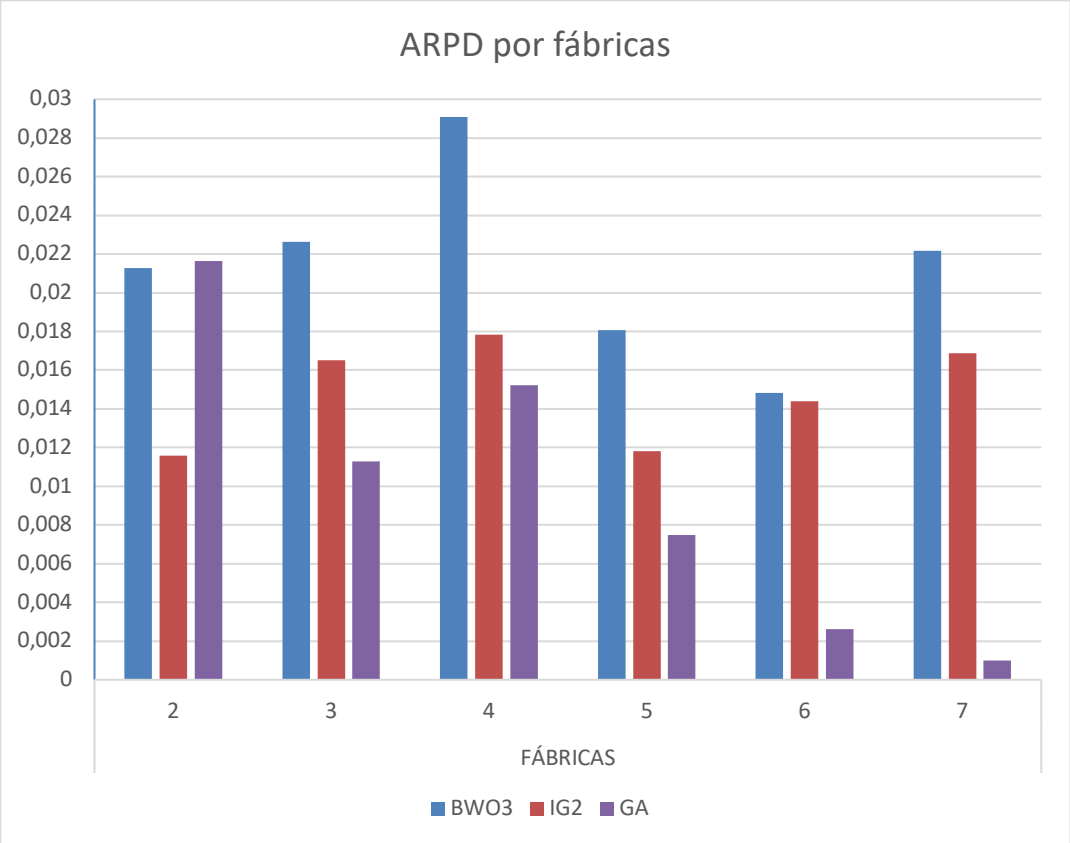

Ilustración 17. Representación del ARPD por fábricas

VALORES ARPD POR TRABAJOS			
TRABAJOS	BWO3	IG2	GA
10	0,0263	**0,0044**	0,0223
12	0,03	**0,0105**	0,0137
14	0,035	0,0113	**0,0087**
16	0,0188	**0,0175**	0,0449
20	0,0158	0,0216	**0,0099**
50	0,0225	0,0202	**0,0007**
100	0,0173	0,0143	**0,0016**
Media ARPD	0,0236	**0,0143**	0,0146

Tabla 8. Resultados ARPD por trabajos

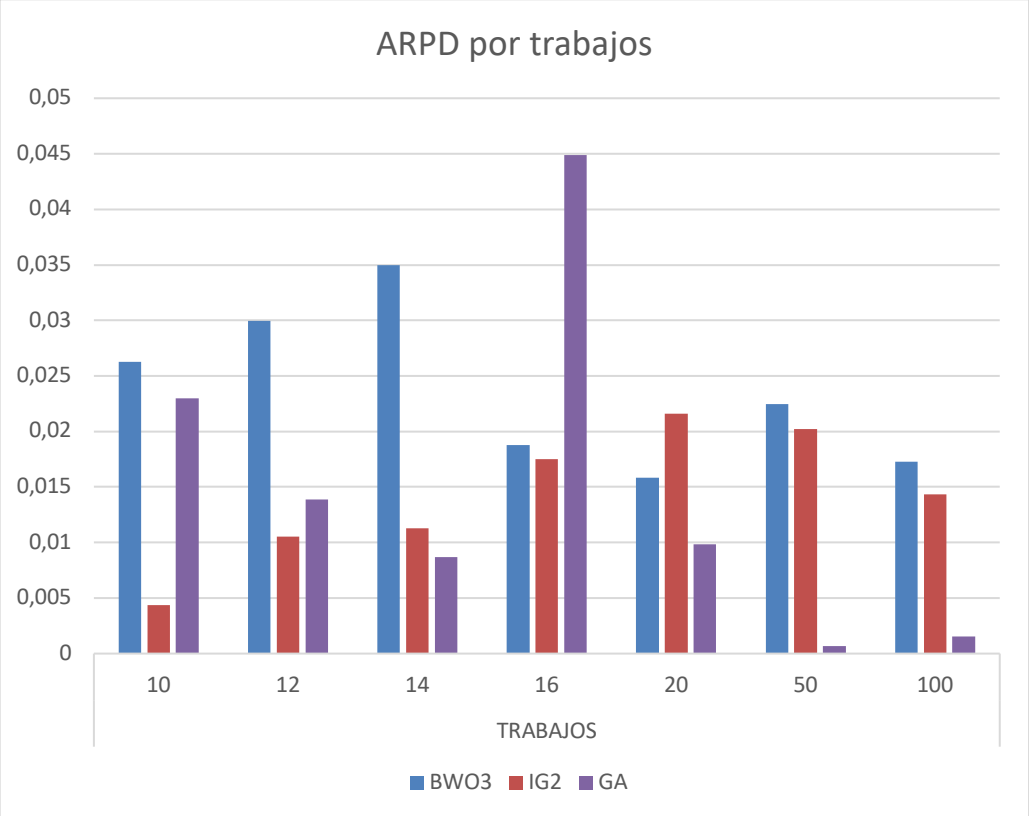

Ilustración 18. Representación del ARPD por trabajos

VALORES ARPD POR MÁQUINAS			
MÁQUINAS	BWO3	IG2	GA
2	**0,0156**	0,0162	0,0213
3	0,0372	**0,0073**	0,0277
4	0,0294	**0,006**	0,0174
5	0,0256	0,0177	**0,0127**
10	0,0146	0,019	**0,0032**
20	0,0167	0,0172	**0,0038**
Media ARPD	0,0232	**0,0139**	0,0143

Tabla 9. Resultados ARPD por máquinas

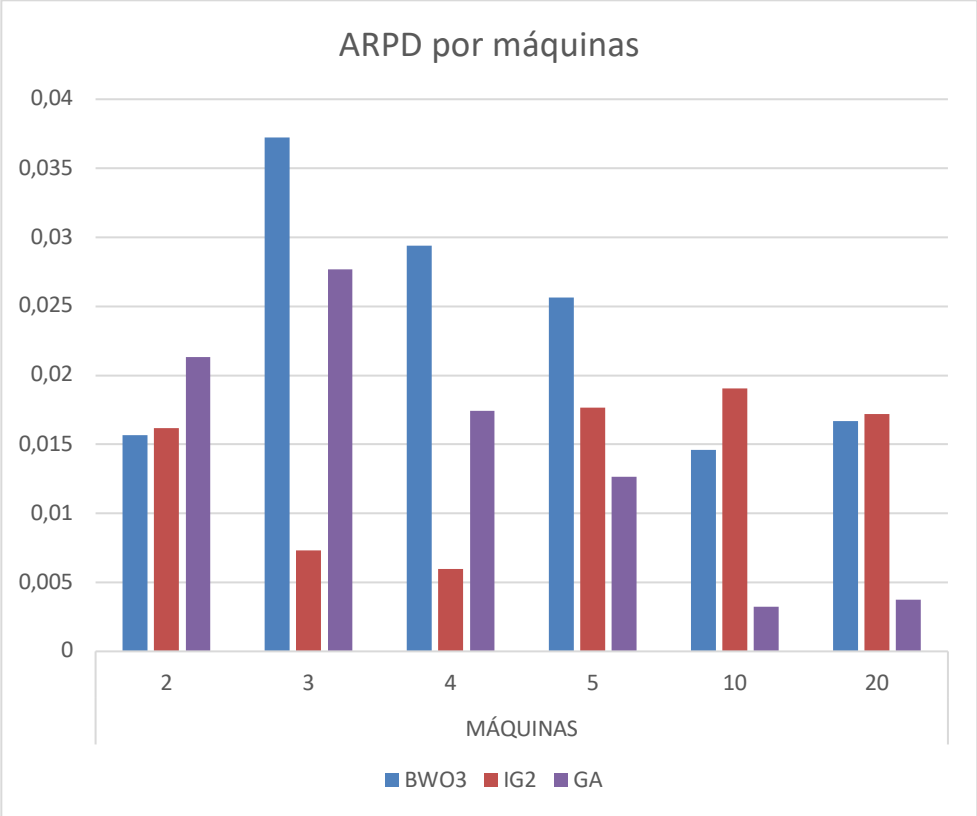

Ilustración 19. Representación del ARPD por máquinas

6 CONCLUSIONES

En este último capítulo se presentan las conclusiones alcanzadas tras la aplicación de diferente metaheurísticas en un entorno Distributed Flowshop con restricción de permutación y donde el objetivo es minimizar el *total weighted completion times* y el coste de transporte. Para el estudio se han seguido los siguientes pasos:

- En primer lugar, en los capitulos 1 y 2, se describe el objetivo del estudio y se hace una introducción a la programación de operaciones.
- Posteriormente en la sección 3, se presenta la descripción del problema, donde se describe el entorno estudiado, así como las restricciones y la función objetivo. Después en la sección 4, se hace una descripción de las reglas de asignación usadas, de las reglas de despacho empleadas para analizar dichas reglas de asignación y de las metaheurísticas usadas para analizar el problema.
- Finalmente en el capítulo 5 , se analizan los resultados obtenidos tras ejecutar el código.

Una vez se han estudiado los resultados, se puede afirmar que para el caso de estudio la mejor regla de asiganción es *first avaliable machine* (FAM). Además, se observa que en la metaheurística *black widow optimization* (BWO) se obtienen mejores resultados cuando en cada iteración se genera una nueva población de individuos y cuando la temperatura de partida es de 50º, y que en el algorirmo *itereted greedy* (IG) se obtiene un mejor compartamiento cuando la temperatura Tp es Tp=0.4 y cuando es número de trabajos a borrar es cuatro. Por último de forma general al evaluar todas las instancias en conjunto, es el algortirmo genético (GA) quien proporciona mejores resultados, obteniendo el menor valor medio del RPD.

BIBLIOGRAFÍA

Dubois-Lacoste, J., Pagnozzi, F., & Stützle, T. (2017). An iterated greedy algorithm with optimization of partial solutions for the makespan permutation flowshop problem. *Computers and Operations Research*, *81*. https://doi.org/10.1016/j.cor.2016.12.021

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*. https://doi.org/10.1016/j.cie.2018.03.014

Framinan, J. M., Leisten, R., & García, R. R. (2014). Manufacturing scheduling systems: An integrated view on models, methods and tools. In *Manufacturing Scheduling Systems: An Integrated View on Models, Methods and Tools* (Vol. 9781447162728). https://doi.org/10.1007/978-1-4471-6272-8

Fu, Y., Hou, Y., Chen, Z., Pu, X., Gao, K., & Sadollah, A. (2022). Modelling and scheduling integration of distributed production and distribution problems via black widow optimization. *Swarm and Evolutionary Computation*, *68*. https://doi.org/10.1016/j.swevo.2021.101015

Graham, R. L., Lawler, E. L., Lenstra, J. K., & Kan, A. H. G. R. (1979). Optimization and heuristic in deterministic sequencing and scheduling: a survey. *Annals of Discrete Mathematics*, *5*.

Hayyolalam, V., & Pourhaji Kazem, A. A. (2020). Black Widow Optimization Algorithm: A novel meta-heuristic approach for solving engineering optimization problems. *Engineering Applications of Artificial Intelligence*, *87*. https://doi.org/10.1016/j.engappai.2019.103249

Kirkpatrick, S., Gelatt, C. D., & Vecchi, M. P. (1983). Optimization by simulated annealing. *Science*, *220*(4598). https://doi.org/10.1126/science.220.4598.671

Metropolis, N., Rosenbluth, A. W., Rosenbluth, M. N., Teller, A. H., & Teller, E. (1953). Equation of state calculations by fast computing machines. *The Journal of Chemical Physics*, *21*(6). https://doi.org/10.1063/1.1699114

Naderi, B., & Ruiz, R. (2010). The distributed permutation flowshop scheduling problem. *Computers and Operations Research*, *37*(4). 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. *Omega*, *11*(1). https://doi.org/10.1016/0305-0483(83)90088-9

Osman, I., & Potts, C. (1989). Simulated annealing for permutation flow-shop scheduling. *Omega*, *17*(6). https://doi.org/10.1016/0305-0483(89)90059-5

Perez-Gonzalez, P., Framinan, J. M., & Navarro-Garcia, B. (2022). Programación de Operaciones. 4o GITI. Universidad de Sevilla

Pinedo, M. L. (2012). Scheduling: Theory, algorithms, and systems: Fourth edition. In *Scheduling: Theory, Algorithms, and Systems: Fourth Edition* (Vol. 9781461423614). https://doi.org/10.1007/978-1-4614-2361-4

Ruiz, R., & Stützle, T. (2007). A simple and effective iterated greedy algorithm for the permutation flowshop scheduling problem. *European Journal of Operational Research*, *177*(3). https://doi.org/10.1016/j.ejor.2005.12.009

Sulbarán, I. (25 de septiembre de 2023). Python en programación: Qué es, para qué sirve y por qué debes aprender a usarlo. https://global.tiffin.edu/noticias/que-es-python-para-que-sirve-y-por-que-aprender-a- usarlo

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

Toptal, A., & Sabuncuoglu, I. (2010). Distributed scheduling: A review of concepts and applications. *International Journal of Production Research*, *48*(18). https://doi.org/10.1080/00207540903121065

ANEXO: CÓDIDO DE PYTHON

ECT

```
from scheptk.scheptk import *
import sys # to stop the execution (funcion exit() )
from scheptk.util import *
import copy,random

class DPFS_ECT(Model):
    def __init__(self, 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(self.machines ==-1):
            print("No machines specified. The program cannot continue.")
            sys.exit()
        else:
            print_tag("MACHINES", self.machines)

        #leer el vector f
        self.f = read_tag(filename,"F")
        if(self.f == -1):
            print("No factories specified. The program cannot continue.")
            sys.exit()
        else:
            print_tag("F",self.f)
```

```
        #leer el vector de pesos de la fábrica
        self.wf = read_tag(filename,"WF")
        if(self.wf == -1):
            print("No weights for factories specified. The program cannot continue.")
            sys.exit()
        else:
            if(len(self.wf) != self.f):
                print("Number of factories does not match the number of weight of the
factories (FACTORIES={}, lenght of WF={}). The program cannot continue".format(self.f,
len(self.wf)))
                sys.exit()
            else:
                print_tag("WF",self.wf)


        # 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)




        # weights (if not, default weights)
        self.check_weights(filename)


        # due dates (if not,  -1 is assumed)
        self.check_duedates(filename)
```

```
        # release dates (if not, 0 is assumed)
        self.check_releasedates(filename)


    def ct(self,sequence):
        ct_fabricas = [0 for i in range(self.f)]
        # print("Inicializamos el vector de ct de las fabricas a cero",ct_fabricas)
        ct = [[0 for j in range(len(sequence))] for i in range(self.machines)]
        # print("Vector de ct de cada trabajo en cada máquina", ct)
        ct_jobs = [[0 for j in range(len(sequence))] for i in range(self.f)]
        # print("Vector de cada trabajo en cada fábrica", ct_jobs)
        vector_aux = [[0 for j in range(len(sequence))] for i in range(self.f) for k in
range(self.machines)] #vector auxiliar donde guardaremos los timepos de proceso de cada
trabajo en cada maquina y en cada fábricas
        # print("Creamos un vector auxiliar",vector_aux)
        #Creamos un vector donde guardar los tiempos de proceso de un trabajo pero SOLO
en su fábrica asignada
        vector_ct = [[0 for j in range(len(sequence))] for i in range(self.f) for k in
range(self.machines)]
        aux_ctjobs = [0 for j in range(self.f)] #vector para guardar lo que tarda cada
trabajo en cada fábrica



        for j in range(0,len(sequence)): #0,self.jobs
            for k in range(self.f):
                for i in range(self.machines):
                    if k == 0:
                        if i == 0:
                            ct[i][j] = max(max(vector_aux[i]), self.r[sequence[j]]) +
self.pt[i][sequence[j]]
                            vector_aux[i][j] = ct[i][j]
                        else:
                            ct[i][j] = max(max(vector_aux[i-1]), max(vector_aux[i])) +
self.pt[i][sequence[j]]
                            vector_aux[i][j] = ct[i][j]
                            aux_ctjobs[k] = vector_aux[i][j]
                    else:
                        if i == 0:
                            ct[i][j] = max(self.r[sequence[j]],
max(vector_aux[self.machines*k])) + self.pt[i][sequence[j]]
                            vector_aux[self.machines*k][j] = ct[i][j]
                        else:
                            ct[i][j] = max(max(vector_aux[self.machines*k+i]),
```

```
max(vector_aux[self.machines*k+i-1]))            + self.pt[i][sequence[j]]
                            vector_aux[self.machines*k+i][j] = ct[i][j]
                            aux_ctjobs[k] = vector_aux[self.machines*k+i][j] #copy


            # print("vector auxiliar ", vector_aux)
            # print("Vector del trabajo en todas las fábricas",aux_ctjobs)
            fabrica_asignada = find_index_min(aux_ctjobs)
            # print("El trabajo en la posición {} va a la fábrica {}"
.format(j,fabrica_asignada))
            for m in range(self.machines):
                vector_ct[self.machines*fabrica_asignada+m][j] =
vector_aux[self.machines*fabrica_asignada+m][j] #copy
            vector_aux = copy.deepcopy(vector_ct)
            # print("Vector auxiliar listo para nueva asignación", vector_aux)
            # print("Vector de ct de cada trabajo en cada máquina de la fábrica
asignada",vector_ct)
            ct_jobs[fabrica_asignada][j] = aux_ctjobs[fabrica_asignada] #copy
        # print(ct_jobs)
        return ct_jobs,sequence
```

FAM

```
from scheptk.scheptk import *
import sys # to stop the execution (funcion exit() )
from scheptk.util import *
import copy,random


class DPFS_FAM(Model):


    def __init__(self, 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(self.machines ==-1):
            print("No machines specified. The program cannot continue.")
            sys.exit()
        else:
            print_tag("MACHINES", self.machines)

        #leer el vector f
        self.f = read_tag(filename,"F")
        if(self.f == -1):
            print("No factories specified. The program cannot continue.")
            sys.exit()
        else:
            print_tag("F",self.f)

        #leer el vector de pesos de la fábrica
        self.wf = read_tag(filename,"WF")
        if(self.wf == -1):
            print("No weights for factories specified. The program cannot continue.")
            sys.exit()
        else:
            if(len(self.wf) != self.f):
                print("Number of factories does not match the number of weight of the
factories (FACTORIES={}, lenght of WF={}). The program cannot continue".format(self.f,
len(self.wf)))
                sys.exit()
            else:
                print_tag("WF",self.wf)

        # 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)


        # weights (if not, default weights)
        self.check_weights(filename)


        # due dates (if not,  -1 is assumed)
        self.check_duedates(filename)


        # release dates (if not, 0 is assumed)
        self.check_releasedates(filename)


    def ct(self,sequence):
        ct = [[0 for j in range(len(sequence))] for i in range(self.machines)]
        # print("Vector de ct de cada trabajo en cada máquina", ct)
        ct_jobs = [[0 for j in range(len(sequence))] for i in range(self.f)]
        # print("Vector de cada trabajo en cada fábrica", ct_jobs)
        vector_aux = [[0 for j in range(len(sequence))] for i in range(self.f) for k in
range(self.machines)]
        # print("Creamos un vector auxiliar",vector_aux)


        aux_ctjobs = [0 for j in range(self.f)] #vector para guardar lo que tarda cada
trabajo en cada fábrica


        for j in range(0,len(sequence)):
            # print("Vector del trabajo en todas las fábricas",aux_ctjobs)
            fabrica_asignada = find_index_min(aux_ctjobs)
            #     print("El     trabajo     {}     va     a     la     fábrica     {}"
.format(sequence[j],fabrica_asignada))
            for i in range(self.machines):
                    if fabrica_asignada == 0:
                        if i == 0:
                            ct[i][j]  =  max(max(vector_aux[i]),  self.r[sequence[j]])  +
self.pt[i][sequence[j]]
```

```
                            vector_aux[i][j] = ct[i][j]
                        else:
                            ct[i][j] = max(max(vector_aux[i-1]), max(vector_aux[i])) +
self.pt[i][sequence[j]]
                            vector_aux[i][j] = ct[i][j]
                            aux_ctjobs[fabrica_asignada] = vector_aux[i][j]
                    else:
                        if i == 0:
                            ct[i][j] = max(self.r[sequence[j]],
max(vector_aux[self.machines*fabrica_asignada])) +     self.pt[i][sequence[j]]
                            vector_aux[self.machines*fabrica_asignada][j] = ct[i][j]
                        else:
                            ct[i][j] =
max(max(vector_aux[self.machines*fabrica_asignada+i]),
max(vector_aux[self.machines*fabrica_asignada+i-1])) + self.pt[i][sequence[j]]
                            vector_aux[self.machines*fabrica_asignada+i][j] = ct[i][j]
                            aux_ctjobs[fabrica_asignada]                                =
copy.deepcopy(vector_aux[self.machines*fabrica_asignada+i][j])


            # print("vector auxiliar ", vector_aux)
            # print("Vector de los completion times finales del trabajo",aux_ctjobs)
            ct_jobs[fabrica_asignada][j] = copy.deepcopy(aux_ctjobs[fabrica_asignada])
        # print(ct_jobs)


        return ct_jobs,sequence
```

ECT_lowestWF

```
from scheptk.scheptk import *
import sys # to stop the execution (funcion exit() )
from scheptk.util import *
import copy,random


class DPFS_ECT_WF(Model):


   def __init__(self, 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(self.machines ==-1):
            print("No machines specified. The program cannot continue.")
            sys.exit()
        else:
            print_tag("MACHINES", self.machines)

        #leer el vector f
        self.f = read_tag(filename,"F")
        if(self.f == -1):
            print("No factories specified. The program cannot continue.")
            sys.exit()
        else:
            print_tag("F",self.f)

        #leer el vector de pesos de la fábrica
        self.wf = read_tag(filename,"WF")
        if(self.wf == -1):
            print("No weights for factories specified. The program cannot continue.")
            sys.exit()
        else:
            if(len(self.wf) != self.f):
                print("Number of factories does not match the number of weight of the
factories (FACTORIES={}, lenght of WF={}). The program cannot continue".format(self.f,
len(self.wf)))
                sys.exit()
            else:
                print_tag("WF",self.wf)

        # 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)




        # weights (if not, default weights)
        self.check_weights(filename)

        # due dates (if not,  -1 is assumed)
        self.check_duedates(filename)

        # release dates (if not, 0 is assumed)
        self.check_releasedates(filename)



   def ct(self,sequence):
       ct = [[0 for j in range(len(sequence))] for i in range(self.machines)]
       #print("Vector de ct de cada trabajo en cada máquina", ct)
       ct_jobs = [[0 for j in range(len(sequence))] for i in range(self.f)]
       #print("Vector de cada trabajo en cada fábrica", ct_jobs)
       vector_aux = [[0 for j in range(len(sequence))] for i in range(self.f) for k in range(self.machines)] #vector auxiliar donde guardaremos los timepos de proceso de cada trabajo en cada maquina y en cada fábricas
       #print("Creamos un vector auxiliar",vector_aux)
       #Creamos un vector donde guardar los tiempos de proceso de un trabajo pero SOLO en su fábrica asignada
       vector_ct = [[0 for j in range(len(sequence))] for i in range(self.f) for k in range(self.machines)]
       aux_ctjobs = [0 for j in range(self.f)] #vector para guardar lo que tarda cada trabajo en cada fábrica
```

```
        for j in range(len(sequence)):
            for k in range(self.f):
                for i in range(self.machines):
                    if k == 0:
                        if i == 0:
                            ct[i][j]  =  max(max(vector_aux[i]),  self.r[sequence[j]])  +
self.pt[i][sequence[j]]
                            vector_aux[i][j] = ct[i][j]
                        else:
                            ct[i][j]  =  max(max(vector_aux[i-1]),  max(vector_aux[i]))  +
self.pt[i][sequence[j]]
                            vector_aux[i][j] = ct[i][j]
                            aux_ctjobs[k] = vector_aux[i][j]


                    else:
                        if i == 0:
                            ct[i][j]              =              max(self.r[sequence[j]],
max(vector_aux[self.machines*k])) + self.pt[i][sequence[j]]
                            vector_aux[self.machines*k][j] = ct[i][j]
                        else:
                            ct[i][j]      =      max(max(vector_aux[self.machines*k+i]),
max(vector_aux[self.machines*k+i-1])) + self.pt[i][sequence[j]]
                            vector_aux[self.machines*k+i][j] = ct[i][j]
                            aux_ctjobs[k]                                                =
copy.deepcopy(vector_aux[self.machines*k+i][j])



            # print("vector auxiliar tras la asignación", vector_aux)
            # print("Vector ct del trabajo en todas las fábricas",aux_ctjobs)
            #asignamos ahora a que fábrica irá el trabajo en función del peso de la
fábrica
            minct = aux_ctjobs[find_index_min(aux_ctjobs)]
            # print("El mínimo completion time es", minct)
            index = []
            for l in range(len(aux_ctjobs)):
                if minct == aux_ctjobs[l]:
                    # print("Posible fábrica asiganada",l)
                    index.append(l)
            # print("Fábricas candidatas a insertar trabajos", index)
            minwf = self.wf[index[0]]
```

```
            fabrica_asignada = copy.deepcopy(index[0])
            for n in range(len(index)):
                if self.wf[index[n]] < minwf:
                    minwf = copy.deepcopy(self.wf[index[n]])
                    fabrica_asignada = copy.deepcopy(index[n])
            #  print("El  trabajo  {}  irá  a  la  fábrica  {}  cuyo  peso  es  {}"
.format(sequence[j],fabrica_asignada,minwf))


            for m in range(self.machines):
                vector_ct[self.machines*fabrica_asignada+m][j]                          =
copy.deepcopy(vector_aux[self.machines*fabrica_asignada+m][j])
            vector_aux = copy.deepcopy(vector_ct)
            # print("Vector auxiliar listo para nueva asignación", vector_aux)
            #  print("Vector  de  ct  de  cada  trabajo  en  cada  máquina  de  la  fábrica
asignada",vector_ct)
            ct_jobs[fabrica_asignada][j] = copy.deepcopy(aux_ctjobs[fabrica_asignada])
        # print(ct_jobs)


        return ct_jobs,sequence
```

FAM_lowestWF

```
from scheptk.scheptk import *
import sys # to stop the execution (funcion exit() )
from scheptk.util import *
import copy,random


class DPFS_FAM_WF(Model):
    def __init__(self, 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(self.machines ==-1):
            print("No machines specified. The program cannot continue.")
            sys.exit()
        else:
            print_tag("MACHINES", self.machines)

        #leer el vector f
        self.f = read_tag(filename,"F")
        if(self.f == -1):
            print("No factories specified. The program cannot continue.")
            sys.exit()
        else:
            print_tag("F",self.f)

        #leer el vector de pesos de la fábrica
        self.wf = read_tag(filename,"WF")
        if(self.wf == -1):
            print("No weights for factories specified. The program cannot continue.")
            sys.exit()
        else:
            if(len(self.wf) != self.f):
                print("Number of factories does not match the number of weight of the
factories (FACTORIES={}, lenght of WF={}). The program cannot continue".format(self.f,
len(self.wf)))
                sys.exit()
            else:
                print_tag("WF",self.wf)
        # 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)
        # weights (if not, default weights)
        self.check_weights(filename)

        # due dates (if not,  -1 is assumed)
        self.check_duedates(filename)

        # release dates (if not, 0 is assumed)
        self.check_releasedates(filename)

    def ct(self,sequence):
        ct = [[0 for j in range(len(sequence))] for i in range(self.machines)]
        #print("Vector de ct de cada trabajo en cada máquina", ct)
        ct_jobs = [[0 for j in range(len(sequence))] for i in range(self.f)]
        #print("Vector de cada trabajo en cada fábrica", ct_jobs)
        vector_aux = [[0 for j in range(len(sequence))] for i in range(self.f) for k in range(self.machines)]
        #print("Creamos un vector auxiliar",vector_aux)

        aux_ctjobs = [0 for j in range(self.f)] #vector para guardar lo que tarda cada trabajo en cada fábrica

        for j in range(0,len(sequence)):
            # print("Vector del trabajo en todas las fábricas",aux_ctjobs)
            # el trabajo irá a la fábrica con menor makespan, en caso de empate a la de menos wf
            minct = aux_ctjobs[find_index_min(aux_ctjobs)]
            # print("El mínimo completion time es", minct)
            index = []
            for l in range(len(aux_ctjobs)):
                if minct == aux_ctjobs[l]:
                    # print("Posible fábrica asignada", l)
                    index.append(l)
            # print("Fábricas candidatas a insertar trabajos", index)
            minwf = self.wf[index[0]]
            fabrica_asignada = copy.deepcopy(index[0])
            for n in range(len(index)):
```

```
                if self.wf[index[n]] < minwf:
                    minwf = copy.deepcopy(self.wf[index[n]])
                    fabrica_asignada = copy.deepcopy(index[n])
            # print("El trabajo {} va a la fábrica {} cuyo peso es {}".format(sequence[j],fabrica_asignada,minwf))
            for i in range(self.machines):
                    if fabrica_asignada == 0:
                        if i == 0:
                            ct[i][j] = max(max(vector_aux[i]), self.r[sequence[j]]) + self.pt[i][sequence[j]]
                            vector_aux[i][j] = ct[i][j]
                        else:
                            ct[i][j] = max(max(vector_aux[i-1]), max(vector_aux[i])) + self.pt[i][sequence[j]]
                            vector_aux[i][j] = ct[i][j]
                            aux_ctjobs[fabrica_asignada] = vector_aux[i][j]
                    else:
                        if i == 0:
                            ct[i][j] = max(self.r[sequence[j]], max(vector_aux[self.machines*fabrica_asignada])) + self.pt[i][sequence[j]]
                            vector_aux[self.machines*fabrica_asignada][j] = ct[i][j]
                        else:
                            ct[i][j] = max(max(vector_aux[self.machines*fabrica_asignada+i]), max(vector_aux[self.machines*fabrica_asignada+i-1])) + self.pt[i][sequence[j]]
                            vector_aux[self.machines*fabrica_asignada+i][j] = ct[i][j]
                            aux_ctjobs[fabrica_asignada] = copy.deepcopy(vector_aux[self.machines*fabrica_asignada+i][j])

            # print("vector auxiliar ", vector_aux)
            # print("Vector de los completion times finales del trabajo",aux_ctjobs)
            ct_jobs[fabrica_asignada][j] = copy.deepcopy(aux_ctjobs[fabrica_asignada])
        # print(ct_jobs)

        return ct_jobs,sequence
```

evaluaECT

```
from DPFS_ECT import *
from scheptk.util import *
import xlsxwriter

#SHORTEST PROCESSING TIME
def SPT(instancia):
    pijtot = [0 for i in range(instancia.jobs)]
    for i in range (instancia.jobs):
        for j in range(instancia.machines):
            pijtot[i] = sum(instancia.pt[j][i] for j in range(instancia.machines))
    S = sorted_index_asc(pijtot)
    obj = instancia.WjCj_Wf(S)
    return S,obj

#LARGEST PROCESSING TIME
def LPT(instancia):
    pijtot = [0 for j in range(instancia.jobs)]
    for i in range(instancia.jobs):
        for j in range(instancia.machines):
            pijtot[i] = sum(instancia.pt[j][i] for j in range(instancia.machines))
    S = sorted_index_desc(pijtot)
    obj = instancia.WjCj_Wf(S)
    return S,obj

#WEIGHTED SHORTEST PROCESSING TIME
def WSPT(instancia):
    pijtot = [0 for j in range(instancia.jobs)]
    for i in range(instancia.jobs):
        for j in range(instancia.machines):
            pijtot[i] = sum(instancia.pt[j][i] for j in range(instancia.machines))
    cociente = [0 for j in range(instancia.jobs)]
    for j in range(instancia.jobs):
        cociente[j] = instancia.w[j]/pijtot[j]
    S = sorted_index_desc(cociente)
    obj = instancia.WjCj_Wf(S)
    return S,obj

#SPT INTERCALADO
def SPT_INT(instancia):
```

```
    S,obj = SPT(instancia)
    nuevaS = []
    for i in range(len(S) // 2):
        nuevaS.append(S[i])
        nuevaS.append(S[len(S) - 1 - i])
    # Si la longitud del vector original es impar, agregamos el elemento central
    if len(S) % 2 != 0:
        nuevaS.append(S[len(S) // 2])
    obj = instancia.WjCj_Wf(nuevaS)

    return nuevaS,obj

#LPT INTERCALADO
def LPT_INT(instancia):
    S,obj = LPT(instancia)
    nuevaS = []
    for i in range(len(S) // 2):
        nuevaS.append(S[i])
        nuevaS.append(S[len(S) - 1 - i])
    # Si la longitud del vector original es impar, agregamos el elemento central
    if len(S) % 2 != 0:
        nuevaS.append(S[len(S) // 2])
    obj = instancia.WjCj_Wf(nuevaS)

    return nuevaS,obj

resultados = []

#empezamos evaluando las instancias pequeñas
for f in range(2,5):
    for j in range (10,18,2):
        for m in range(2,6):
            archivo = 'instancia_f' + str(f) + '_j' + str(j) + '_m' + str(m) + '.txt'
            instancia = DPFS_ECT(archivo)
            S0,obj0 = SPT(instancia)
            S1,obj1 = LPT(instancia)
            S2,obj2 = WSPT(instancia)
            S3,obj3 = SPT_INT(instancia)
            S4,obj4 = LPT_INT(instancia)
```

```
            with open('resultadosECT.txt','a') as resultadosECT:
                resultadosECT.write(archivo + ' objSPT: ' + str(obj0) +
                                    ' objLPT: ' +  str(obj1) +
                                    ' objWSPT: ' +  str(obj2) +
                                    ' objSPT_INT: ' + str(obj3) +
                                    ' objLPT_INT: ' + str(obj4) + '\n')
            datos = [obj0,obj1,obj2,obj3,obj4]
            resultados.append(datos)

#evaluamos ahora las instancias grandes
for f in range(2,8):
    for j in (20,50,100):
        for m in (5,10,20):
            archivo = 'instancia_f' + str(f) + '_j' + str(j) + '_m' + str(m) + '.txt'
            instancia = DPFS_ECT(archivo)
            S0,obj0 = SPT(instancia)
            S1,obj1 = LPT(instancia)
            S2,obj2 = WSPT(instancia)
            S3,obj3 = SPT_INT(instancia)
            S4,obj4 = LPT_INT(instancia)

            with open('resultadosECT.txt','a') as resultadosECT:
                resultadosECT.write(archivo + ' objSPT: ' + str(obj0) +
                                    ' objLPT: ' +  str(obj1) +
                                    ' objWSPT: ' +  str(obj2) +
                                    ' objSPT_INT: ' + str(obj3) +
                                    ' objLPT_INT: ' + str(obj4) + '\n')
            datos = [obj0,obj1,obj2,obj3,obj4]
            resultados.append(datos)

doc = xlsxwriter.Workbook('resultadosECT.xlsx')
hoja = doc.add_worksheet()
for k in range(len(resultados)):
    for i in range(len(resultados[k])):
        hoja.write(k,i,resultados[k][i])
doc.close()
```

evaluaFAM

```
from DPFS_FAM import *
from scheptk.util import *
import xlsxwriter

#SHORTEST PROCESSING TIME
def SPT(instancia):
    pijtot = [0 for i in range(instancia.jobs)]
    for i in range (instancia.jobs):
        for j in range(instancia.machines):
            pijtot[i] = sum(instancia.pt[j][i] for j in range(instancia.machines))
    S = sorted_index_asc(pijtot)
    obj = instancia.WjCj_Wf(S)
    return S,obj

#LARGEST PROCESSING TIME
def LPT(instancia):
    pijtot = [0 for j in range(instancia.jobs)]
    for i in range(instancia.jobs):
        for j in range(instancia.machines):
            pijtot[i] = sum(instancia.pt[j][i] for j in range(instancia.machines))
    S = sorted_index_desc(pijtot)
    obj = instancia.WjCj_Wf(S)
    return S,obj

#WEIGHTED SHORTEST PROCESSING TIME
def WSPT(instancia):
    pijtot = [0 for j in range(instancia.jobs)]
    for i in range(instancia.jobs):
        for j in range(instancia.machines):
            pijtot[i] = sum(instancia.pt[j][i] for j in range(instancia.machines))
    cociente = [0 for j in range(instancia.jobs)]
    for j in range(instancia.jobs):
        cociente[j] = instancia.w[j]/pijtot[j]
    S = sorted_index_desc(cociente)
    obj = instancia.WjCj_Wf(S)
    return S,obj

#SPT INTERCALADO
def SPT_INT(instancia):
```

```
    S,obj = SPT(instancia)
    nuevaS = []
    for i in range(len(S) // 2):
        nuevaS.append(S[i])
        nuevaS.append(S[len(S) - 1 - i])
    # Si la longitud del vector original es impar, agregamos el elemento central
    if len(S) % 2 != 0:
        nuevaS.append(S[len(S) // 2])
    obj = instancia.WjCj_Wf(nuevaS)

    return nuevaS,obj

#LPT INTERCALADO
def LPT_INT(instancia):
    S,obj = LPT(instancia)
    nuevaS = []
    for i in range(len(S) // 2):
        nuevaS.append(S[i])
        nuevaS.append(S[len(S) - 1 - i])
    # Si la longitud del vector original es impar, agregamos el elemento central
    if len(S) % 2 != 0:
        nuevaS.append(S[len(S) // 2])
    obj = instancia.WjCj_Wf(nuevaS)

    return nuevaS,obj

resultados = []

#empezamos evaluando las instancias pequeñas
for f in range(2,5):
    for j in range (10,18,2):
        for m in range(2,6):
            archivo = 'instancia_f' + str(f) + '_j' + str(j) + '_m' + str(m) + '.txt'
            instancia = DPFS_FAM(archivo)
            S0,obj0 = SPT(instancia)
            S1,obj1 = LPT(instancia)
            S2,obj2 = WSPT(instancia)
            S3,obj3 = SPT_INT(instancia)
            S4,obj4 = LPT_INT(instancia)
```

```
            with open('resultadosFAM.txt','a') as resultadosFAM:
                resultadosFAM.write(archivo + ' objSPT: ' + str(obj0) +
                                    ' objLPT: ' +  str(obj1) +
                                    ' objWSPT: ' +  str(obj2) +
                                    ' objSPT_INT: ' + str(obj3) +
                                    ' objLPT_INT: ' + str(obj4) + '\n')
            datos = [obj0,obj1,obj2,obj3,obj4]
            resultados.append(datos)
#evaluamos ahora las instancias grandes
for f in range(2,8):
    for j in (20,50,100):
        for m in (5,10,20):
            archivo = 'instancia_f' + str(f) + '_j' + str(j) + '_m' + str(m) + '.txt'
            instancia = DPFS_FAM(archivo)
            S0,obj0 = SPT(instancia)
            S1,obj1 = LPT(instancia)
            S2,obj2 = WSPT(instancia)

            with open('resultadosFAM.txt','a') as resultadosFAM:
                resultadosFAM.write(archivo + ' objSPT: ' + str(obj0) +
                                    ' objLPT: ' +  str(obj1) +
                                    ' objWSPT: ' +  str(obj2) +
                                    ' objSPT_INT: ' + str(obj3) +
                                    ' objLPT_INT: ' + str(obj4) + '\n')
            datos = [obj0,obj1,obj2,obj3,obj4]
            resultados.append(datos)

doc = xlsxwriter.Workbook('resultadosFAM.xlsx')
hoja = doc.add_worksheet()
for k in range(len(resultados)):
    for i in range(len(resultados[k])):
        hoja.write(k,i,resultados[k][i])
doc.close()
```

evaluaECT_lowestWF

```
from DPFS_ECT_lowestWF import *
from scheptk.util import *
import xlsxwriter

#SHORTEST PROCESSING TIME
def SPT(instancia):
```

```
    pijtot = [0 for i in range(instancia.jobs)]
    for i in range (instancia.jobs):
        for j in range(instancia.machines):
            pijtot[i] = sum(instancia.pt[j][i] for j in range(instancia.machines))
    S = sorted_index_asc(pijtot)
    obj = instancia.WjCj_Wf(S)
    return S,obj

#LARGEST PROCESSING TIME
def LPT(instancia):
    pijtot = [0 for j in range(instancia.jobs)]
    for i in range(instancia.jobs):
        for j in range(instancia.machines):
            pijtot[i] = sum(instancia.pt[j][i] for j in range(instancia.machines))
    S = sorted_index_desc(pijtot)
    obj = instancia.WjCj_Wf(S)
    return S,obj

#WEIGHTED SHORTEST PROCESSING TIME
def WSPT(instancia):
    pijtot = [0 for j in range(instancia.jobs)]
    for i in range(instancia.jobs):
        for j in range(instancia.machines):
            pijtot[i] = sum(instancia.pt[j][i] for j in range(instancia.machines))
    cociente = [0 for j in range(instancia.jobs)]
    for j in range(instancia.jobs):
        cociente[j] = instancia.w[j]/pijtot[j]
    S = sorted_index_desc(cociente)
    obj = instancia.WjCj_Wf(S)
    return S,obj

#SPT INTERCALADO
def SPT_INT(instancia):
    S,obj = SPT(instancia)
    nuevaS = []
    for i in range(len(S) // 2):
        nuevaS.append(S[i])
        nuevaS.append(S[len(S) - 1 - i])
    # Si la longitud del vector original es impar, agregamos el elemento central
    if len(S) % 2 != 0:
```

```
        nuevaS.append(S[len(S) // 2])
    obj = instancia.WjCj_Wf(nuevaS)

    return nuevaS,obj

#LPT INTERCALADO
def LPT_INT(instancia):
    S,obj = LPT(instancia)
    nuevaS = []
    for i in range(len(S) // 2):
        nuevaS.append(S[i])
        nuevaS.append(S[len(S) - 1 - i])
    # Si la longitud del vector original es impar, agregamos el elemento central
    if len(S) % 2 != 0:
        nuevaS.append(S[len(S) // 2])
    obj = instancia.WjCj_Wf(nuevaS)

    return nuevaS,obj

resultados = []

#empezamos evaluando las instancias pequeñas
for f in range(2,5):
    for j in range (10,18,2):
        for m in range(2,6):
            archivo = 'instancia_f' + str(f) + '_j' + str(j) + '_m' + str(m) + '.txt'
            instancia = DPFS_ECT_WF(archivo)
            S0,obj0 = SPT(instancia)
            S1,obj1 = LPT(instancia)
            S2,obj2 = WSPT(instancia)
            S3,obj3 = SPT_INT(instancia)
            S4,obj4 = LPT_INT(instancia)

            with open('resultadosECTlowestWF.txt','a') as resultadosECTlowestWF:
                resultadosECTlowestWF.write(archivo + ' objSPT: ' + str(obj0) +
                                            ' objLPT: ' +  str(obj1) +
                                            ' objWSPT: ' +  str(obj2) +
                                            ' objSPT_INT: ' + str(obj3) +
                                            ' objLPT_INT: ' + str(obj4) + '\n')
            datos = [obj0,obj1,obj2,obj3,obj4]
```

```
            resultados.append(datos)

#evaluamos ahora las instancias grandes
for f in range(2,8):
    for j in (20,50,100):
        for m in (5,10,20):
            archivo = 'instancia_f' + str(f) + '_j' + str(j) + '_m' + str(m) + '.txt'
            instancia = DPFS_ECT_WF(archivo)
            S0,obj0 = SPT(instancia)
            S1,obj1 = LPT(instancia)
            S2,obj2 = WSPT(instancia)
            S3,obj3 = SPT_INT(instancia)
            S4,obj4 = LPT_INT(instancia)

            with open('resultadosECTlowestWF.txt','a') as resultadosECTlowestWF:
                resultadosECTlowestWF.write(archivo + ' objSPT: ' + str(obj0) +
                                            ' objLPT: ' +  str(obj1) +
                                            ' objWSPT: ' +  str(obj2) +
                                            ' objSPT_INT: ' + str(obj3) +
                                            ' objLPT_INT: ' + str(obj4) + '\n')
            datos = [obj0,obj1,obj2,obj3,obj4]
            resultados.append(datos)

doc = xlsxwriter.Workbook('resultadosECTlowestWF.xlsx')
hoja = doc.add_worksheet()
for k in range(len(resultados)):
    for i in range(len(resultados[k])):
        hoja.write(k,i,resultados[k][i])
doc.close()
```

evaluaFAM_lowestWF

```
from DPFS_FAM_lowestWF import *
from scheptk.util import *
import xlsxwriter

#SHORTEST PROCESSING TIME
def SPT(instancia):
    pijtot = [0 for i in range(instancia.jobs)]
    for i in range (instancia.jobs):
        for j in range(instancia.machines):
            pijtot[i] = sum(instancia.pt[j][i] for j in range(instancia.machines))
```

```
    S = sorted_index_asc(pijtot)
    obj = instancia.WjCj_Wf(S)
    return S,obj

#LARGEST PROCESSING TIME
def LPT(instancia):
    pijtot = [0 for j in range(instancia.jobs)]
    for i in range(instancia.jobs):
        for j in range(instancia.machines):
            pijtot[i] = sum(instancia.pt[j][i] for j in range(instancia.machines))
    S = sorted_index_desc(pijtot)
    obj = instancia.WjCj_Wf(S)
    return S,obj

#WEIGHTED SHORTEST PROCESSING TIME
def WSPT(instancia):
    pijtot = [0 for j in range(instancia.jobs)]
    for i in range(instancia.jobs):
        for j in range(instancia.machines):
            pijtot[i] = sum(instancia.pt[j][i] for j in range(instancia.machines))
    cociente = [0 for j in range(instancia.jobs)]
    for j in range(instancia.jobs):
        cociente[j] = instancia.w[j]/pijtot[j]
    S = sorted_index_desc(cociente)
    obj = instancia.WjCj_Wf(S)
    return S,obj

#SPT INTERCALADO
def SPT_INT(instancia):
    S,obj = SPT(instancia)
    nuevaS = []
    for i in range(len(S) // 2):
        nuevaS.append(S[i])
        nuevaS.append(S[len(S) - 1 - i])
    # Si la longitud del vector original es impar, agregamos el elemento central
    if len(S) % 2 != 0:
        nuevaS.append(S[len(S) // 2])
    obj = instancia.WjCj_Wf(nuevaS)

    return nuevaS,obj
```

```
#LPT INTERCALADO
def LPT_INT(instancia):
    S,obj = LPT(instancia)
    nuevaS = []
    for i in range(len(S) // 2):
        nuevaS.append(S[i])
        nuevaS.append(S[len(S) - 1 - i])
    # Si la longitud del vector original es impar, agregamos el elemento central
    if len(S) % 2 != 0:
        nuevaS.append(S[len(S) // 2])
    obj = instancia.WjCj_Wf(nuevaS)

    return nuevaS,obj

resultados = []

#empezamos evaluando las instancias pequeñas
for f in range(2,5):
    for j in range (10,18,2):
        for m in range(2,6):
            archivo = 'instancia_f' + str(f) + '_j' + str(j) + '_m' + str(m) + '.txt'
            instancia = DPFS_FAM_WF(archivo)
            S0,obj0 = SPT(instancia)
            S1,obj1 = LPT(instancia)
            S2,obj2 = WSPT(instancia)
            S3,obj3 = SPT_INT(instancia)
            S4,obj4 = LPT_INT(instancia)

            with open('resultadosFAMlowestWF.txt','a') as resultadosFAMlowestWF:
                resultadosFAMlowestWF.write(archivo + ' objSPT: ' + str(obj0) +
                                            ' objLPT: ' +  str(obj1) +
                                            ' objWSPT: ' +  str(obj2) +
                                            ' objSPT_INT: ' + str(obj3) +
                                            ' objLPT_INT: ' + str(obj4) + '\n')
            datos = [obj0,obj1,obj2,obj3,obj4]
            resultados.append(datos)

#evaluamos ahora las instancias grandes
for f in range(2,8):
```

```
    for j in (20,50,100):
        for m in (5,10,20):
            archivo = 'instancia_f' + str(f) + '_j' + str(j) + '_m' + str(m) + '.txt'
            instancia = DPFS_FAM_WF(archivo)
            S0,obj0 = SPT(instancia)
            S1,obj1 = LPT(instancia)
            S2,obj2 = WSPT(instancia)
            S3,obj3 = SPT_INT(instancia)
            S4,obj4 = LPT_INT(instancia)

            with open('resultadosFAMlowestWF.txt','a') as resultadosFAMlowestWF:
                resultadosFAMlowestWF.write(archivo + ' objSPT: ' + str(obj0) +
                                            ' objLPT: ' +  str(obj1) +
                                            ' objWSPT: ' +  str(obj2) +
                                            ' objSPT_INT: ' + str(obj3) +
                                            ' objLPT_INT: ' + str(obj4) + '\n')
            datos = [obj0,obj1,obj2,obj3,obj4]
            resultados.append(datos)

doc = xlsxwriter.Workbook('resultadosFAMlowestWF.xlsx')
hoja = doc.add_worksheet()
for k in range(len(resultados)):
    for i in range(len(resultados[k])):
        hoja.write(k,i,resultados[k][i])
doc.close()
```

BWO

```
from DPFS_FAM import *
from scheptk.util import *
import random,copy,math,time
import numpy as np

def BWO(instancia,PS,PP,CR,PM):
    # creamos la población inicial fija
    # IND = []
    # while len(IND)<PS:
    #     S = random_sequence(instancia.jobs)
    #     # print(S)
    #     if S not in IND:
    #         IND.append(S)
    # print("Tenemos ya la población de {} individuos" .format(len(IND)),IND)
```

```
    #vector para guardar los mejores individuos en cad iteración
    mejores_ind = []
    #establecemos el número de individuos elegidos igual a 4
    a = 4
    # it = 0
    tiempo_maximo = instancia.jobs*instancia.machines*60/1000
    # print(tiempo_maximo)
    tpo = 0 #(fin - inicio)
    ini = time.time()
    while tpo<tiempo_maximo:
        # creamos la población inicial en cada it
        IND = []
        while len(IND)<PS:
            S = random_sequence(instancia.jobs)
            # print(S)
            if S not in IND:
                IND.append(S)
        # print("Tenemos ya la población de {} individuos" .format(len(IND)),IND)

        ####################FASE DE PROCREACIÓN####################

        #elegimos nr indiviudos como población candidata
        nr = int(PS*PP)
        # print(nr)
        pares_padres = []
        # creamos nr pares de padres a partir de la población candidata de nr individuos
        while len(pares_padres)<nr:
            candidatos = random.sample(IND,nr)
            # print("Población candidata", candidatos)
            #seleccionamos cuatro individuos y nos quedamos con el mejor
            individuos_torneo1 = random.sample(candidatos,a)
            # print("Indiviudos del torneo 1",individuos_torneo1)
            padre1 = min(individuos_torneo1, key=instancia.WjCj_Wf)
            # print("Padre 1",padre1)
            #volvemos a seleecionar cuatro individuos y nos quedamos con el mejor, si
coincide con el padre1 repetimos
            individuos_torneo2 = random.sample(candidatos,a)
            # print("Indiviudos del torneo 2",individuos_torneo2)
            padre2 = min(individuos_torneo2, key=instancia.WjCj_Wf)
            # print("Padre 2 provisonal", padre2)
```

```
            #si el padre 2 es igual que el padre 1 volvemos a realizar el torneo de
selección hasta que sean distintos
            while padre1 == padre2:
                # print("El padre 2 es igual que el padre 1", padre1,padre2)
                individuos_torneo2 = random.sample(candidatos,a)
                padre2 = min(individuos_torneo2, key=instancia.WjCj_Wf)
            # print("Tras comprobar que los padres son distintos:")
            # print("Padre 2", padre2)
            # cuando ya tenemos los dos pares de padres el que tenga menor valor de la
FO será la hembra
            # print("Los valores de las funciones objetivos son: FO padre1: {}, FO padre
2: {}".format(instancia.WjCj_Wf(padre1),instancia.WjCj_Wf(padre2)))
            if instancia.WjCj_Wf(padre1) > instancia.WjCj_Wf(padre2):
                hembra = padre2[:]
                macho = padre1[:]
            else:
                hembra = padre1[:]
                macho = padre2[:]
            # print("Hembra y macho",hembra,macho)


            if [hembra,macho] not in pares_padres:
                pares_padres.append([hembra,macho])


        # print("Pares de padres sin elementos repetido",pares_padres,len(pares_padres))


        #una vez tenemos los padres generamos dos hijos por cada par de padres, esto lo
repetimos 4 veces, obteniendo un total de 8·nr nuevos individuos
        #vector para guardar los hijos que se van creando
        hijos = []
        hijos_set = set()
        for k in range(len(pares_padres)):
            while len(hijos)<(8*(k+1)):
                    # definimos dos puntos, los elementos del padre que se encuentren
entre estos dos puntos, serán heredados directamente por los hijos
                    punto = sorted(random.sample(range(instancia.jobs), 2))
                    # print("Punto",punto)
                    child1 = [None] * instancia.jobs
                    child2 = [None] * instancia.jobs
                    child1[punto[0]:punto[1]+1]                                          =
pares_padres[k][0][punto[0]:punto[1]+1]
                    child2[punto[0]:punto[1]+1]                                          =
pares_padres[k][1][punto[0]:punto[1]+1]
```

```
                    # print("Empezamos a construir los hijos",child1,child2)
                    #ahora insertamos en las posiciones vacías del hijo 1 los elementos
del padre2,
                    #que no están repetidos, en el orden en el que aparecen
                    for j in range(instancia.jobs):
                        if pares_padres[k][1][j] not in child1:
                            index = child1.index(None)
                            child1[index] = pares_padres[k][1][j]
                    # print("Hijo 1",child1)

                    #hacemos lo mismo para crear el hijo 2
                    for j in range(instancia.jobs):
                        if pares_padres[k][0][j] not in child2:
                            index = child2.index(None)
                            child2[index] = pares_padres[k][0][j]
                    # print("Hijo 2",child2)

                    if  tuple(child1)  not  in  hijos_set  and  tuple(child2)  not  in
hijos_set:
                        hijos.append(child1)
                        hijos_set.add(tuple(child1))
                        hijos.append(child2)
                        hijos_set.add(tuple(child2))
        # print("Hijos",hijos,len(hijos))

        ####################FASE DE CANIBALISMO####################

        #empezamos calculando el número de individuos superviviente nc según la tasa de
canibalismo
        nc = round((8*nr)*(1-CR))
        # print("nc:", nc)
        #ahora ordenamos los hijos de menor a mayor en función de la FO y nos quedamos
con los nc mejores
        hijos_ordenados = sorted(hijos, key=instancia.WjCj_Wf)
        supervivientes = hijos_ordenados[:nc]
        # print("Supervivientes",supervivientes,len(supervivientes))

        ####################FASE DE MUTACIÓN####################

        #primero seleccionamos nm indiviudos de la población candidata
        nm = round(PS*PM)
```

```
        # print("nm",nm)
        pob = random.sample(IND,nm)
        # print("Poblacion",pob,len(pob))
        #una vez que tenemos los nm individuos seleccionados aplicamos SA a cada uno de
ellos
        for j in range(nm):
            #definimos en primer lugar la temperatura(T = -(Ow-Ob)/ln(pr)), donde pr=0.1
            # worst = max(pob, key=instancia.WjCj_Wf)
            # Ow = instancia.WjCj_Wf(worst)
            # print(Ow)
            # best = min(pob, key=instancia.WjCj_Wf)
            # Ob = instancia.WjCj_Wf(best)
            # print(Ob)
            # T = -(Ow-Ob)/np.log(0.1)
            T = 50
            # print("Temperatura",T)
            # establecemos una temperatura final
            Tf = 1e-9
            pi1 = pob[j]
            #establecemos pi1 como la mejor solución conocida hasta ahora
            pib = pi1[:]
            #definimos el valor de la FO de pi1
            obj1 = instancia.WjCj_Wf(pi1)
            objb = obj1
            # print("Pi1",pi1,objb)
            #establecemos la solución actual
            pic = pib[:]
            tiempo = 0 #(fin - inicio)
            inicio = time.time()
            #mientras no se cumplan las condiciones de parada iteramos y construimos
vecinos
            # v = 0
            while True:
                if T<=Tf:
                    # print("Se ha alcanzado la temperatura")
                    break
                pi = pic[:]
                #construimos un vecino de pi1
                x = random.randint(0,instancia.jobs-2)
                # print(x)
                pi2 = pi[:]
```

```
                pi2[x] = pi[x+1]
                pi2[x+1] = pi[x]
                obj2 = instancia.WjCj_Wf(pi2)
                # print("Pi2",pi2,obj2)
                if obj2 < objb:
                    # print("El vecino {} es mejor que {} porque su valor de la FO es
mejor {} < {}" .format(pi2,pib,obj2,objb))
                    pib = pi2[:]
                    pic = pi2[:]
                    objb = obj2
                    break
                    #si el valor de la FO no es mejor podemos aceptar la solución como
admisible con una cierta probabilidad, esto nos ayudará a escapar de óptimos locales
                else:
                    if random.uniform(0,1)<=math.exp(-(obj2-objb)/T):
                        # print(math.exp(-(obj2-objb)/T))
                        # print("Admisible")
                        pic = pi2[:]
                    # else:
                    #     print("No se cumple el criterio de aceptación")
                # print("Para el individuo {} ({}) el mejor individuo obtenido es {}
({})" .format(pi1,obj1,pib,objb))
                # print("pic", pic)
                pob[j] = pib
                T = T*0.99
                # print("Temperatura", T)
                final = time.time()
                tiempo = final - inicio
                # v = v + 1
            # print("It de vecinos",v,j)
            # print("Tiempo",tiempo)


        # print(pob)


        #una vez tenemos la poblacion modificada tenemos en total nc+nm individuos, de
este grupo nos quedamos con el mejor
        pob_total = pob + supervivientes
        # print(pob_total,len(pob_total))
        #buscamos de la población cual es el mejor individuo
        best = min(pob_total, key=instancia.WjCj_Wf)
        # print("Mejor individuo",best, instancia.WjCj_Wf(best))
```

```
        fin = time.time()
        tpo = fin - ini
        mejores_ind.append(best)
        # it = it + 1
        # print("Iteraciones totales",it)
    # print("Mejores individuos otenidos en cada iteración",mejores_ind)
    # print("Tpo",tpo)
    mejor_ind = min(mejores_ind, key=instancia.WjCj_Wf)
    obj_fin = instancia.WjCj_Wf(mejor_ind)
    # print("El mejor individuo obtenido es", mejor_ind)

    return mejor_ind,obj_fin
```

evaluaBWO

```
from DPFS_FAM import DPFS_FAM
from BWO import BWO
import xlsxwriter
resultados = []
#empezamos evaluando las instancia pequeñas
for f in range(2,5):
    for j in range(10,18,2):
        for m in range(2,6):
            archivo = 'instancia_f' + str(f) + '_j' + str(j) + '_m' + str(m) + '.txt'
            instancia = DPFS_FAM(archivo)
            S,obj = BWO(instancia,40,0.60,0.45,0.45)

            with open('resultadosBWO3.txt','a') as resultadosBWO3:
                resultadosBWO3.write(archivo + 'FO' + str(obj) + '\n')
            datos = [obj]
            resultados.append(datos)

#evaluamos ahora las instancias grandes
for f in range(2,8):
    for j in (20,50,100):
        for m in (5,10,20):
            archivo = 'instancia_f' + str(f) + '_j' + str(j) + '_m' + str(m) + '.txt'
            instancia = DPFS_FAM(archivo)
            S,obj = BWO(instancia,40,0.60,0.45,0.45)

            with open('resultadosBWO3.txt','a') as resultadosBWO3:
                resultadosBWO3.write(archivo + 'FO' + str(obj) + '\n')
```

```
            datos = [obj]
            resultados.append(datos)


doc = xlsxwriter.Workbook('resultadosBWO3.xlsx')
hoja = doc.add_worksheet()
for k in range(len(resultados)):
    for i in range(len(resultados[k])):
        hoja.write(k,i,resultados[k][i])
doc.close()
```

IteratedGreedy

```
from DPFS_FAM import *
from scheptk.util import *
from NEH_FO import NEH_FO
from localSearch import localSearch
import copy, random, math, time

def IG(instancia):
    #usamos la heurística NEH para generar una solución inicial
    pi,obj_ini = NEH_FO(instancia)
    # print("Solución inicial PI:", pi,obj_ini)
    pib = copy.deepcopy(pi)
    objb = obj_ini
    #calculamos la temperatura T, para el criterio de aceptación
    #calculamos primero el tiempo total de proceso
    pijtot = [0 for i in range(instancia.jobs)]
    for i in range (instancia.jobs):
        for j in range(instancia.machines):
            pijtot[i] = sum(instancia.pt[j][i] for j in range(instancia.machines))
    ptot = sum(pijtot)
    T = (ptot/(instancia.jobs*instancia.machines*10))*0.4

    tiempo_maximo = instancia.jobs*instancia.machines*60/1000
    tpo = 0 #(fin - inicio)
    ini = time.time()
    # v = 0
    while tpo<tiempo_maximo:
        k = 2
        # k = random.randint(2,instancia.jobs/2)
        # print("Número de trabajos a borrar",k)
```

```
        pip = pi[:]
        # print(pip)
        pir = [None]*k
        # print("pir", pir)
        pid = pip[:]
        #construimos la secuencia pir
        for j in range(0,k):
            job = random.choice(pid)
            # print("Trabajo a borrar", job)
            pid.remove(job)
            # print("PId tras borrar el trabajo {}: {}" .format(job,pid))
            pir[j] = job
            # print("Pir tras insertar los trabajos borrados", pir)
        #una vez tenemos la secuencia parcial pir aplicamos una busqueda local para
intentar obtener una mejor secuencia
        pir_ls = localSearch(pir, instancia)
        # print("La nueva secuencia pir tras una búsqueda local es:", pir_ls)
        # print("La secuencia pid tras borrar los trabajos es",pid)
        #construimos la secuencia pid insertando los trabajos de la sec. pir en la mejor
posición de la sec. pid
        for j in range(0,k):
            best_obj = float('inf')
            best_sec = [None]
            for i in range(len(pid)+1):
                sec = pid[:]
                # print("Secuencia antes de insertar trabajos", sec)
                sec.insert(i,pir_ls[j])
                #  print("Secuencia tras insertar {} en la posición {}: {}"
.format(pir_ls[j],i,sec))
                obj = instancia.WjCj_Wf(sec)
                # print("El valor de la función objetivo es", obj)
                if obj<best_obj or i==0:
                    #  print("La secuencia es mejor porque {}({}) < {}({})"
.format(obj,sec,best_obj,best_sec))
                    best_obj = obj
                    best_sec = sec[:]
                    if i>0:
                        break
                if len(sec)>25 and i>10:
                    # print("Se ha alcanzado el número máximo de iterraciones")
                    break
```

```
            pid = best_sec[:]
            # print("Secuencia tras hacer la inserción",pid,instancia.WjCj_Wf(pid))
        pid_ls = localSearch(pid, instancia)
        pip = pid_ls[:]
        objp = instancia.WjCj_Wf(pip)
        # print("Secuencia definitva tras hacer la búsqueda local",pid_ls,objp)
        if objp < obj_ini:
            pi = pip[:]
            obj_ini = objp
            if obj_ini < objb:
                pib = pi[:]
                objb = obj_ini
        else:
            if random.uniform(0,1)<= math.exp(-(objp-objb)/T):
                # print("Se cumple el criterio de aceptación")
                pi = pip[:]
        fin = time.time()
        tpo = fin - ini
        # v = v + 1
        # print("Iteraciones",v)
    # print(tpo)
    # print("La secuencia obtenida por Iterated Greedy es:", pib,objb)

    return pib,objb
```

evalualG

```
from DPFS_FAM import DPFS_FAM
from IteratedGreedy import IG
import xlsxwriter

resultados = []
#empezamos evaluando las instancia pequeñas
for f in range(2,5):
    for j in range(10,18,2):
        for m in range(2,6):
            archivo = 'instancia_f' + str(f) + '_j' + str(j) + '_m' + str(m) + '.txt'
            instancia = DPFS_FAM(archivo)
            S,obj = IG(instancia)

            with open('resultadosIG2.txt','a') as resultadosIG2:
                resultadosIG2.write(archivo + ' FO ' + str(obj) + '\n')
```

```
            datos = [obj]
            resultados.append(datos)

#evaluamos ahora las instancias grandes
for f in range(2,8):
    for j in (20,50,100):
        for m in (5,10,20):
            archivo = 'instancia_f' + str(f) + '_j' + str(j) + '_m' + str(m) + '.txt'
            instancia = DPFS_FAM(archivo)
            S,obj = IG(instancia)

            with open('resultadosIG2.txt','a') as resultadosIG2:
                resultadosIG2.write(archivo + ' FO ' + str(obj) + '\n')
            datos = [obj]
            resultados.append(datos)

doc = xlsxwriter.Workbook('resultadosIG2.xlsx')
hoja = doc.add_worksheet()
for k in range(len(resultados)):
    for i in range(len(resultados[k])):
        hoja.write(k,i,resultados[k][i])
doc.close()
```

NEH_FO

```
from DPFS_FAM import *
from scheptk.util import *
import copy,time

def NEH_FO(instancia):
    pijtot = [0 for i in range(instancia.jobs)]
    for i in range (instancia.jobs):
        for j in range(instancia.machines):
            pijtot[i] = sum(instancia.pt[j][i] for j in range(instancia.machines))
    S = sorted_index_desc(pijtot)
    # print("Secuencia",S)
    pi = [S[0]]
    # print("Pi",pi)
    tpo = 0 #(fin - inicio)
    ini = time.time()
    for j in range(1,instancia.jobs):
        best_obj = float('inf')
```

```
        best_sec = [None]
        for i in range(len(pi)+1):
            # print("Mejor secuencia y valor objetivo", best_sec,best_obj)
            sec = pi[:]
            # print("Secuencia antes de insertar",sec)
            sec.insert(i,S[j])
            #   print("Tras   insertar   {}   en   la   posicion   {}   nos   queda   {}"
.format(S[j],i,sec))
            obj = instancia.WjCj_Wf(sec)
            # print("Valor objetivo d1e la función", obj)
            if obj < best_obj or i==0:
                #   print("La   secuencia   es   mejor   porque   {}   ({})   <   {}   ({})"
.format(obj,sec,best_obj,best_sec))
                best_obj = obj
                best_sec = sec[:]
                # cuando encontramos una solución mejor paramos y no seguimos explorando
insertando en el resto de posiciones
                if i>0:
                    # print("BREAK")
                    break
            #si  la  longitud  de  la  secuencia  es  mayor  de  25  y  no  se  han  encontrado
mejoras tras 10 movimientos paramos
            if len(sec)>25 and i>10:
                # print("Se ha alcanzado el número máximo de iteraciones")
                break
        pi = best_sec[:]
        # print("Secuencia defenitiva",pi)
    fin = time.time()
    tpo = fin - ini
    # print(tpo)
    # print("Secuencia final", pi)
    return pi,best_obj
```

Localsearch

```
import copy,time
from DPFS_FAM import *
from scheptk.util import random_sequence



#VECINDAD INSERTION(LOCAL SEARCH --> FIRST IMPROVEMENT)
def localSearch(seq, instancia):
    best_obj = instancia.WjCj_Wf(seq)
    # print("Valor inicial de la FO",best_obj)
    best_seq = seq[:]
    # print("Secuencia inicial",best_seq)
    tpo = 0 #(fin - inicio)
    ini = time.time()
    for j in range(len(seq)):
        elemento = seq[j]  # Elemento a insertar
        temp_seq = seq[:j] + seq[j+1:]  # Creamos una copia temporal sin el elemento en
la posición j
        # print("Secuencia temporal", temp_seq)
        for i in range(len(seq)):
            if j != i and j!=i+1:
                vecino = temp_seq[:i] + [elemento] + temp_seq[i:]  # Insertamos en la
posición i el elemento
                #  print("Vecino tras insertar el elemento borrado {} en {}: {}"
.format(seq[j],i,vecino))
                # Calculamos el valor de la función objetivo una vez
                obj = instancia.WjCj_Wf(vecino)
                # print("Valor de la FO", obj)


                #cuando la secuencia obtenida es mejor, paramos
                if obj < best_obj:
                    # print("Esta secuencia es mejor porque el valor de la FO es menor
{}<{}".format(obj,best_obj))
                    best_obj = obj
                    best_seq = vecino[:]
                    # if len(seq)>=20:
                    # print("BREAK",i,j)
                    break
        if obj == best_obj:
            break
```

```
    fin = time.time()
    tpo = fin - ini
    # print(tpo)

    return best_seq
```

GA

```
from NEH_FO import NEH_FO
from funcionesvecindades import FIAdjacent,FIGeneral,FIInsertion
from scheptk.util import *
from DPFS_FAM import *
from localSearch import localSearch
import random,time

def GA(instancia):
    #creamos la población inicial
    S,valorFO = NEH_FO(instancia)
    obj1,S1 = FIAdjacent(S,instancia)
    obj2,S2 = FIGeneral(S,instancia)
    obj3,S3 = FIInsertion(S,instancia)
    # print(S1,S2,S3)
    pop = []
    pop = [S1] + [S2] + [S3]
    while len(pop)<50:
        Spop = random_sequence(instancia.jobs)
        if Spop not in pop:
            pop.append(Spop)
    # print(pop,len(pop))

    pib = pop[0]
    objb = instancia.WjCj_Wf(pib)
    # print("Mejor valor",objb)
    for j in range(len(pop)):
        objc = instancia.WjCj_Wf(pop[j])
        # print("Valor de la FO",objc)
        if objc < objb:
            # print("El valor de la FO es menor")
            pib = pop[j]
            objb = objc
    # print("El nuevo valor de pib es {}, cuyo valor de la FO es {}".format(pib,objb))
```

```
tiempo_maximo = instancia.jobs*instancia.machines*60/1000
tpo = 0 #(fin - inicio)
ini = time.time()
# it = 0
while tpo<tiempo_maximo:
# for j in range(1):
    #primero elegimos dos padres de la población inicial
    parent1,parent2 = random.sample(pop,2)
    # print("Par de padres",parent1,parent2)

    #construimos ahora los hijos
    punto = sorted(random.sample(range(instancia.jobs), 2))
    # print("Punto",punto)
    child1 = [None] * instancia.jobs
    child1[punto[0]:punto[1]+1] = parent1[punto[0]:punto[1]+1]

    # print(child1)
    for i in range(instancia.jobs):
        if parent2[i] not in child1:
            index = child1.index(None)
            child1[index] = parent2[i]
    # print("Hijo 1",child1)
    #construimos el hijo 2
    x = random.randint(0,len(parent1)-1)
    # print("X",x)
    longitudchild1 = len(child1)-1
    if x == longitudchild1:
        child2 = copy.deepcopy(child1)
        child2[x] = child1[x-1]
        child2[x-1] = child1[x]
    else:
        child2 = copy.deepcopy(child1)
        child2[x] = child1[x+1]
        child2[x+1] = child1[x]
    # print("Hijo2",child2)

    obj = instancia.WjCj_Wf(child2)

    if obj < objb:
        pib = localSearch(child2, instancia)
```

```
            objb = instancia.WjCj_Wf(pib)
            # print("Individuo y valor FO",pib,objb)
        fin = time.time()
        tpo = fin - ini
        # it = it+1
    # print("IT",it)

    return(pib,objb)
```

evaluaGA

```
from DPFS_FAM import DPFS_FAM
from GA import GA
import xlsxwriter

resultados = []
#empezamos evaluando las instancia pequeñas
for f in range(2,5):
    for j in range(10,18,2):
        for m in range(2,6):
            archivo = 'instancia_f' + str(f) + '_j' + str(j) + '_m' + str(m) + '.txt'
            instancia = DPFS_FAM(archivo)
            S,obj = GA(instancia)

            with open('resultadosGA.txt','a') as resultadosGA:
                resultadosGA.write(archivo + ' FO ' + str(obj) + '\n')
            datos = [obj]
            resultados.append(datos)

#evaluamos ahora las instancias grandes
for f in range(2,8):
    for j in (20,50,100):
        for m in (5,10,20):
            archivo = 'instancia_f' + str(f) + '_j' + str(j) + '_m' + str(m) + '.txt'
            instancia = DPFS_FAM(archivo)
            S,obj = GA(instancia)

            with open('resultadosGA.txt','a') as resultadosGA:
                resultadosGA.write(archivo + ' FO ' + str(obj) + '\n')
            datos = [obj]
            resultados.append(datos)
```

```
doc = xlsxwriter.Workbook('resultadosGA.xlsx')
hoja = doc.add_worksheet()
for k in range(len(resultados)):
    for i in range(len(resultados[k])):
        hoja.write(k,i,resultados[k][i])
doc.close()
```

funcionesvecindades

```
from scheptk.scheptk import *
import sys # to stop the execution (funcion exit() )
import copy

#Intercambio por adyacentes con first improvement
def FIAdjacent(S,instancia):
    best_obj = instancia.max_WjLj(S)
    best_vecino = copy.deepcopy(S)
    for j in range(instancia.jobs-1):
        vecino = copy.deepcopy(S)
        vecino[j] = S[j+1]
        vecino[j+1] = S[j]
        obj = instancia.max_WjLj(vecino)
        if obj<best_obj:
            break;
    return(best_obj,best_vecino)

#General swap con first improvement
def FIGeneral(S,instancia):
    best_obj = instancia.max_WjLj(S)
    best_vecino = copy.deepcopy(S)
    for j in range(instancia.jobs):
        for i in range(instancia.jobs):
            if j<i:
                vecino = copy.deepcopy(S)
                vecino[j] = S[i]
                vecino[i] = S[j]
                obj = instancia.max_WjLj(vecino)
                if obj<best_obj:
                    best_obj = copy.deepcopy(obj)
                    best_vecino = copy.deepcopy(vecino)
                    break;
        if obj == best_obj:
```

```
                break;
    return(best_obj,best_vecino)

#Insertion con first improvement
def FIInsertion(S,instancia):
    best_obj = instancia.max_WjLj(S)
    best_vecino = copy.deepcopy(S)
    for j in range(instancia.jobs):
        for i in range(instancia.jobs):
            if j!= i and j!=i+1:
                vecino = copy.deepcopy(S)
                vecino.pop(j)
                vecino.insert(i,S[j])
                obj = instancia.SumLj(vecino)
                if obj<best_obj:
                    best_obj = copy.deepcopy(obj)
                    best_vecino = copy.deepcopy(vecino)
                    break;
        if obj == best_obj:
            break;
    return(best_obj,best_vecino)
```

crea_instancias_grandes

```
from scheptk.util import write_tag
import numpy as np
import random as rd

def generainstancia():
    for f in range(2,8):
        for num in range(1,2):
            with open("DPFSP/DPFSP_Large/" + str(f) + "/Ta00" + str(num)+ "_"
+str(f)+".txt",'r') as f_obj:
                archivo = f_obj.read()
            # print("Contenido del archivo:\n",archivo)
            #transformamos el archivo en una lista de cadenas
            lista = archivo.split()
            #transformamos esta lista en un lista de enteros
            datos = []
            for j in range(len(lista)):
                datos.append(int(lista[j]))
            # print(datos)
```

```
            jobs = datos[0]
            # print("El número de trabajos es", jobs)
            maq = datos[1]
            # print("El número de máquinas es", maq)
            fab = datos[2]
            # print("El número de fábricas es", fab)
            #leemos ahora los tiempos de proceso
            tpo = []
            i = 0
            for j in range(maq):
                i = 4 + i
                for k in range(jobs):
                    tpo.append(datos[i])
                    i = i + maq*2
                i = 2*(j+1)
            # print("Los tiempos de procesos son",tpo)

            #dividimos el vector tpo en el número de máquinas que tenemos
            pt = np.array(tpo)
            tiempos= np.split(pt, maq)

            #trasnformamos el vector tiempos a un formato admisible para nuestra forma
de leer instancias
            pij = [[0 for j in range(jobs)] for i in range(maq)]

            for i in range(maq):
                for j in range(jobs):
                    pij[i][j] = tiempos[i][j]
            # print("Los processing times son",pij)



            r = [0 for j in range(jobs)]
            w = [rd.randint(1,99) for j in range(jobs)]
            wf = rd.sample(range(10,100),fab)
            dd = [0 for j in range(jobs)]
            documento = 'instancia' + '_f' + str(fab) + '_j' + str(jobs) + '_m' +
str(maq) + '.txt'

            write_tag('JOBS', jobs, documento)
            write_tag('MACHINES', maq, documento)
            write_tag('F', fab, documento)
```

```
            write_tag('PT', pij, documento)
            write_tag('R', r, documento)
            write_tag('W', w, documento)
            write_tag('WF', wf, documento)
            write_tag('DD', dd, documento)
        for num in [11,21,31,41,51,61,71,81]:
            with open("DPFSP/DPFSP_Large/" + str(f) + "/Ta0" + str(num)+ "_"
+str(f)+".txt",'r') as f_obj:
                archivo = f_obj.read()
            # print("Contenido del archivo:\n",archivo)
            #transformamos el archivo en una lista de cadenas
            lista = archivo.split()
            #transformamos esta lista en un lista de enteros
            datos = []
            for j in range(len(lista)):
                datos.append(int(lista[j]))
            # print(datos)
            jobs = datos[0]
            # print("El número de trabajos es", jobs)
            maq = datos[1]
            # print("El número de máquinas es", maq)
            fab = datos[2]
            # print("El número de fábricas es", fab)
            #leemos ahora los tiempos de proceso
            tpo = []
            i = 0
            for j in range(maq):
                i = 4 + i
                for k in range(jobs):
                    tpo.append(datos[i])
                    i = i + maq*2
                i = 2*(j+1)
            # print("Los tiempos de procesos son",tpo)

            #dividimos el vector tpo en el número de máquinas que tenemos
            pt = np.array(tpo)
            tiempos= np.split(pt, maq)

            #trasnformamos el vector tiempos a un formato admisible para nuestra forma
de leer instancias
            pij = [[0 for j in range(jobs)] for i in range(maq)]
```

```
            for i in range(maq):
                for j in range(jobs):
                    pij[i][j] = tiempos[i][j]
            # print("Los processing times ",pij)



            r = [0 for j in range(jobs)]
            w = [rd.randint(1,99) for j in range(jobs)]
            wf = rd.sample(range(10,100),fab)
            dd = [0 for j in range(jobs)]
            documento = 'instancia' + '_f' + str(fab) + '_j' + str(jobs) + '_m' +
str(maq) + '.txt'

            write_tag('JOBS', jobs, documento)
            write_tag('MACHINES', maq, documento)
            write_tag('F', fab, documento)
            write_tag('PT', pij, documento)
            write_tag('R', r, documento)
            write_tag('W', w, documento)
            write_tag('WF', wf, documento)
            write_tag('DD', dd, documento)


generainstancia()
```

crea_instancias_pequeñas

```
from scheptk.util import write_tag
import numpy as np
import random


def generainstancia():
    for f in range(2,5):
        for trabajos in range(10,18,2):
            for maquinas in range(2,6):
                for num in range(1,2):
                    with
open("DPFSP/DPFSP_Small/"+str(f)+"/I_"+str(f)+"_"+str(trabajos)+"_"+str(maquinas)+"_"+st
r(num)+".txt",'r') as f_obj:
                        archivo = f_obj.read()
                    # print("Contenido del archivo:\n",archivo)
                    #transformamos el archivo en una lista de cadenas
                    lista = archivo.split()
```

```
                    #transformamos esta lista en un lista de enteros
                    datos = []
                    for j in range(len(lista)):
                        datos.append(int(lista[j]))
                    # print(datos)
                    jobs = datos[0]
                    # print("El número de trabajos es", jobs)
                    maq = datos[1]
                    # print("El número de máquinas es", maq)
                    fab = datos[2]
                    # print("El número de fábricas es", fab)
                    #leemos ahora los tiempos de proceso
                    tpo = []
                    i = 0
                    for j in range(maq):
                        i = 4 + i
                        for k in range(jobs):
                            tpo.append(datos[i])
                            i = i + maq*2
                        i = 2*(j+1)
                    # print("Los tiempos de procesos son",tpo)

                    #dividimos el vector tpo en el número de máquinas que tenemos
                    pt = np.array(tpo)
                    tiempos= np.split(pt, maq)

                    #trasnformamos el vector tiempos a un formato admisible para nuestra
forma de leer instancias
                    pij = [[0 for j in range(jobs)] for i in range(maq)]

                    for i in range(maq):
                        for j in range(jobs):
                            pij[i][j] = tiempos[i][j]
                    # print("Los processing times son",pij)



                    r = [0 for j in range(jobs)]
                    w = [random.randint(1,99) for j in range(jobs)]
                    wf = random.sample(range(10,100),fab)
                    dd = [0 for j in range(jobs)]
```

```
                    documento = 'instancia' + '_f' + str(fab) + '_j' + str(jobs) + '_m'
+ str(maq) + '.txt'

                    write_tag('JOBS', jobs, documento)
                    write_tag('MACHINES', maq, documento)
                    write_tag('F', fab, documento)
                    write_tag('PT', pij, documento)
                    write_tag('R', r, documento)
                    write_tag('W', w, documento)
                    write_tag('WF', wf, documento)
                    write_tag('DD', dd, documento)

generainstancia()
```