Trabajo Fin de Grado

Grado en Ingeniería de Tecnologías Industriales

Estudio de la filosofía Just-In-Time en un taller de flujo regular con la restricción de permutación

Autora: Lara van Wamelen Mariano

Tutor: Víctor Fernández-Viagas Escudero

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

Sevilla, 2023

Trabajo Fin de Grado
Grado en Ingeniería de Tecnologías Industriales

Estudio de la filosofía Just-In-Time en un taller de flujo regular con la restricción de permutación

Autora:
Lara van Wamelen Mariano

Tutor:
Víctor Fernández-Viagas Escudero
Profesor Titular

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

Trabajo Fin de Grado: Estudio de la filosofía Just-In-Time en un taller de flujo regular con la restricción de permutación

Autora: Lara van Wamelen Mariano

Tutor: Víctor Fernández-Viagas Escudero

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

Presidente:

Vocales:

Secretario:

Acuerdan otorgarle la calificación de:

Sevilla, 2023

El Secretario del Tribunal

Agradecimientos

Quiero expresar mi más sincero agradecimiento a todas aquellas personas que han sido fundamentales en la realización y culminación de este Trabajo de Fin de Grado.

En primer lugar, quiero agradecer a mi tutor, Víctor Fernández-Viagas Escudero, por su apoyo, tiempo y orientación a lo largo de todo este proceso. Gracias por su dedicación y compromiso, así como por su enseñanza que ha sido fundamental para la realización de este TFG.

También quiero mostrar mi agradecimiento a mis compañeros y amigos, quienes han sido una fuente inagotable de motivación y compañerismo a lo largo de estos años. Gracias por compartir conmigo buenos momentos y también algunos complicados, por estar siempre ahí para escucharme, animarme y ayudarme cuando más lo necesitaba.

No puedo dejar de mencionar a mi familia, en especial a mis padres, hermanos y seres queridos. Gracias por su amor incondicional, por creer en mí y por brindarme todo el apoyo y comprensión necesarios para lograr este objetivo. Su constante aliento y motivación han sido fundamentales en mi trayectoria académica. Sin ellos nada de esto habría sido posible.

A todos y cada uno de ustedes, mi más profundo agradecimiento por formar parte de esta etapa y por contribuir de alguna manera en la realización de este Trabajo de Fin de Grado.

Lara van Wamelen Mariano

Sevilla, 2023

Resumen

En este Trabajo de Fin de Grado se aborda un problema específico del ámbito de la programación de la producción. Se trata de un taller de flujo regular con restricción de permutación, conocido en la literatura como *Permutation FlowShop Scheduling Problem*. Para este problema específico, se estudiarán y compararán diferentes políticas de producción, algunas relacionadas con la implementación de una filosofía *Just-In-Time* y otras enfocadas en objetivos de producción más tradicionales.

Dado que la resolución de este problema es computacionalmente compleja debido a que forma parte de la clase NP-*hard*, se propone utilizar métodos aproximados, en particular, una metaheurística evolutiva, un algoritmo genético. Este tipo de métodos busca obtener soluciones cercanas al óptimo en un tiempo de cálculo razonable, aunque no necesariamente exactas. Se desarrollarán dos variantes del algoritmo genético con el fin de comparar su desempeño y determinar cuál de ellas ofrece mejores resultados.

En cuanto a la obtención de los resultados, se evaluará un amplio conjunto de 540 instancias con el objetivo de analizar la eficiencia y efectividad de la metodología propuesta. Además, se llevará a cabo tanto una comparativa entre ambas variantes del algoritmo genético como una entre las distintas políticas de producción mencionadas anteriormente. Por último, se expondrán las conclusiones obtenidas a lo largo del estudio y se propondrán nuevas líneas de investigación en este campo.

Abstract

In this Thesis, a specific problem in the field of production scheduling is addressed. It is a regular flow shop with permutation restriction, known in the literature as the Permutation FlowShop Scheduling Problem. For this specific problem, different production policies will be studied and compared, some related to implementing a Just-In-Time philosophy and others focused on more traditional production goals.

Since solving this problem is computationally complex as it belongs to the NP-hard class, approximate methods are proposed to be used, specifically an evolutionary metaheuristic, a genetic algorithm. This type of method aims to obtain solutions close to the optimum in a reasonable computation time, although not necessarily exact. Two versions of the genetic algorithm will be developed in order to compare their performance and determine which one offers better results.

Regarding the obtainment of results, a large set of 540 instances will be evaluated in order to analyze the efficiency and effectiveness of the proposed methodology. Additionally, a comparison will be carried out between both versions of the genetic algorithm as well as between the different production policies mentioned above. Finally, the conclusions obtained throughout the study will be presented and new research lines in this field will be proposed.

Índice

ÍNDICE DE TABLAS

ÍNDICE DE FIGURAS

1 INTRODUCCIÓN

El presente capítulo comprende una breve exposición de las razones que han llevado a la realización de este trabajo, así como una concisa explicación acerca de cuál es su propósito, ámbito y alcance. Asimismo, al final de este capítulo se detallan las diferentes partes en las que se estructura el contenido del documento.

1.1. Motivación y ámbito del trabajo

La programación de la producción es un área de estudio y aplicación de vital importancia en el ámbito de la ingeniería industrial y de la gestión de operaciones en una empresa. Este campo de investigación se enfoca en el diseño de herramientas y métodos para optimizar la asignación de recursos y la secuenciación de actividades en un entorno de producción, con el objetivo de maximizar la eficiencia y minimizar los costes.

En la actualidad, las empresas se enfrentan a un entorno cada vez más competitivo y dinámico, en el que se requiere una respuesta rápida y eficiente a la demanda del mercado. En esta situación, la programación de la producción se convierte en una tarea cada vez más compleja y necesaria, ya que implica la asignación óptima de todos los recursos disponibles de manera que se consiga satisfacer la demanda de los clientes con un alto nivel de calidad a la vez que se minimizan los costes de producción.

En este contexto, surge la filosofía *Just-In-Time* (JIT), la cual ha revolucionado la forma en que las empresas gestionan sus procesos productivos. Esta metodología se centra en minimizar el desperdicio, reducir los tiempos de producción y optimizar los recursos disponibles. Como se acaba de comentar, en esta era de competencia global, es crucial para las organizaciones implementar estrategias basadas en esta filosofía.

Una correcta programación de la producción proporciona una gran ventaja competitiva. Sin embargo, una mala puede ocasionar retrasos en la entrega de los productos y un gran número de despilfarros: exceso de stock, sobreproducción, tiempos de espera, sobre procesamiento, etc. Incluso se puede llegar a generar la incapacidad de satisfacer la demanda de los clientes, lo cual genera un impacto negativo en la rentabilidad, productividad y reputación de la empresa.

Resulta por tanto esencial contar con herramientas y técnicas que permitan una correcta planificación y control de la producción. Cabe destacar que existen una gran y diversa cantidad de métodos para llevar a cabo este proceso.

En primer lugar, se encuentran los métodos tradicionales, como el diagrama de Gantt y el método PERT, que se han utilizado durante décadas en la industria. Estos métodos, a pesar de permitir una visualización clara de las actividades a realizar, de su secuenciación, de la estimación de tiempos y los recursos necesarios, presentan limitaciones en entornos de producción más complejos y cambiantes, donde se requiere un mayor grado de flexibilidad y adaptabilidad.

Debido a esto, apareció la necesidad de desarrollar otras metodologías más avanzadas, como la programación lineal entera, las heurísticas o las metaheurísticas, las cuáles permiten abordar la programación de la producción de problemas más complejos.

Para la aplicación de estos métodos existe un amplio conjunto de herramientas de software disponibles en el mercado. Estas herramientas ofrecen a las empresas soluciones efectivas en tiempo real lo cual permite la toma de decisiones a corto plazo.

1.2. Objeto del trabajo

En este Trabajo de Fin de Grado, se abordará el análisis de un problema específico de programación de la producción: un taller de flujo regular (*FlowShop*) con restricción de permutación ($F_m \mid prmu \mid$). Para este caso en concreto se estudiarán y compararán diversas políticas de producción, algunas relacionadas con la implementación de una filosofía JIT y otras con objetivos de producción tradicionales tales como minimizar el *makespan*, etc.

Para la resolución del problema, debido a su complejidad computacional (*NP-hard*), se propone utilizar métodos aproximados, en concreto una metaheurística evolutiva, un algoritmo genético. Este tipo de métodos lo que buscan no son soluciones exactas sino conseguir soluciones próximas al óptimo en un tiempo de cálculo razonable. Se desarrollarán dos variantes del algoritmo genético con el propósito de comparar ambas y determinar cuál de ellas proporciona mejores resultados.

Respecto a la obtención de los resultados, se evaluará un amplio conjunto de instancias con la intención de poder estudiar la eficiencia y efectividad de la metodología propuesta, así como llevar a cabo la comparación y el análisis de las diferentes políticas de producción. Finalmente, se expondrán las conclusiones alcanzadas y se propondrán mejoras y recomendaciones para futuras investigaciones en este campo.

1.3. Sumario

Respecto a la estructura del presente documento, se distinguen los siguientes capítulos:

- Capítulo 1: Introducción

  En este primer capítulo se presentan brevemente las razones que han motivado la realización de este trabajo, así como se expone cuál es el objetivo, ámbito y alcance del mismo. También se detalla al final la estructura que sigue el documento, incluyendo un breve resumen del contenido de cada capítulo.

- Capítulo 2: Marco teórico

  En este segundo capítulo se sitúa y enmarca el desarrollo del trabajo dentro del área de la programación de la producción. Se presentan de forma general los principios teóricos y las ideas fundamentales requeridas para comprender adecuadamente el resto del trabajo. Asimismo, se ofrece una concisa descripción de qué es y en qué consiste la filosofía *Just-In-Time* (JIT).

- Capítulo 3: Caracterización del problema

  De manera ya más específica, en este capítulo se brinda una explicación detallada del problema concreto que se resolverá en este trabajo: *Permutation Flowshop Scheduling Problem* (PFSP). Se exponen también los indicadores desarrollados para evaluar y comparar las soluciones obtenidas, la codificación de la solución y la complejidad computacional del problema.

- Capítulo 4: Introducción a la metodología

  En primer lugar, este capítulo proporciona un resumen conciso y la clasificación de los diferentes métodos que existen a la hora de abordar problemas de programación de la producción, con el fin de brindar una perspectiva general. En segundo lugar, se explica la metodología específica empleada en este trabajo y la razón por la cual se ha elegido.

- Capítulo 5: Metodología

  Luego de haber explicado de manera general el funcionamiento de la metodología elegida, en este capítulo se detalla su implementación específica. La metodología seleccionada consiste en una metaheurística evolutiva, en concreto un algoritmo genético. Se proponen dos variantes diferentes con el objetivo de comparar ambas y finalmente utilizar aquella que ofrezca mejores soluciones. En este capítulo se analizan de forma exhaustiva todos los operadores genéticos y mecanismos que componen ambas variantes.

- Capítulo 6: Evaluación computacional

  En este último capítulo se exponen, analizan y comparan los resultados obtenidos. En primer lugar, se llevará a cabo una comparativa entre ambas variantes del algoritmo genético con el fin de determinar cuál de ellas proporciona mejores resultados. Posteriormente, se compararán cuatro políticas de producción distintas, dos tradicionales y dos relacionadas con la filosofía JIT, utilizando la variante que haya resultado ser mejor. Todos los resultados se obtienen evaluando un conjunto común de instancias que se expone también en este capítulo. Asimismo, se incluye una explicación de los indicadores de desempeño utilizados y se muestran los valores tomados para cada parámetro del algoritmo genético.

2 MARCO TEÓRICO

Con el propósito de contextualizar y enmarcar el trabajo dentro del ámbito de la programación de la producción, en el presente capítulo se introducen brevemente los fundamentos teóricos y los conceptos básicos necesarios para una adecuada comprensión del mismo. Asimismo, se proporciona una breve explicación de qué es y en qué consiste la filosofía *Just-In-Time*.

Este apartado permite establecer una base sólida de conocimientos y conceptos como punto de partida, sobre la cual se fundamenta el resto del trabajo.

2.1 Introducción y conceptos básicos de programación de la producción

Como se ha comentado anteriormente, la programación de la producción (*Manufacturing Scheduling*) aborda la asignación de recursos para la fabricación de un conjunto de productos, siendo el resultado un programa de producción (*Production Schedule*) que contiene información sobre cuándo debe comenzar cada recurso a procesar qué producto (Pérez-González et al., 2021).

Los recursos productivos disponibles se denominan máquinas (*machines*). Este concepto hace referencia no sólo a aparatos o herramientas, sino también a la mano de obra. Estos recursos son capaces de transformar una entrada o materia prima en una salida o producto, aportándole en el proceso un valor añadido. También se consideran recursos aquellos con capacidad para realizar operaciones de transporte de material.

Por otra parte, todos aquellos productos que son procesados en una máquina, es decir, aquellos a los que un recurso les realiza algún tipo de operación, se denominan trabajos (*jobs*). Entran dentro de este conjunto tanto materias primas como trabajos en proceso, etc.

Se suele emplear habitualmente la siguiente notación para representar estos dos conceptos: el conjunto de máquinas se denota como $M = \{1, ..., m\}$ mientras que el conjunto de trabajos como $N = \{1, ..., n\}$. Asimismo, los índices que se utilizan respetivamente son $i \in M$ y $j, k \in N$.

Como se ha comentado anteriormente, el resultado que se obtiene al llevar a cabo la programación de la producción es un programa. Es este el que asigna las fechas de comienzo y, si es necesario, de fin, de cada trabajo en cada máquina. Sin embargo, es una secuencia (*sequence*) la que proporciona el orden en que los diferentes trabajos son procesados en cada máquina. En general, a una secuencia le corresponden infinitos programas (Pérez-González et al., 2021). La notación que se suele emplear para representar una secuencia es $\pi := (\pi_1, \ldots, \pi_k, \ldots, \pi_n)$.

La representación más habitual de un programa de producción es un diagrama de Gantt, en el cual se visualiza de manera muy clara la asignación de cada trabajo en cada máquina, la duración de cada operación y sus instantes de inicio y fin.

Cabe destacar que el objetivo de toda programación de la producción es obtener un programa admisible (*Feasible Schedule*), es decir, que cumpla con todas las restricciones y características del entorno productivo. Para obtenerlo se hace uso de algoritmos: "conjunto de pasos ordenados y finitos que permiten resolver un problema o realizar una tarea específica" (Cairó Osvaldo, 2006).

En este documento se van a considerar únicamente programas semiactivos (*Semiactive Schedule*), los cuales se caracterizan por la imposibilidad de adelantar ninguna operación sin cambiar el orden en que alguna máquina procesa los trabajos (Pérez-González et al., 2021).

Además de los conceptos que se acaban de exponer, existen unos datos básicos cuyo conocimiento es necesario para una adecuada comprensión del problema que se abordará en los siguientes capítulos (Pinedo, 2016):

- Tiempo de proceso (*Processing time*) (p_{ij}): tiempo que se tarda en procesar el trabajo j en la máquina i. El subíndice i se omite si el tiempo de proceso del trabajo j no depende de la máquina o si dicho trabajo es procesado únicamente en una máquina.

- Fecha de llegada (*Release date*) (r_j): se refiere a la fecha en la que el trabajo j está disponible para empezar a ser procesado. Es el momento en el que el trabajo llega al sistema, es decir, el instante más temprano en el que el dicho trabajo puede iniciar su procesamiento.

- Fecha de entrega (*Due date*) (d_j): representa la fecha de entrega o finalización comprometida, es decir, la fecha que se le promete al cliente. Se permite la finalización de un trabajo después de su fecha de vencimiento, pero entonces se incurre en una penalización. Cuando el cumplimiento de este plazo es obligatorio, se denomina fecha límite y se indica con $\bar{d}_j$.

- Peso (*Weight*) (w_j): se trata de un factor de prioridad que denota la importancia del trabajo j en relación con los demás trabajos del sistema. Por ejemplo, este peso puede representar el coste de mantenimiento o de inventario del trabajo en cuestión.

2.2 Clasificación de los modelos de programación de la producción

El primer paso en la resolución de un problema de esta naturaleza es su proceso de modelado. Es imprescindible analizar detalladamente la situación real con el fin de poder identificar las variables relevantes y establecer relaciones entre ellas, permitiendo así la formulación de un modelo matemático que represente adecuadamente su comportamiento.

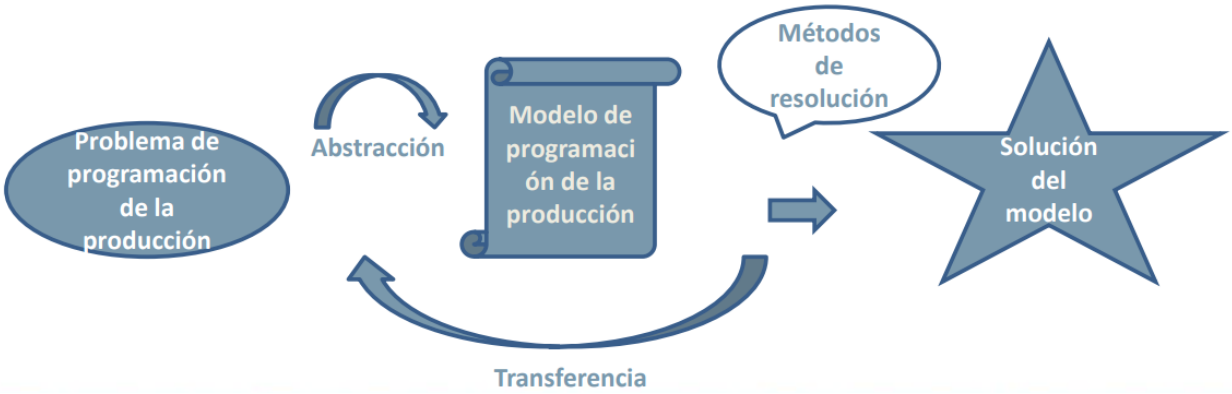


Figura 2.1. Proceso de modelado y resolución de un problema de programación de la producción [Fuente: Programación de operaciones. 4º del GITI. Universidad de Sevilla. Escuela Técnica Superior de Ingeniería]

Un modelo de programación de la producción (*Scheduling model*) consta de una serie de factores que lo definen. Estos incluyen el entorno en el que se sitúa el sistema productivo (α), las restricciones a las que este se encuentra sujeto (β) y los objetivos que se pretenden cumplir o alcanzar (γ).

Podemos concluir por tanto que un problema de programación de la producción queda definido por un triplete, un conjunto de tres parámetros: α | β | γ. A continuación, se procede al desarrollo de estos con un mayor grado de detalle.

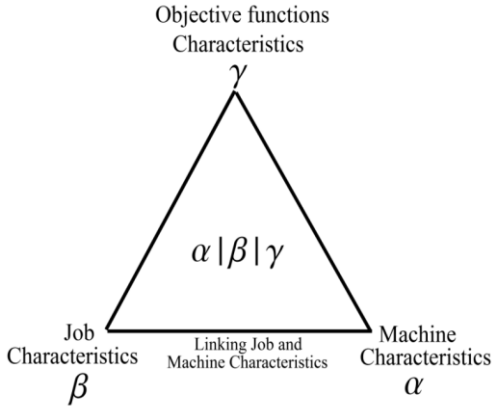


Figura 2.2. Factores que definen un modelo de programación de la producción [Fuente: Programación de operaciones. 4º del GITI. Universidad de Sevilla. Escuela Técnica Superior de Ingeniería]

2.2.1 Entornos (α)

Este apartado detalla el entorno de fabricación existente, denominado *layout*. Se encuentra sujeto a variables como el número de máquinas, su disposición y sus características. A continuación, se exponen de manera concisa algunos de los entornos más comunes.

- Una máquina (*Single Machine*) ($\alpha = 1$) → situación elemental que sirve como simplificación para entornos más complejos. Consta de un solo recurso en el cual se procesan todos los trabajos, una única vez cada uno, de manera que no existe ruta posible.

- Máquinas paralelas (*Parallel Machines*) → se trata de un entorno que evoluciona del anterior con el fin de abordar el caso de un aumento en la producción. En este sentido, se mantiene la existencia de una única etapa, no obstante, ahora se compone de diversas máquinas. Cada trabajo es sometido a procesamiento una sola vez en una de las máquinas, lo que implica que no se establece una ruta específica, al igual que en el anterior entorno. Existen tres casos diferentes en función de la velocidad y capacidad de procesamiento de las máquinas:

    - Máquinas idénticas (*Identical parallel machines*) ($\alpha = P_m$): las máquinas son exactamente iguales, tanto en términos de velocidad de procesamiento como de capacidad para realizar tareas. Esto significa que, para un trabajo dado, todas las máquinas tienen el mismo tiempo de procesamiento.
    - Máquinas uniformes (*Uniform parallel machines*) ($\alpha = Q_m$): las máquinas tienen la misma capacidad de procesamiento, pero diferentes velocidades.
    - Máquinas no relacionadas (*Unrelated parallel machines*) ($\alpha = R_m$): cada máquina tiene una velocidad de procesamiento y capacidad diferentes. Esto implica que el tiempo de procesamiento varía para cada trabajo en cada máquina.

Los siguientes entornos que se van a abordar son aquellos de tipo taller. En este tipo de entornos las máquinas se disponen en serie, realizando cada una de ellas una operación específica. Cada trabajo debe pasar por todas las máquinas siguiendo el orden establecido por su ruta de fabricación. Dependiendo del tipo de ruta, se distinguen tres categorías distintas de talleres:

- Taller de flujo regular (*FlowShop*) ($\alpha = F_m$) → la ruta de fabricación es la misma para todos los trabajos. Existe un caso particular denominado *Permutation FlowShop*, el cual se aborda en este trabajo y se expondrá con mayor nivel de detalle en el capítulo 3.

- Taller de trabajos (*JobShop*) ($\alpha = J_m$) → la ruta de fabricación es diferente para cada trabajo. Dichas rutas son un dato necesario para poder resolver el modelo de producción.

- Taller abierto (*OpenShop*) ($\alpha = O_m$) → es el entorno tipo taller más general y complejo ya que no existen rutas predeterminadas para cada trabajo si no que estos pueden seguir cualquier orden.

Por último, se encuentran los entornos híbridos (*Hybrid layout*). Estos se caracterizan por ser el entorno más general posible, conformados por un taller de cualquier tipo con múltiples etapas. Cada una de estas etapas puede estar compuesta a su vez por una sola máquina o por máquinas paralelas.

2.2.2 Restricciones (β)

Las limitaciones y restricciones a las que se encuentra sujeto el proceso de producción se detallan en este apartado. Este campo puede estar vacío, tener una sola entrada o varias en función de las características propias del sistema. Existe una amplia variedad de restricciones posibles, a continuación, en la Tabla 2.1, se muestran las más habituales (Pinedo, 2016).

Restricción	β	Definición
Fechas de llegada (*Release dates*)	r_j	El trabajo *j* no puede comenzar a ser procesado antes de su fecha de llegada r_j. Si esta restricción no aparece, el procesamiento del trabajo *j* puede empezar en cualquier momento.
Interrupción (*Preemption*)	*prmp*	Una vez iniciado el procesamiento de un trabajo en una máquina, no es necesario mantenerlo hasta su finalización. Está permitido interrumpir el procesamiento de cualquier trabajo en cualquier momento.
Precedencia (*Precedence*)	*prec*	Para que un trabajo pueda comenzar a ser procesado, todos los trabajos que lo preceden deben haber sido completados.
Tiempos de cambio (*Set-up times*)	s_{jk}	s_{jk} representa el tiempo de configuración en el que se incurre al pasar de procesar el trabajo *j* a procesar el trabajo *k*. Si este tiempo depende también de la máquina, entonces se incluye el subíndice *i*, es decir, s_{ijk}. En caso de que esta restricción no aparezca, se suponen todos los tiempos de cambio despreciables.
Lotes (*Batching*)	*batch (b)*	Una máquina es capaz de procesar simultáneamente, como mucho, un número *b* de trabajos. Es decir, puede procesar un lote de hasta *b* trabajos al mismo tiempo. Existen lotes en paralelo (*Parallel batching*) y en serie (*Serial batching*).
Permutación (*Permutation*)	*prmu*	Limitación que puede aparecer en un entorno de tipo taller de flujo regular (*FlowShop*). En esta situación, todas las máquinas procesan los trabajos en el mismo orden.
Bloqueo (*Blocking*)	*block*	Fenómeno que ocurre cuando existe un buffer limitado entre dos máquinas, de manera que, si dicho buffer esta completo, la máquina aguas arriba no puede finalizar un trabajo. Dicho trabajo debe mantenerse en la máquina hasta que haya espacio suficiente en el buffer, obligándola a no procesar ningún otro trabajo mientras tanto.

No tiempo de espera (*No-wait*)	*no-wait*	Los trabajos no pueden esperar entre máquina y máquina. Esto implica que, si resulta necesario, se debe retrasar el comienzo de un trabajo en la primera máquina para garantizar que este pueda recorrer el flujo de producción sin tener que esperar a ninguna máquina para ser procesado.
No tiempo ocioso (*No-idle*)	*no-idle*	Las máquinas no pueden estar paradas entre trabajo y trabajo, es decir, una vez una máquina comience a procesar el primer trabajo no puede parar hasta haber finalizado el último.

Tabla 2.1. Restricciones habituales en un modelo de programación de la producción

En cuanto a las fechas de entrega, no suelen especificarse en este campo a menos que sean de obligado cumplimiento, en cuyo caso se denominan *deadlines* y toman la notación $\beta = \bar{d}_j$.

Otros casos concretos que se especifican en este campo son aquellos en los que todos los tiempos de proceso o todas las fechas de entrega son iguales, siendo la notación, respectivamente, $\beta = p_j = p$ y $\beta = d_j = d$.

2.2.3 Objetivos (γ)

Este último campo detalla cuál es el objetivo que se persigue a la hora de resolver el problema de producción y alcanzar una solución. Este objetivo viene definido por una función que se pretende minimizar o maximizar y la cual depende siempre de los tiempos de terminación de los trabajos, los cuales dependen a su vez del programa de producción. A veces, la función objetivo está relacionada también con las fechas de entrega.

Algunas de las medidas más comunes que se utilizan para evaluar el rendimiento de los diferentes programas obtenidos son las siguientes:

- Tiempo de terminación (*Completion time*) (C_{ij}) →instante en el que el trabajo j finaliza su procesamiento en la máquina i. El instante en que el trabajo j abandona el sistema se corresponde con el tiempo de terminación de dicho trabajo en la última máquina en la que debe ser procesado y se denota como C_j.

- Tiempo de flujo (*Flowtime*) (F_j) → periodo de tiempo que el trabajo j está en el sistema, es decir, desde su fecha de llegada hasta que se completa en la última máquina.

$$F_j = C_j - r_j \tag{2–1}$$

- Retraso (*Lateness*) (L_j) → tiempo que transcurre entre la fecha de entrega del trabajo j y su instante de finalización. Toma un valor positivo cuando el trabajo j se completa tarde, es decir, después de su fecha de entrega, y negativo en el caso contrario.

$$L_j = C_j - d_j \tag{2–2}$$

- Tardanza (*Tardiness*) (T_j) → tiempo que lleva de retraso el trabajo j. La diferencia con la medida anterior es que, si el trabajo j se entrega antes de tiempo, T_j toma un valor de cero, de manera que nunca alcanza valores negativos.

$$T_j = \text{máx}\{0, L_j\} = \text{máx}\{0, C_j - d_j\} \tag{2–3}$$

- Adelanto (*Earliness*) (E_j) → tiempo que lleva de adelanto el trabajo j. Al contrario que ocurre con la tardanza, E_j toma un valor de cero cuando el trabajo j se completa después de su fecha de entrega. Esta medida tampoco alcanza nunca valores negativos.

$$E_j = \text{máx}\{0, -L_j\} = \text{máx}\{0, d_j - C_j\} \tag{2–4}$$

- Trabajo tarde (*Tardy job*) (U_j) → medida binaria que toma el valor 1 cuando el trabajo j se entrega tarde y cero en caso contrario.

$$U_j = \begin{cases} 1 \; si \; C_j > d_j \; (T_j > 0) \\ 0 \; en \; caso \; contrario \end{cases} \tag{2–5}$$

- Trabajo pronto (*Early job*) (V_j) → medida binaria que toma el valor 1 cuando el trabajo j se completa antes de su fecha de entrega y cero en caso contrario.

$$V_j = \begin{cases} 1 \; si \; C_j < d_j \; (E_j > 0) \\ 0 \; en \; caso \; contrario \end{cases} \tag{2–6}$$

- Trabajo justo a tiempo (*JIT job*) (W_j) → medida binaria que toma el valor 1 cuando el trabajo j se completa justo en su fecha de entrega, ni antes ni después, y cero en caso contrario.

$$W_j = \begin{cases} 1 \; si \; C_j = d_j \; (T_j, E_j = 0) \\ 0 \; en \; caso \; contrario \end{cases} \tag{2–7}$$

Con la intención de tener en cuenta todos los trabajos, estas medidas se pueden incluir en la función objetivo tanto en forma de sumatorio (p.e. $\sum_{j=1}^{n} C_j$) como en forma de máximo (p.e. $\max_{1 \le j \le n} C_j$).

Si, además, cada trabajo tiene una importancia diferente, las medidas se suelen ponderar utilizando una variable denominada peso (*weight*) (w_j). Esta variable toma valores en función de la prioridad del trabajo j, de manera que, cuanto mayor sea w_j más importante es el trabajo j, siempre en relación con el resto.

Más adelante, en el capítulo 3, se explicarán en profundidad los objetivos considerados en este trabajo a la hora de resolver el problema planteado.

2.3 Filosofía *Just-In-Time* (JIT)

La filosofía *Just-in-Time* (JIT) fue presentada por primera vez por la compañía japonesa del sector automovilístico Toyota en los años 50. Esta filosofía puede ser definida brevemente como la eliminación del desperdicio y la mejora continua de la productividad (Józefowska Joanna, 2007). En un sistema productivo existe una gran variedad de fuentes de desperdicio por lo que, para implementar de manera efectiva esta filosofía, resulta necesario llevar a cabo muchos cambios y actividades.

El término desperdicio se refiere a cualquier actividad que no proporciona un valor añadido al producto y por la cual el cliente no está dispuesto a pagar. Dentro del *Lean Manufacturing* se distinguen 8 desperdicios: defectos, sobreprocesamiento, sobreproducción, tiempos de espera, inventario, transporte, movimientos innecesarios y desperdicio del talento humano. Entre todos ellos, los tiempos de espera, la sobreproducción y el inventario pueden ser reducidos e incluso eliminados mediante una correcta planificación y programación de la producción.

Figura 2.3. Los 8 despilfarros del *Lean Manufacturing* [Fuente: https://artedalogistica.blogspot.com/2018/08/logistica-lean-8-desperdicios-serem.html]

Los objetivos de la programación JIT difieren de los objetivos considerados en la programación tradicional de producción. A la hora de aplicar políticas de producción basadas en esta filosofía se consideran dos objetivos diferentes de optimización. El primero consiste en la minimización de la variación de producción, lo que significa que se debe producir la misma cantidad de cualquier producto todos los días. El segundo objetivo, y en el cual se centra este trabajo, busca minimizar tanto la finalización temprana como la tardía de los trabajos (Ríos Roger et al., 2012). El propósito de realizar esta minimización es reducir el inventario y, al mismo tiempo, satisfacer la demanda de los clientes con la entrega a tiempo de los productos. Además, de esta manera se consigue un flujo continuo de producción, reduciendo e incluso eliminando los tiempos de espera en el sistema.

Indudablemente, finalizar un pedido pasada su fecha de entrega no es deseable. En el caso de los productos terminados, completarlos tarde no solo genera la insatisfacción del cliente, sino que también puede conllevar la pérdida del pedido o incluso del cliente en sí. Por lo general, la finalización tardía de un trabajo incurre en costes adicionales como una penalización económica, el tiempo invertido en tratar con el cliente, el gasto de acelerar las tareas atrasadas y el coste de reajustar el programa de producción.

Por otro lado, los trabajos terminados antes de su fecha de entrega deben almacenarse, creando de esta manera un inventario no deseado, uno de los principales despilfarros que busca eliminar la filosofía JIT. El coste en el que se incurre depende de la cantidad de productos almacenados, del valor de estos artículos y del tiempo que se mantengan. Este coste incluye tanto el coste de almacenamiento como el coste de mantenimiento, así como otros costes como por ejemplo la obsolescencia de los artículos acumulados (Józefowska Joanna, 2007).

En los sistemas tradicionales de planificación y control de producción los objetivos de programación (campo γ) considerados están generalmente relacionados directamente con medidas como el tiempo de flujo (*Flowtime*) o los tiempos de terminación (*Completion times*). Estas políticas de producción no tienen en cuenta desperdicios como la sobreproducción o el inventario, sino que intentan generar el mayor número de trabajos posible con los recursos disponibles y en el mínimo tiempo. Se centran en la producción rápida de grandes cantidades de productos para poder satisfacer la demanda de los clientes, aunque esto incurra en costes adicionales. Debido a esto, este tipo de políticas se aplican generalmente a sistemas de fabricación contra stock (MTS: *Make To Stock*).

Por otro lado, el enfoque JIT, como ya se ha comentado anteriormente, lo que busca es completar los pedidos lo más cerca posible de su fecha de vencimiento (*Due date*), ni antes ni después. Como resultado, surge una nueva clase de problemas de programación en los cuales se pueden encontrar tanto objetivos relacionados con el tiempo en sí de retraso y/o adelanto de los trabajos (*Tardiness* y *Earliness*) como objetivos relacionados con el número de trabajos completados a tiempo, tarde y/o pronto (*Number of JIT, Tardy* y *Early Jobs*). Puesto que las políticas de producción JIT se centran en la satisfacción del cliente, estas se adaptan muy bien a sistemas de fabricación contra pedido (MTO: *Make To Order*) donde las fechas de entrega desempeñan un papel fundamental.

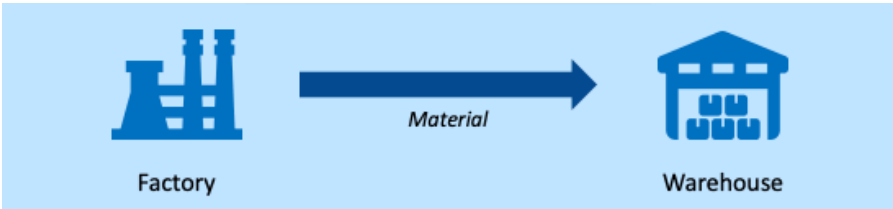


Figura 2.4. Sistema de fabricación contra stock (MTS) [Fuente: https://www.sketchbubble.com/en/presentation-make-to-stock.html]

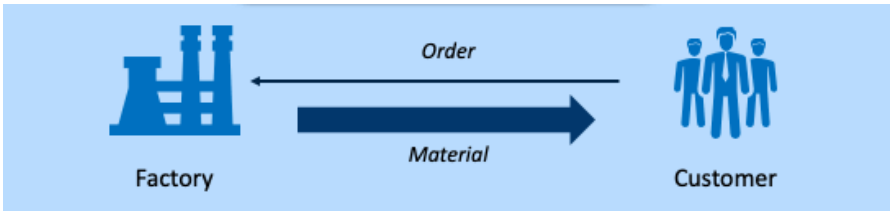


Figura 2.5. Sistema de fabricación contra pedido (MTO) [Fuente: https://www.sketchbubble.com/en/presentation-make-to-stock.html]

3 CARACTERIZACIÓN DEL PROBLEMA

En el presente capítulo se describe de manera detallada el problema concreto que se va a resolver en este trabajo: $F_m \mid prmu \mid \gamma$. En el último campo no se concreta ningún objetivo ya que el propósito de este trabajo es analizar varios de ellos con la intención de poder comparar políticas de producción tradicionales con aquellas basadas en la filosofía JIT. Cada una de las funciones objetivo estudiadas se detallan también en este capítulo.

Asimismo, se exponen los indicadores desarrollados para el análisis de los resultados, la codificación de la solución y la complejidad computacional del problema.

3.1 Descripción del problema: entorno, restricciones y objetivos

El problema de programación de la producción abordado en este trabajo consiste en un taller de flujo regular con restricción de permutación. Se denomina *Permutation Flowshop Scheduling Problem*, de ahora en adelante, PFSP y la notación correspondiente es $F_m \mid prmu \mid$.

Este tipo de problema es uno de los más estudiados en el área de Investigación Operativa debido a varias razones (Pérez-González et al., 2021). Por un lado, el taller de flujo regular es una de las configuraciones más comunes en sistemas productivos reales, ya que presenta muchas ventajas frente a otros entornos como los talleres de trabajos o los talleres abiertos. Por otra parte, se ha demostrado que los procedimientos de resolución desarrollados para este problema en concreto resultan muy eficientes a la hora de resolver problemas similares.

En primer lugar, en relación con el campo α, un taller de flujo regular se caracteriza por el hecho de que todos los trabajos siguen la misma ruta de fabricación, es decir, el orden en el que visitan las diferentes máquinas es el mismo para todos los trabajos, tal y como se muestra en la Figura 3.1.

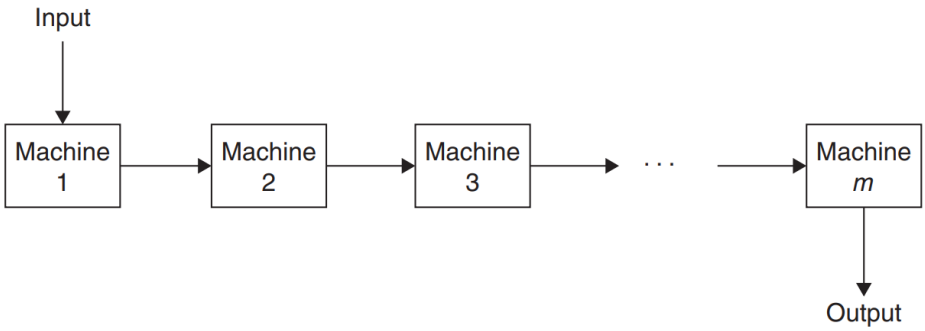


Figura 3.1. Flujo de producción en un taller de flujo regular (*Flowshop)* [Fuente: Baker, & Trietsch, D. (2019). Principles of sequencing and scheduling]

En segundo lugar, la única restricción existente en el campo β es la de permutación, lo cual indica que todas las máquinas procesan los trabajos en el mismo orden.

Cabe destacar que, debido a que nuestro problema en concreto no está sujeto a otras limitaciones, se asumen las siguientes suposiciones (Pérez-González et al., 2021):

- Los trabajos están disponibles desde el primer instante del horizonte de programación, es decir, las fechas de llegada (r_j) se suponen cero.
- Los trabajos no se pueden interrumpir.
- Máquinas siempre disponibles, es decir, no se les realiza mantenimiento preventivo ni correctivo.
- Cada máquina puede hacer un trabajo, y un trabajo puede ser realizado sólo en una máquina.
- El buffer entre máquinas se supone infinito, de manera que nunca aparece situación de bloqueo.
- El tiempo de transporte de los trabajos entre una máquina y otra es despreciable.

En la Figura 3.2 se muestra un diagrama de Gantt asociado a un problema de tipo PFSP, en el cual se puede observar la restricción de permutación, así como el resto de las hipótesis asumidas.

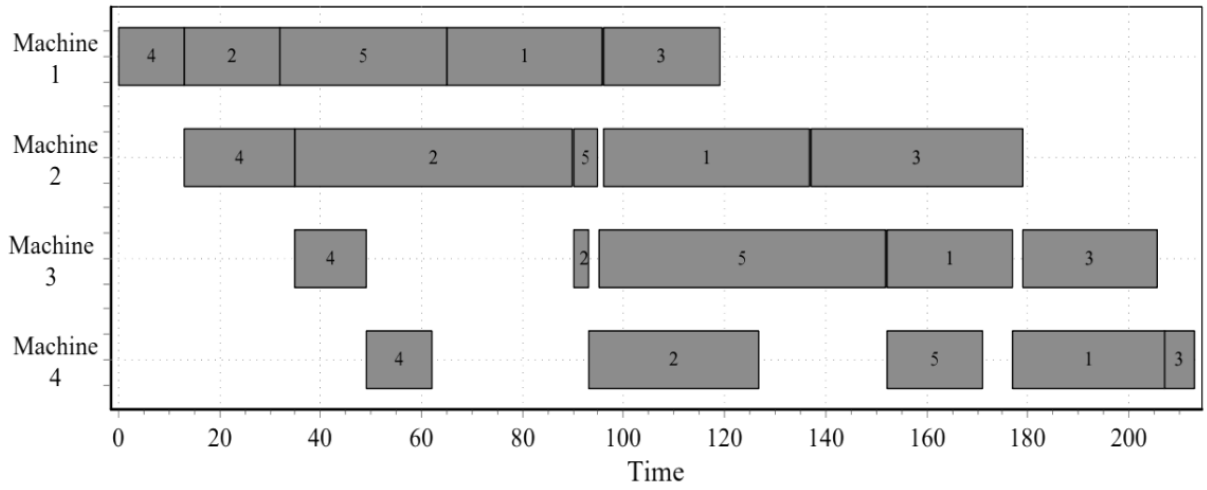


Figura 3.2. Diagrama de Gantt de un PFSP [Fuente: Programación de operaciones. 4º del GITI. Universidad de Sevilla. Escuela Técnica Superior de Ingeniería]

Por último, en relación con el campo γ, se describen en la Tabla 3.1 y en la Tabla 3.2 los diferentes objetivos considerados a la hora de resolver el problema y/o evaluar las soluciones alcanzadas. Mientras que los objetivos relacionados con políticas de producción tradicionales se exponen en la Tabla 3.1, aquellos basados en la filosofía JIT se muestran en la Tabla 3.2.

MINIMIZAR/MAXIMIZAR	γ	Objetivo
MIN	$C_{máx} = \max_{1 \le j \le n} C_j$	Makespan (Maximum Completion Time)
MIN	$\sum_{j=1}^{n} C_j$	Total Completion Time

Tabla 3.1. Objetivos de producción tradicionales

MINIMIZAR/MAXIMIZAR	γ	Objetivo
MIN	$\sum_{j=1}^{n} T_j$	Total Tardiness
MIN	$\sum_{j=1}^{n} E_j$	Total Earliness
MIN	$\sum_{j=1}^{n} T_j + E_j$	Total Tardiness + Earliness
MIN	$\sum_{j=1}^{n} U_j$	Number of Tardy Jobs
MIN	$\sum_{j=1}^{n} V_j$	Number of Early Jobs
MIN	$\sum_{j=1}^{n} U_j + V_j$	Number of Tardy + Early Jobs
MAX	$\sum_{j=1}^{n} W_j$	Number of JIT Jobs

Tabla 3.2. Objetivos de producción JIT

Con la intención de poder comparar las diferentes políticas de producción, se han analizado tanto objetivos relacionados con sistemas de producción tradicionales como objetivos relacionados con sistemas de producción JIT. Cabe destacar que, para la mayoría de las funciones objetivo lo que se pretende es minimizar excepto para el caso del *Number of JIT Jobs*, que se busca maximizar ya que se refiere al número de trabajos completados justo en su fecha de entrega.

Resulta conveniente aclarar también que, en las funciones objetivo que buscan minimizar el número de trabajos completados tarde o pronto, un trabajo se penaliza igualmente independientemente de la duración de su tardanza o su adelanto (Ríos Roger et al., 2012).

Asimismo, resaltar que algunos de los objetivos basados en la filosofía JIT dan lugar a medidas de rendimiento no regulares, como es el caso del adelanto. Las medidas no regulares se caracterizan por ser funciones decrecientes con los tiempos de terminación y plantean nuevos problemas metodológicos en el diseño de algoritmos de programación (Józefowska Joanna, 2007).

En relación con la notación adoptada, basada en el triplete α | β | γ, los problemas objeto de estudio en este documento se denotan como:

- $F_m \mid prmu \mid C_{máx}$
- $F_m \mid prmu \mid \sum C_j$
- $F_m \mid prmu \mid \sum T_j + E_j$
- $F_m \mid prmu \mid \sum U_j + V_j$

A pesar de considerar los valores obtenidos en todos los objetivos comentados con anterioridad a la hora de evaluar y comparar las soluciones obtenidas, sólo se va a resolver el problema para estas 4 funciones objetivo, es decir, para 4 políticas de producción. En concreto, se ha decidido escoger dos políticas de producción tradicional y otras dos basadas en la filosofía JIT. Respecto a estas últimas, se han elegido aquellas que representan mejor esta filosofía ya que tienen en cuenta tanto el adelanto como el retraso de los trabajos.

3.2 Indicadores

Además de todos los objetivos, se han desarrollado una serie de indicadores para evaluar los diferentes programas de producción obtenidos:

- Tiempo ocioso (*Idle time*) (I_{time}) → tiempo total que las máquinas del taller están paradas entre trabajo y trabajo, es decir, sin llevar a cabo ninguna operación. Una máquina se encuentra en tiempo ocioso cada vez que finaliza de procesar un trabajo antes que la máquina aguas arriba ha finalizado de procesar el siguiente.

$$I_{time} = \sum_{i=1}^{m}\sum_{k=1}^{n} \max\{0, C_{i-1,\pi_k} - C_{i,\pi_{k-1}}\} \quad siendo\ C_{i0} = C_{0j} = 0 \tag{3–1}$$

  Donde π_k es el trabajo que está en la posición *k* de la secuencia.

- Tiempo de espera (*Waiting time*) (W_{time}) → tiempo total que los trabajos están esperando a ser procesados entre máquina y máquina, es decir, tiempo que están en los buffers. Se denomina buffer al espacio donde esperan los trabajos para ser procesados, cada máquina tiene el suyo y suele posicionarse delante de esta. Un trabajo se encuentra en tiempo de espera cada vez que finaliza en una máquina, pero no puede entrar en la siguiente ya que esta está ocupada, es decir, todavía sigue procesando el trabajo anterior.

$$W_{time} = \sum_{i=1}^{m}\sum_{k=1}^{n} \max\{0, C_{i,\pi_{k-1}} - C_{i-1,\pi_k}\} \quad siendo\ C_{i0} = C_{0j} = 0 \tag{3–2}$$

- Tiempo medio en inventario (*Stock average time*) (S_{time}) → se trata de una medida porcentual que expresa el porcentaje de tiempo que están de media los trabajos en inventario en proceso (WIP).
  Un trabajo está en inventario cuando está esperando a ser procesado por una máquina, es decir, cuando está en un buffer. El tiempo en inventario de un trabajo *j* se calcula por tanto como el tiempo de espera de dicho trabajo dividido entre el tiempo que el trabajo está en el sistema. Realizando la media para todos los trabajos se obtiene el tiempo medio en inventario S_{time}.

$$S_{time} = \sum_{k=1}^{n} \frac{S_{time,\pi_k}}{n} \tag{3–3}$$

$$\text{Donde:} \quad S_{time,\pi_k} = \frac{\sum_{i=1}^{m} \max\{0, C_{i,\pi_{k-1}} - C_{i-1,\pi_k}\}}{C_{\pi_k} - (C_{1,\pi_k} - p_{1,\pi_k})} \quad siendo\ C_{i0} = C_{0j} = 0 \tag{3–4}$$

Cabe destacar que estos indicadores se calculan a partir del instante en el que el primer trabajo del programa comienza a ser procesado en la primera máquina, y hasta que el último trabajo se completa en la última máquina.

3.3 Codificación de la solución

Respecto a la codificación de la solución, en primer lugar, resaltar que un problema de taller de flujo regular sin la restricción de permutación tiene un total de $(n!)^m$ programas semiactivos. En este caso es necesario proporcionar una secuencia para cada máquina del taller (Pérez-González et al., 2021).

Sin embargo, para el PFSP el número de programas semiactivos posibles se reduce a $n!$ La solución que se busca determinar en este caso es una secuencia de *n* trabajos, la misma para todas las máquinas, que proporciona un menor valor de la función objetivo. Esta secuencia indica el orden en el que los trabajos deben ser procesados en el taller.

Con esta representación de la solución resulta fácil construir un programa semiactivo secuenciando el primer trabajo en todas las *m* máquinas, luego el segundo, y así sucesivamente hasta que todos los *n* trabajos estén programados (Vallada Eva et al., 2010).

3.4 Complejidad computacional del problema

Los problemas de programación se clasifican en dos categorías principales según su complejidad computacional:

- Polinomial (P) → problemas en los que el tiempo de ejecución está acotado por una función polinómica en función del tamaño de los datos de entrada. Esto significa que, a medida que aumenta el tamaño del problema, el tiempo de ejecución también aumenta, pero de manera controlada. Este tipo de problemas se consideran eficientes y pueden resolverse en un tiempo razonable.

- No polinomial (NP) → problemas en los que el tiempo de ejecución no puede ser acotado por una función polinómica en función del tamaño de los datos de entrada. Esto significa que, a medida que se incrementa el tamaño del problema, el tiempo necesario para resolverlo aumenta rápidamente. Estos problemas son considerados computacionalmente difíciles.

Dentro de los NP, existe una clase de problemas denominados NP-completos. Estos son los problemas más difíciles de la categoría y se caracterizan por el hecho de que, encontrar un algoritmo de tiempo polinomial capaz de resolver uno de ellos permitiría automáticamente resolver todos los problemas NP-completos. Este tipo de algoritmo aún no existe (Lopez Pierre et al., 2008).

Aquellos problemas de optimización igual, o incluso más difíciles que los NP-completos se denominan NP-hard. Estos problemas se dividen a su vez en dos clases: NP-hard en sentido fuerte (*strong sense*) y NP-hard en sentido débil (*ordinary sense*) (Baker Kenneth et al., 2019). Para los problemas en sentido débil existen algoritmos que permiten alcanzar soluciones óptimas en un tiempo pseudo-polinomial. Sin embargo, los de sentido fuerte resultan muy complicados, incluso imposibles, de resolver de manera exacta.

El problema que se resuelve en este trabajo, el PFSP, excepto algunos casos específicos, pertenece a la categoría NP-hard en sentido fuerte (Lopez Pierre et al., 2008).

Esto implica que no es posible, de manera general, alcanzar la solución óptima del problema en un tiempo razonable. Si se quiere resolver de manera exacta, sería necesario evaluar los $n!$ programas semiactivos posibles.

Debido a su complejidad computacional, suele ser mejor aplicar métodos aproximados, como las heurísticas o metaheurísticas, para resolver el problema ya que permiten alcanzar una solución cercana al óptimo en un tiempo computacional razonable. Generalmente cuanto mayor sea el tiempo de cómputo, mejores soluciones se obtendrán.

En el siguiente capítulo se describirán de manera detallada las diferentes metodologías existentes a la hora de resolver problemas de programación de la producción.

4 INTRODUCCIÓN A LA METODOLOGÍA

El presente capítulo comprende un breve resumen de las metodologías generales que existen para resolver los problemas de programación de la producción con el propósito de proporcionar una primera visión general. Posteriormente se describe el método específico seleccionado en este trabajo para resolver el problema propuesto teniendo en cuenta su complejidad computacional, incluyendo su diagrama de flujo y pseudocódigo.

4.1 Introducción a los métodos generales de resolución

El campo de la programación de la producción ha sido ampliamente investigado durante más de medio siglo. Su estudio ha captado la atención de especialistas en gestión, ingeniería industrial, investigación operativa e informática. A lo largo de este tiempo, se ha demostrado que conseguir una correcta planificación supone una tarea enormemente complicada (Jarboui Bassem et al., 2013). Por consiguiente, la necesidad de desarrollar metodologías que resuelvan eficazmente este tipo de problemas se ha ido haciendo cada vez más latente.

Existen diferentes métodos empleados para la resolución de problemas de programación de la producción. La primera clasificación que se puede llevar a cabo es (Pérez-González et al., 2021):

- **Métodos exactos**: aquellos que proporcionan la solución óptima del problema. Estos métodos tienen a menudo un uso limitado ya que, a pesar de ser capaces de resolver ciertos problemas específicos e instancias pequeñas, existen muchos otros problemas para los que el tiempo de cómputo resulta demasiado elevado. Dentro de esta categoría se distinguen a su vez: Los métodos aproximados se dividen a su vez en:

    - Algoritmos constructivos exactos: algoritmos que construyen una solución paso a paso, siguiendo una estrategia determinada, hasta que consiguen alcanzar la solución óptima. Se trata de algoritmos de tiempo polinomial capaces de resolver únicamente ciertos problemas específicos de manera exacta.

      A diferencia de otros métodos en los que se exploran todas las soluciones posibles y se selecciona la mejor, los algoritmos constructivos exactos se dedican a construir la solución de manera incremental, de tal forma que, en cada paso van tomando decisiones que garantizan la mejora continua de la función objetivo.

      Para lograr esto, se comienza con una solución inicial vacía, o parcialmente construida, y en cada paso del algoritmo se le añade un elemento que maximice o minimice la función objetivo. Esto se repite hasta que se hayan agregado todos los elementos, completando así la solución final.

      Algunos de los algoritmos que pertenecen a esta clase junto con los problemas específicos que son capaces de resolver de manera exacta se muestran en la Tabla 4.1.

Método de resolución	Obtiene óptimo para:
Regla FIFO	$1 \mid r_j \mid C_{máx}$
Regla SPT	$1 \mid\mid \sum C_j$ $P_m \mid\mid \sum C_j$
Regla EDD	$1 \mid\mid máx\ T_j$ $1 \mid\mid máx\ L_j$
Regla WSPT	$1 \mid\mid \sum w_j C_j$
Regla SRPT-FM	$Q_m \mid pmtn \mid \sum C_j$
Algoritmo de Johnson	$F_2 \mid\mid C_{máx}$
Algoritmo de Lawler	$1 \mid prec \mid máx\ g(C_j)$
Algoritmo de Moore	$1 \mid\mid \sum U_j$

Tabla 4.1. Algunos algoritmos constructivos exactos junto con el problema específico para el cual obtienen el óptimo

- Algoritmos enumerativos: algoritmos que exploran implícita o explícitamente todas las posibles soluciones del problema y eligen finalmente la óptima, es decir, aquella que proporcione un mejor valor de la función objetivo. Se trata de algoritmos de tiempo no polinomial capaces de proporcionar soluciones exactas únicamente para instancias pequeñas, es decir, para problemas de tamaño pequeño, en un tiempo razonable.

  Pueden ser utilizados en diferentes contextos y adaptados según las necesidades específicas del problema a resolver.

  Pertenecen a esta categoría la programación matemática, en particular la programación lineal entera mixta (MILP), el *Branch and Bound* (B&B) y la programación dinámica.

- **Métodos aproximados**: aparecen debido al uso limitado de los métodos exactos. Suelen emplearse para resolver problemas complejos, especialmente en el caso de instancias grandes. Se trata de métodos que, aunque no garanticen la optimalidad, son capaces de ofrecer buenas soluciones en un tiempo computacional razonable. Los métodos aproximados se dividen a su vez en:

  - Heurísticas: procedimientos prácticos que se utilizan para encontrar soluciones aproximadas a problemas complejos de manera rápida y eficiente. Estas técnicas se basan en la intuición y la experiencia previa, y a menudo implican el uso de reglas o estrategias simplificadas.

    Según su modo de operar, se distinguen dos tipos de heurísticas. Por un lado, están las heurísticas constructivas, que se caracterizan por construir una solución desde cero paso a paso, agregando elementos de manera incremental. Por otro lado, están las heurísticas de mejora, que se utilizan para mejorar una solución ya existente a través de iteraciones, modificándola gradualmente.

A pesar de todo, las heurísticas presentan dos inconvenientes significativos. En primer lugar, están diseñadas para un problema específico y resulta difícil, si no imposible, adaptarlas a otros problemas (Jarboui Bassem et al., 2013). En segundo lugar, tienen el riesgo de quedar atrapadas en mínimos locales. Cuando esto ocurre, la heurística no es capaz de escapar para explorar otras regiones del espacio de búsqueda y converge hacia una solución subóptima. Para superar estas limitaciones, se han desarrollado las metaheurísticas.

- Metaheurísticas: el concepto de metaheurísticas alude a heurísticas generales que tienen la capacidad de adaptarse fácilmente a una amplia variedad de problemas de optimización. Además de esta versatilidad y su relativamente sencilla implementación, la ventaja principal de las metaheurísticas es que tienen la habilidad de evitar quedar atrapadas en un óptimo local de la función objetivo y, por lo tanto, se aproximan a un óptimo global.

  Dentro de las diversas metaheurísticas que se han ido desarrollando, destacan dos tipos principales de métodos (Jarboui Bassem et al., 2013):

  - Métodos de búsqueda local: se trata de métodos que, partiendo de una solución existente, exploran sus vecinos en busca de una nueva solución que proporcione un mejor valor de la función objetivo. Algunas de las metaheurísticas que pertenecen a esta categoría son el recocido simulado y la búsqueda tabú.

  - Métodos evolutivos: son métodos inspirados en la teoría de la evolución biológica que simulan procesos como la reproducción, mutación y selección natural. En este tipo de métodos se busca que una "población" de soluciones vaya evolucionando y mejorando en cada iteración. Uno de los algoritmos más conocidos dentro de esta categoría es el algoritmo genético, aunque también existen otros como el algoritmo de colonia de hormigas.

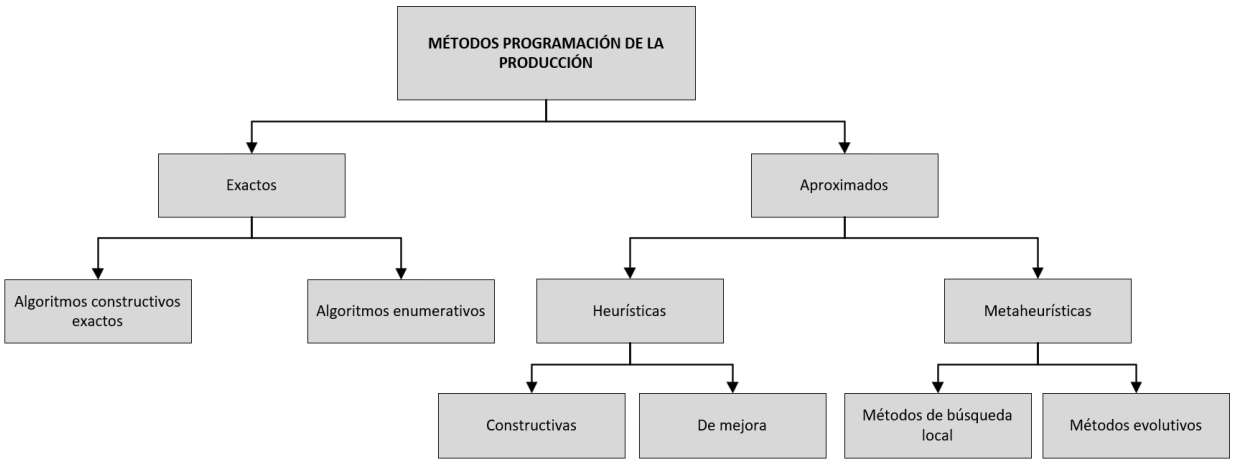


Figura 4.1. Clasificación de los métodos de resolución para problemas de programación de la producción [Fuente: Elaboración propia]

4.2 Metodología específica seleccionada

Como ya se ha comentado en el apartado 3.4, el PFSP resulta ser, de manera general, un problema NP-hard en sentido fuerte. Debido a esto, no resulta eficiente resolverlo mediante métodos exactos, por lo que se van a emplear métodos aproximados con el objetivo de conseguir soluciones próximas al óptimo en un tiempo computacional razonable. En concreto, en este trabajo se va a aplicar una metaheurística evolutiva, un algoritmo genético. A continuación, se describe en profundidad esta metodología.

En 1859, Charles Darwin propuso la teoría de la evolución natural en el libro "*On the Origin of Species*". Esta teoría afirma que los organismos biológicos, a lo largo de varias generaciones, evolucionan basándose en el principio de la selección natural "supervivencia del más apto". Esta teoría funcionaba tan bien en la naturaleza que resultó interesante aplicarla en el ámbito de los problemas de optimización, desarrollando de esta manera un método basado en la evolución natural que consigue obtener buenos resultados hasta para los problemas más complejos: el algoritmo genético.

En la naturaleza, los individuos de una población compiten unos contra otros por los recursos disponibles. En esta situación, los individuos con un rendimiento deficiente tienen menos oportunidades de sobrevivir que los individuos más adaptados o "aptos", los cuales producen una amplia descendencia. También se puede observar que, durante la reproducción, la recombinación de las buenas características de cada progenitor puede llegar a generar descendencia aún más apta. Después de unas cuantas generaciones, las especies evolucionan para adaptarse cada vez mejor al entorno (Sivanandam S.N. et al., 2008).

John H. Holland decidió desarrollar esta idea en su libro *"Adaptation in Natural and Artificial Systems"*, publicado en el año 1975. En él describió cómo aplicar los principios de la evolución biológica a problemas de optimización y construyó los primeros algoritmos genéticos. A lo largo de los años, estos algoritmos se han ido desarrollando y mejorando cada vez más hasta el punto tal que, hoy en día, se consideran una poderosa herramienta para resolver problemas complejos de búsqueda y optimización.

Un Algoritmo Genético (AG) parte de un conjunto, conocido como población inicial, de posibles soluciones denominadas individuos. Posteriormente, se lleva a cabo un proceso iterativo con el propósito de incrementar progresivamente la calidad de dicha población. Para manipular y modificar los diferentes individuos, el AG utiliza una serie de operadores genéticos como la selección, el cruce o la mutación. Estos operadores son realmente determinantes, ya que el comportamiento del algoritmo depende en gran medida de ellos. Resulta por tanto muy importante escoger los operadores correctos según las características del problema y el objetivo que se pretende alcanzar.

El funcionamiento general de un AG se resume en los siguientes pasos (Sivanandam S.N. et al., 2008):

1. Inicialización (*Initialisation*): en primer lugar, se genera una población inicial de individuos, que representan posibles soluciones al problema. En nuestro caso, al tratarse de un PFSP, los individuos se corresponderán con secuencias de *n* trabajos.

   Para evitar que el algoritmo converja rápidamente y se quede estancado, la población inicial debe ser lo suficientemente diversa. Una población con una amplia diversidad tiene más probabilidades de evolucionar que una población donde todos los individuos son muy parecidos.

   Esta inicialización se puede llevar a cabo de varias formas diferentes. La más habitual consiste en generar todos los individuos de la población de forma aleatoria.

2. Evaluación (*Evaluation*): cada individuo de la población se evalúa mediante una función de aptitud, denominada *fitness*, que mide la calidad de la solución que representa.

   En general, se puede tomar como *fitness* la propia función objetivo del problema, aunque también se pueden generar otras funciones *fitness* que dependan tanto de la función objetivo como de otros aspectos del problema. En cualquier caso, esta función está bien definida siempre y cuando el individuo con mejor *fitness* se corresponda con aquel que proporciona un mejor valor de la función objetivo (Pérez-González et al., 2021).

3. Selección (*Selection*): se seleccionan individuos de la población actual con el objetivo de reproducirlos y dar lugar a la generación de descendientes (*offsprings*).

   Esta selección puede realizarse a través de diversos procedimientos, tales como la selección aleatoria o estrategias donde los individuos con mayor aptitud tienen más probabilidades de ser seleccionados, como es el caso de la selección por torneo o la selección por ranking.

4. Cruce (*Crossover*): los individuos seleccionados se cruzan entre sí, generando de esta manera descendientes. El propósito del cruce es combinar características favorables de ambos "padres" para crear *offsprings* aún mejores.

   Existes muchas formas diferentes de efectuar dicha combinación: *single point crossover, uniform crossover, order crossover*, etc. Este paso permite la exploración de diferentes soluciones pertenecientes a la región de admisibilidad.

5. Mutación (*Mutation*): con la intención de introducir una variabilidad adicional al proceso, algunos de los nuevos individuos generados se someten a mutaciones aleatorias, es decir, se llevan a cabo pequeñas modificaciones en sus "genes".

   Existen distintos tipos de mutación. Algunas de las más comunes son la mutación por intercambio de pares, por inserción y la de tipo *shift* (Pérez-González et al., 2021).

   Al igual que ocurría con el paso anterior, estos pequeños cambios permiten explorar nuevas soluciones del espacio de búsqueda y evitan una convergencia prematura hacia óptimos locales.

6. Reemplazo (*Replacement*): los individuos de la nueva generación son evaluados mediante la función *fitness* y reemplazan a individuos menos aptos que ellos de la generación anterior. Esto garantiza que la calidad de la población mejore, o al menos se mantenga, pero nunca empeore, en cada iteración.

   La elección de los individuos a reemplazar se puede llevar a cabo mediante diversas metodologías. En algunos casos, los nuevos individuos sustituyen a sus progenitores, mientras que en otras ocasiones suplantan a aquellos individuos menos aptos de la población, entre otros escenarios posibles.

El AG itera hasta que se satisfaga algún criterio de parada, como un número máximo de iteraciones, un tiempo computacional límite o un nivel suficiente de aptitud alcanzado. A continuación, se muestran el pseudocódigo, en la Figura 4.2, y el diagrama de flujo, en la Figura 4.3, de un AG general.

Entrada: datos de la instancia, P_{size}
Salida: *Población,* Π_{best}, $fitness_{best}$
Inicio
    Población:= P_{size} secuencias iniciales;
    Π_{best}:= $Población_1$;
    $fitness_{best}$:= Fitness de Π_{best};
    Para *j*=1 **hasta** P_{size} **hacer**
        $fitness_j$:= Fitness de la secuencia $Población_j$;
        Si $fitness_j < fitness_{best}$ **entonces**
            Π_{best}:= $Población_j$;
            $fitness_{best}$:= $fitness_j$;

    Mientras *Criterio de parada* **no** *satisfecho* **hacer**
        $Padre_1$, $Padre_2$:= secuencias seleccionadas de *Población;*
        Hijo:= cruzar $Padre_1$ y $Padre_2$;
        Hijo':= mutar *Hijo;*
        $fitness_{hijo}$:= Fitness de *Hijo';*
        Para *j*=1 **hasta** P_{size} **hacer**
            Si $fitness_{hijo} < fitness_j$ **entonces**
                Reemplazar $Población_j$ por *Hijo'* en *Población;*
        Si $fitness_{hijo} < fitness_{best}$ **entonces**
            Π_{best}:= *Hijo';*
            $fitness_{best}$:= $fitness_{hijo}$;

    Devolver *Población,* Π_{best}, $fitness_{best}$

Figura 4.2. Pseudocódigo general de un AG [Fuente: Elaboración propia]

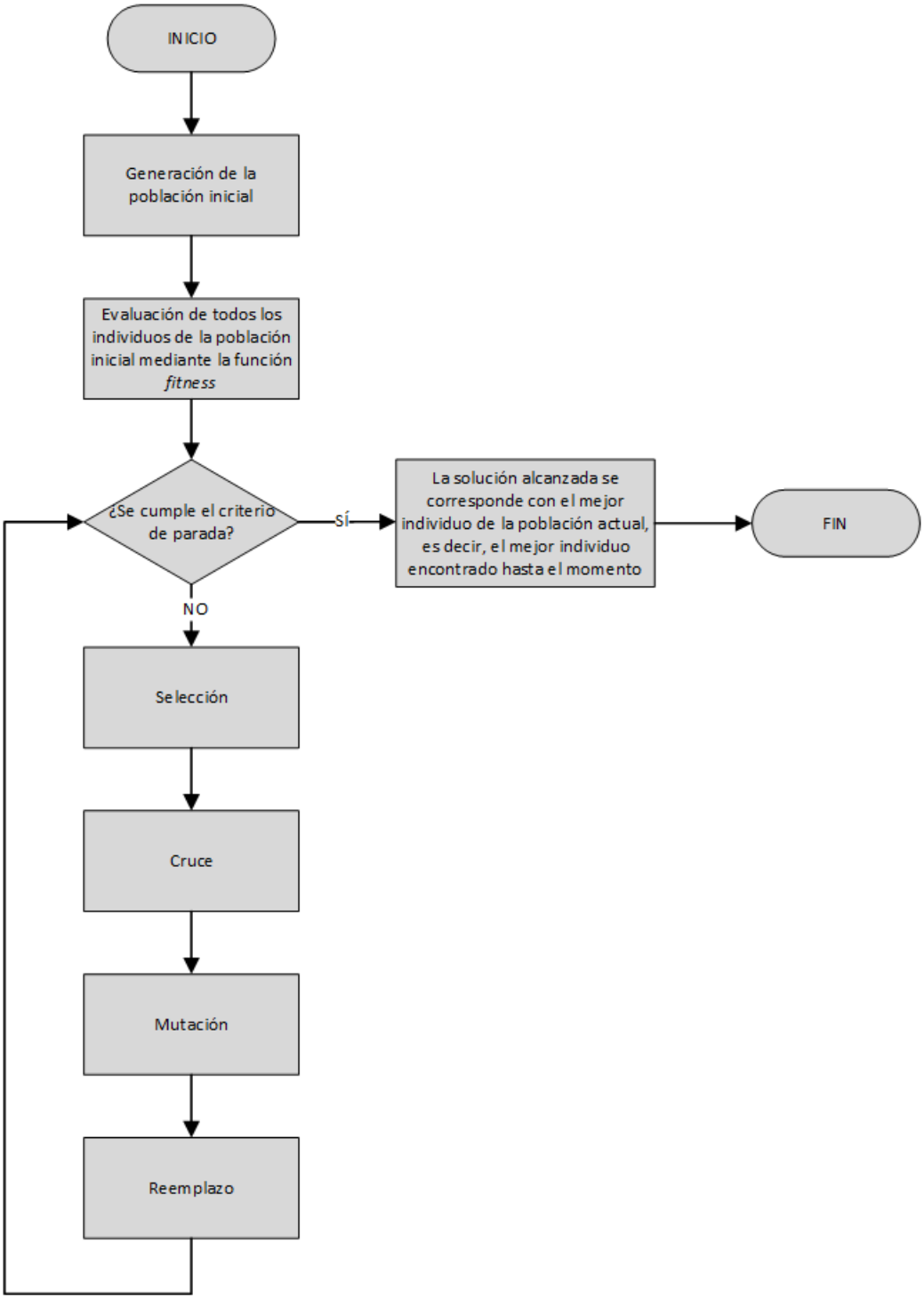


Figura 4.3. Diagrama de flujo del procedimiento general de un AG [Fuente: Elaboración propia]

5 METODOLOGÍA

Una vez explicado el funcionamiento general de la metodología seleccionada en este trabajo, se procede a describir su implementación específica. Se han desarrollado dos variantes del AG con la intención de comparar ambos procedimientos de resolución y emplear finalmente aquel que proporcione mejores resultados para el problema específico del PFSP. A continuación, se comentarán de manera detallada todos los operadores genéticos seleccionados, así como cada uno de los procedimientos concretos empleados por ambas variantes.

5.1 Variante 1 (V1)

La primera variante propuesta se fundamenta en el algoritmo genético desarrollado por Eva Vallada et al. (2010), sin embargo, se han efectuado algunas modificaciones.

5.1.1 Inicialización de la población

Como se ha comentado en el capítulo anterior, el primer paso del AG consiste en generar una población inicial compuesta por un número dado de P_{size} individuos. Dentro de las diversas metodologías existentes para crear esta población inicial, se ha escogido utilizar una de las más comunes: la generación de todos los individuos de manera aleatoria.

Cada individuo representa una posible solución al problema por lo que, teniendo en cuenta la codificación utilizada, se corresponde con una secuencia de *n* trabajos. Esta secuencia proporciona el orden en el que dichos trabajos deben ser procesados en las máquinas del taller. Al tratarse de un PFSP, la secuencia es la misma para todas las máquinas.

5.1.2 Evaluación de los individuos: función *fitness*

En este trabajo se ha decidido tomar la función *fitness* directamente como la función objetivo del problema. Cabe destacar que, como se han analizado varios objetivos diferentes, tanto objetivos relacionados con políticas de producción tradicionales como objetivos relacionados con la filosofía JIT, se han empleado varias funciones *fitness* diferentes. En la Figura 5.1 se muestra un resumen de esto.

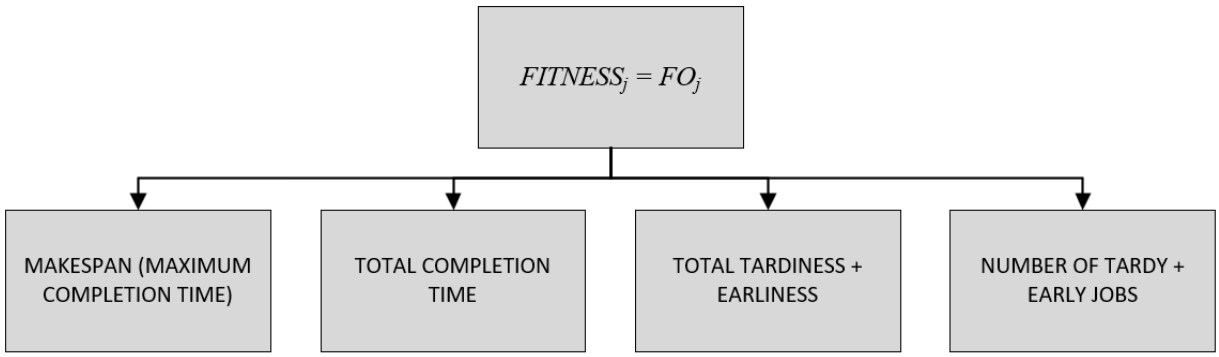


Figura 5.1. Funciones fitness empleadas por el AG [Fuente: Elaboración propia]

Resaltar que, el propósito en los cuatro casos es minimizar, es decir, cuanto menor sea el valor asignado por la función *fitness* para un individuo, mejor será la calidad de la solución representada por dicho individuo.

Es importante diseñar una función *fitness* adecuada para el problema en cuestión, ya que esto puede tener un gran impacto en la eficiencia y efectividad del AG. Una función mal diseñada puede hacer que el AG converja lentamente o que no logre encontrar soluciones de alta calidad.

5.1.3 Selección

Respecto al proceso de selección, los algoritmos genéticos tradicionales suelen emplear operadores conocidos como torneo y ranking. Estos operadores requieren la evaluación constante de todos los individuos de la población mediante la función *fitness*, lo que resulta ser un proceso tedioso y largo.

Con la idea de mejorar esto, en este trabajo se emplea un procedimiento de selección mucho más sencillo y veloz denominado *n-tournament*. En este caso, se selecciona aleatoriamente un porcentaje determinado de la población según un parámetro denominado *pressure*. De entre todos los individuos elegidos al azar, aquel individuo que proporcione un mejor valor de la función *fitness* gana el torneo y es finalmente seleccionado.

Este mecanismo de selección proporciona un AG considerablemente rápido, ya que no requiere la evaluación de la población completa, sino solamente de un porcentaje dado.

Por último, cabe destacar que es posible ajustar fácilmente el número de individuos seleccionados para el torneo aumentando o disminuyendo el parámetro *pressure.*

5.1.4 Cruce

En la literatura se proponen muchos operadores de cruce diferentes para los algoritmos genéticos. Por lo general, el objetivo de estos operadores es generar dos nuevos individuos, conocidos como *offsprings*, a partir de los dos progenitores seleccionados. En el artículo de Eva Vallada et al. (2010) los autores llegan a la conclusión, mediante experimentos estadísticos, de que los métodos de cruce más efectivos resultan ser: *similar block order crossover* (SBOX), *similar block two point order crossover* (SB2OX) y *one point order crossover* (OP).

En este trabajo, se opta por emplear el operador OP. En primer lugar, se selecciona aleatoriamente un punto de cruce en ambos individuos padres, elegidos en este caso mediante *n-tournament*. A partir de este punto, se toman los elementos de un padre y se colocan en la misma posición en el hijo, luego los elementos que faltan se colocan en el orden en que aparecen en el padre restante. Esto permite preservar algunos elementos de uno de los padres y, para el resto, mantener el orden relativo del otro padre. Finalmente, se consiguen obtener dos *offsprings* distintos. A continuación, en la Figura 5.2, se muestra el funcionamiento del operador de cruce OP.

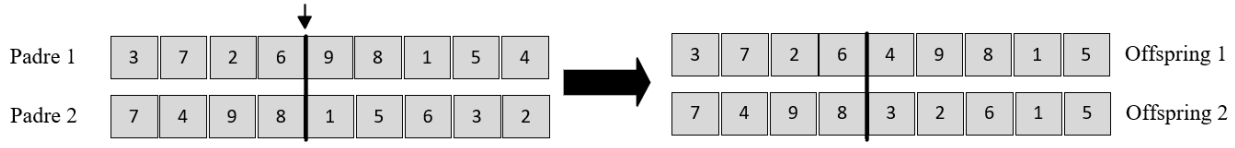


Figura 5.2. Funcionamiento operador de cruce *one point order crossover* (OP) [Fuente: Elaboración propia]

5.1.5 Mutación

Respecto al proceso de mutación, el operador más común y el que mejor funciona para el problema específico de PFSP es la mutación con desplazamiento, conocida como *shift mutation*. En ella cada elemento de cada *offspring* generado tiene una probabilidad P_m de ser cambiado de posición a otra aleatoria.

La probabilidad de mutación P_m es un parámetro muy importante en este proceso ya que controla la probabilidad de que ocurra una mutación en cada elemento. Si no hay mutación, los descendientes se generan inmediatamente después del cruce, es decir, no se produce ninguna modificación en ellos. En cambio, cuando hay mutación, los descendientes pueden sufrir algunos cambios pequeños.

En términos generales, este proceso procura una diversidad suficiente en la población evitando así la convergencia prematura del AG hacia óptimos locales. Aun así, la mutación no debería ocurrir muy a menudo ya que, en ese caso, la mayoría de los elementos cambiarían de posición creando individuos completamente nuevos y lo que se estaría llevando a cabo sería una búsqueda aleatoria, no un AG (Sivanandam S.N. et al., 2008).

5.1.6 Reemplazo

El último aspecto a tener en cuenta es la manera en la que los descendientes generados tras realizar la selección, el cruce y la mutación son introducidos en la población. Este proceso se conoce como esquema generacional (*generational scheme*). Un método debe determinar cuáles de los individuos actuales de la población, si alguno, deben ser sustituidos por los nuevos individuos.

La técnica empleada a la hora de decidir que miembros permanecen en la población y cuáles son reemplazados influye en gran medida en la convergencia del AG. Principalmente se distinguen dos tipos diferentes de métodos: las actualizaciones generacionales y las actualizaciones de estado estacionario (Sivanandam S.N. et al., 2008).

Un procedimiento muy común dentro de la primera categoría es aquel en el que los descendientes reemplazan directamente a sus progenitores. En algunos AGs, conocidos como elitistas, esta sustitución padre-hijo se lleva a cabo únicamente después de haber preservado a los mejores individuos de la población anterior, con el fin de evitar perderlos.

Por otra parte, en los AGs de estado estacionario, los descendientes no reemplazan a sus progenitores, sino que sustituyen a otros individuos de la población. En el algoritmo propuesto en este trabajo, los descendientes son admitidos en la población únicamente si son mejores que los peores individuos de dicha población y si, al mismo tiempo, son únicos, es decir, no existen individuos idénticos ya presentes en la población. En caso contrario, son rechazados.

Como resultado, la calidad de la población va mejorando constantemente, o en el peor de los casos se mantiene, pero nunca empeora. Al mismo tiempo, la población contiene únicamente individuos diferentes unos de otros, lo cual contribuye a mantener la diversidad y a impedir una convergencia prematura hacia óptimos locales (Vallada Eva et al., 2010).

A continuación, se expone el pseudocódigo de la primera variante propuesta, en la Figura 5.3.

Entrada: datos de la instancia, P_{size}
Salida: *Población,* Π_{best}, $fitness_{best}$
Inicio
    Población:= P_{size} secuencias iniciales generadas aleatoriamente;
    Π_{best}:= $Población_1$;
    $fitness_{best}$:= Fitness de Π_{best};

    Para j=1 **hasta** P_{size} **hacer**
        $fitness_j$:= Fitness de la secuencia $Población_j$;
        Si $fitness_j < fitness_{best}$ **entonces**
            Π_{best}:= $Población_j$;
            $fitness_{best}$:= $fitness_j$;

    Mientras *Criterio de parada* **no** *satisfecho* **hacer**
        $Padre_1$, $Padre_2$:= secuencias seleccionadas de *Población* mediante *n-tournament;*
        $Hijo_1$, $Hijo_2$:= cruzar $Padre_1$ y $Padre_2$ mediante el operador OP;
        $Hijo_1$' , $Hijo_2$':= mutar $Hijo_1$ y $Hijo_2$ mediante *shift mutation;*

        Para *Hijo* = ($Hijo_1$' **y** $Hijo_2$') **hacer**
            Π_{worst}:= $Población_1$;
            $fitness_{worst}$:= Fitness de Π_{worst};
            Para j=1 **hasta** P_{size} **hacer**
                $fitness_j$:= Fitness de la secuencia $Población_j$;
                Si $fitness_j > fitness_{worst}$ **entonces**
                    Π_{worst}:= $Población_j$;
                    $fitness_{worst}$:= $fitness_j$;

            $fitness_{hijo}$:= Fitness de *Hijo;*
            Si $fitness_{hijo} < fitness_{worst}$ **y** *Hijo* **no en** *Población* **entonces**
                Reemplazar Π_{worst} por *Hijo* en *Población;*
                Si $fitness_{hijo} < fitness_{best}$ **entonces**
                    Π_{best}:= *Hijo;*
                    $fitness_{best}$:= $fitness_{hijo}$;

    Devolver *Población,* Π_{best}, $fitness_{best}$

Figura 5.3. Pseudocódigo de la variante 1 del AG [Fuente: Elaboración propia]

5.2 Variante 2 (V2)

Respecto a la segunda variante propuesta, se toma como punto de partida la variante anterior, por lo tanto, los operadores genéticos escogidos son los mismos. Sin embargo, a esta nueva versión, se le incorporan además los siguientes mecanismos y técnicas.

5.2.1 Inicialización de la población con regla de despacho

Una tendencia actual en la literatura consiste en incluir en la población inicial uno o varios individuos de calidad obtenidos mediante heurísticas eficaces o reglas de despacho. Esta forma de inicializar la población garantiza una convergencia más rápida hacia buenas soluciones ya que se parte de una población que contiene ya de antemano individuos "buenos". Los AG que utilizan esta técnica son ampliamente utilizados en la práctica y se pueden encontrar en la mayoría de los estudios relacionados con el PFSP (Vallada Eva et al., 2010).

En esta segunda variante, en vez de generar todos los individuos de la población inicial de forma aleatoria como se hacía en la variante anterior, se ha optado por emplear esta tendencia reciente y utilizar una regla de despacho para inicializar la población. En esta ocasión, todos los individuos son generados de manera aleatoria, excepto uno que se obtiene mediante una regla de despacho. De este modo se asegura que al menos exista un "buen" individuo en la población inicial.

Como ya se ha comentado anteriormente, en este trabajo se analizan tanto objetivos relacionados con políticas de producción tradicionales como objetivos relacionados con la filosofía JIT. Debido a esto, se proponen dos reglas de despacho diferentes según el caso en el que se esté:

- Regla SPT (*Shortest Processing Time first*): se secuencian los trabajos en orden creciente de los tiempos de proceso, es decir, el primer trabajo de la secuencia es aquel con menor tiempo de proceso y el último aquel con mayor tiempo de proceso.
  Como el problema abordado es un taller que cuenta con más de una máquina, el tiempo de proceso empleado para ordenar cada trabajo se calcula sumando los tiempos de proceso de dicho trabajo en todas las máquinas del taller, tal y como se muestra en la ecuación 5-1.

$$p_j = \sum_{i=1}^{m} p_{ij} \tag{5–1}$$

  Esta regla de despacho se utiliza principalmente en sistemas con políticas de producción tradicionales ya que, al dar prioridad a aquellos trabajos con tiempo de procesamiento más corto, se consiguen obtener tiempos de terminación menores y reducir el tiempo total de producción.

- Regla EDD (*Earlier Due Date first*): se secuencian los trabajos en orden creciente de las fechas de entrega, es decir, el primer trabajo de la secuencia es aquel con la fecha de entrega más temprana y el último aquel con la fecha de entrega más tardía.
  El objetivo principal de esta regla de despacho es minimizar el retraso total e intentar cumplir con los plazos de entrega establecidos, es decir, evitar que los trabajos se atrasen más allá de su fecha límite. Se trata por tanto de una regla que funciona muy bien en sistemas de producción basados en la filosofía JIT.

5.2.2 Mecanismo de reinicio: diversidad de la población

Como ya se ha comentado anteriormente, una población poco diversa, es decir, una población en la que los individuos, aunque no iguales, son muy parecidos entre sí, provoca que el AG se estanque rápidamente de manera que la población para de evolucionar y mejorar. Con la idea de evitar esto, se ha decidido aplicar un mecanismo de reinicio que permite abandonar la población actual en la que el AG está estancado y comenzar a trabajar con una población completamente nueva.

El mecanismo de reinicio implementado en este trabajo consiste en la regeneración completa de la población, es decir, de todos los individuos que la conforman, cada vez que el valor de la diversidad cae por debajo de un valor determinado, denominado umbral *Div*. La población se regenera de la misma manera que se inicializa: un individuo se obtiene mediante la regla de despacho SPT o EDD, según corresponda, y el resto se generan aleatoriamente.

En este trabajo se propone calcular el valor de la diversidad de la misma manera que en el artículo de Eva Vallada et al. (2010), basándose en la cantidad de veces que cada trabajo aparece en cada posición de los individuos que conforman la población.

El valor final que se obtiene, cuyo calculo se detalla más adelante, varía entre cero y uno, de tal forma que un valor cercano a uno señala una población altamente diversa, donde cada trabajo ocupa distintas posiciones dentro de los individuos. Por el contrario, un valor cercano a cero indica que todos los individuos (aunque diferentes debido al operador de reemplazo descrito en el apartado 5.1.6) son muy parecidos, lo cual denota una población empobrecida con escasa diversidad.

El procedimiento utilizado para calcular el valor de la diversidad de la población es el siguiente:

1. Se calcula una matriz, denominada *GeneCount,* que indica el número de veces que un trabajo α aparece en una determinada posición *k* para todos los individuos de la población.

$$c_k(\alpha) = \sum_{i=1}^{P_{size}} \sum_{j=1}^{n} \delta_{ij}(\alpha) \qquad \alpha, k = 1, \dots, n \tag{5–2}$$

   Donde $\delta_{ij}(\alpha)$ es una variable binaria que toma el valor 1 cuando el trabajo en la posición *j* del individuo *i* coincide con el trabajo α y, al mismo tiempo, $j = k$. En caso contrario toma el valor 0.

2. Se calcula una segunda matriz, denominada *GeneFrecuency,* que representa el ratio entre el recuento de cada trabajo en cada posición y el tamaño de la población.

$$f_k(\alpha) = \frac{c_k(\alpha)}{P_{size}} \qquad \alpha, k = 1, \dots, n \tag{5–3}$$

3. Se calcula el valor de la diversidad de la población de la siguiente manera:

$$Diversidad = \frac{1}{n-1} \sum_{k=1}^{n} \sum_{\alpha=1}^{n} f_k(\alpha)(1 - f_k(\alpha)) \tag{5–4}$$

   Como se puede observar, finalmente se obtiene un valor de la diversidad entre cero y uno.

5.2.3 Búsqueda local

El último procedimiento incorporado en esta segunda variante consiste en una búsqueda local. Se propone utilizar para esta la vecindad de inserción ya que es la más común en el caso del PFSP.

Esta búsqueda local propuesta se aplica a cada *offspring* generado tras el cruce y la mutación con una probabilidad P_{ls}.

En caso de que por probabilidad toque realizar la búsqueda local, al *offspring* generado se le calculan sus vecinos mediante inserción, es decir, se extrae un trabajo de la secuencia y se inserta en todas las posiciones posibles *n*. Iterando y realizando esto para todos los trabajos del *offspring*, se consigue un total de $(n-1)^2$ vecinos.

En cada iteración de la búsqueda local se evalúa mediante la función *fitness* el vecino obtenido y, si resulta ser mejor que el mejor individuo encontrado hasta el momento, se actualiza la solución. Finalmente, la solución obtenida se corresponde con el mejor vecino de todos los evaluados.

Ejecutar esta búsqueda local requiere de un elevado tiempo computacional por lo que se ha optado por imponer un criterio de parada, de manera que, aunque no se hayan evaluado todos los vecinos, si se alcanza dicho criterio de parada, la búsqueda local finaliza y proporciona el mejor individuo encontrado hasta el momento. Esto asegura que el algoritmo no se quede "encerrado" en esta búsqueda local, evaluando vecinos, más tiempo del impuesto. El criterio de parada utilizado es el mismo que se emplea para las dos variantes del AG y se expone en el capítulo 6.

El pseudocódigo de la segunda variante propuesta se muestra a continuación, en la Figura 5.4.

Entrada: datos de la instancia, P_{size}
Salida: *Población,* Π_{best}, $fitness_{best}$
Inicio
    Población:= P_{size} secuencias iniciales (una por regla EDD o SPT y resto aleatoriamente)*;*
    Π_{best}*:= Población*$_1$*;*
    $fitness_{best}$*:=* Fitness de Π_{best}*;*

    Para *j*=1 **hasta** P_{size} **hacer**
        $fitness_j$*:=* Fitness de la secuencia *Población*$_j$*;*
        Si $fitness_j < fitness_{best}$ **entonces**
            Π_{best}*:= Población*$_j$*;*
            $fitness_{best}$*:=* $fitness_j$*;*

    Mientras *Criterio de parada* **no** *satisfecho* **hacer**
        Si *Diversidad Población < Div* **entonces**
            Población:= P_{size} secuencias (una por regla EDD o SPT y resto aleatoriamente);

        Padre$_1$ *, Padre*$_2$*:=* secuencias seleccionadas de *Población* mediante *n-tournament;*
        Hijo$_1$ *, Hijo*$_2$*:=* cruzar *Padre*$_1$ y *Padre*$_2$ mediante el operador OP*;*
        Hijo$_1$*'*, *Hijo*$_2$*':=* mutar *Hijo*$_1$ y *Hijo*$_2$ mediante *shift mutation;*

        Mientras *Criterio de parada* **no** *satisfecho* **hacer**
            $Vecino_{1,best}$, $Vecino_{2,best}$ *:=* aplicar búsqueda local de inserción a *Hijo*$_1$*'* y *Hijo*$_2$*';*

        Para *Vecino* = (*Vecino*$_{1,best}$ **y** *Vecino*$_{2,best}$) **hacer**
            Π_{worst}*:= Población*$_1$*;*
            $fitness_{worst}$*:=* Fitness de Π_{worst}*;*
            Para *j*=1 **hasta** P_{size} **hacer**
                $fitness_j$*:=* Fitness de la secuencia *Población*$_j$*;*
                Si $fitness_j > fitness_{worst}$ **entonces**
                    Π_{worst}*:= Población*$_j$*;*
                    $fitness_{worst}$*:=* $fitness_j$*;*

            $fitness_{vecino}$*:=* Fitness de *Vecino;*
            Si $fitness_{vecino} < fitness_{worst}$ **y** *Vecino* **no en** *Población* **entonces**
                Reemplazar Π_{worst} por *Vecino* en *Población;*
                Si $fitness_{vecino} < fitness_{best}$ **entonces**
                    Π_{best}*:= Vecino;*
                    $fitness_{best}$*:=* $fitness_{vecino}$*;*

    Devolver *Población,* Π_{best}, $fitness_{best}$

Figura 5.4. Pseudocódigo de la variante 2 del AG [Fuente: Elaboración propia]

Asimismo, en la Tabla 5.1 se muestra un resumen de los operadores genéticos y mecanismos empleados por cada una de las variantes.

Procedimiento	Variante (V1)	Variante (V2)
Inicialización población	Todos los individuos se generan de manera aleatoria	Un individuo se obtiene mediante la regla de despacho SPT o EDD y el resto se generan aleatoriamente
Selección	*n-tournament*	*n-tournament*
Cruce	*One point order crossover* (OP)	*One point order crossover* (OP)
Mutación	Con desplazamiento (S*hift mutation)*	Con desplazamiento (S*hift mutation)*
Reemplazo	Los *offprings* sustituyen a los peores individuos de la población sólo si son mejores que estos y si, al mismo tiempo, son únicos	Los *offprings* sustituyen a los peores individuos de la población sólo si son mejores que estos y si, al mismo tiempo, son únicos
Mecanismo de reinicio	-	Regeneración completa de la población cada vez que el valor de la diversidad cae por debajo del umbral
Búsqueda local	-	Basada en la vecindad de inserción (*Insertion neighbourhood*)

Tabla 5.1. Resumen operadores genéticos y mecanismos empleados en cada variante

6 EVALUACIÓN COMPUTACIONAL

En el presente capítulo se exponen, analizan y comparan los resultados obtenidos al evaluar un conjunto de instancias para cuatro políticas de producción diferentes, mediante las dos variantes del AG propuestas. En primer lugar, se realizará una comparación entre ambas variantes con el propósito de determinar cuál de ellas arroja mejores resultados. Posteriormente, se procederá a comparar las distintas políticas de producción, tanto las tradicionales como aquellas basadas en la filosofía JIT, haciendo uso de aquella variante que haya proporcionado mejores resultados.

También se detallan en este capítulo los valores tomados para los parámetros del AG, así como la batería de instancias evaluadas y los indicadores de desempeño empleados para el análisis y la comparación de los resultados.

Las dos variantes del AG propuestas en este trabajo se han implementado en lenguaje de programación Python en el entorno de desarrollo Spyder en su versión 3.9. Además, todas las evaluaciones se han ejecutado en un ordenador con procesador AMD Ryzen 7 5800H que funciona a 3.20 GHz, una memoria RAM de 16 GB y el sistema operativo Microsoft Windows 11 Home 64 bits.

6.1 Batería de instancias

Tanto para la comparación de las dos variantes del AG como para la comparación de las cuatro diferentes políticas de producción, se ha llevado a cabo la evaluación de un conjunto común de instancias. Este conjunto de instancias se ha obtenido de http://www.upv.es/gio/rruiz. Cada instancia contiene los siguientes datos de entrada al problema: número de trabajos n, número de máquinas m, tiempos de proceso p_{ij} y fechas de entrega d_j.

Se manejan un total de 540 instancias, generadas de la siguiente manera:

- Los tiempos de proceso p_{ij} siguen una distribución uniforme entre 1 y 99, como es común en la literatura.

- En cuanto al número de trabajos n y el número de máquinas m, se analizan las siguientes combinaciones: n = {50, 150, 250, 350} y m = {10, 30, 50}.

- Las fechas de entrega d_j se generan de acuerdo con el *tardiness factor* (T) y *el due date range* (R), mediante una distribución uniforme entre P (1-T-R/2) y P (1-T+R/2), siendo P un límite inferior estricto del *makespan*.
  Se estudian las siguientes combinaciones de T y R: T = {0.2, 0.4, 0.6} y R = {0.2, 0.6, 1}. Cabe destacar que, en ciertos escenarios, especialmente cuando T toma valores bajos y R valores altos, es posible que algunas fechas de entrega resulten negativas. Para solucionar esto, se sustituyen dichas fechas de entrega por cero.

En total, se tienen 12 combinaciones de n y m, para cada una de las cuales se generan 5 replicaciones. Se tienen por tanto 60 instancias para cada una de las 9 combinaciones de T y R, obteniendo finalmente un total de 540 instancias.

6.2 Valores de los parámetros del AG

Los valores tomados para los parámetros del AG son los mismos para ambas variantes y se mantienen fijos a lo largo de todas las evaluaciones realizadas, garantizando así que todos los resultados obtenidos sean comparables entre sí.

Estos valores se han establecido tomando como referencia el artículo de Eva Vallada et al. (2010), dónde se estudian diferentes combinaciones de valores para cada parámetro y se calibran para tres AG similares a los propuestos en este trabajo.

A continuación, en la Tabla 6.1, se muestra el valor asignado a cada parámetro:

Parámetro	Notación	Valor asignado
Tamaño de la población	P_{size}	30
Porcentaje de selección	*Pressure*	0.3
Probabilidad de mutación	P_m	0.02
Umbral de diversidad	*Div*	0.4
Probabilidad de búsqueda local	P_{ls}	0.15

Tabla 6.1. Valores de los parámetros del AG

Por otro lado, respecto al criterio de parada, se establece un tiempo computacional límite, de manera que el AG itera hasta que se alcanza dicho tiempo computacional, momento en el que finaliza su ejecución.

Cabe destacar que el criterio de parada empleado es el mismo para ambas variantes y depende del tamaño de la instancia evaluada, es decir, depende del número de trabajos *n* y del número de máquinas *m*. La ecuación 6-1 muestra la fórmula utilizada para calcular el tiempo de parada en función de estos dos parámetros. Se trata de una fórmula muy común en la literatura.

$$Tiempo\ parada = \frac{30 \cdot n \cdot m}{2000}\ segundos \tag{6–1}$$

Establecer el tiempo de parada de esta manera implica un mayor esfuerzo computacional a medida que aumenta el número de trabajos y/o máquinas. Como se puede observar, para unos valores determinados de *n* y *m*, es decir, para un determinado tamaño de instancia, el tiempo de parada se mantiene constante. Esto implica que, como los valores obtenidos para cada instancia se comparan entre sí y no con los de otra instancia, los resultados obtenidos vuelven a ser comparables entre sí.

Tamaño de instancia {*n, m*}	Tiempo de parada (segundos)
{50, 10}	7.5
{50, 30}	22.5
{50, 50}	37.5
{150, 10}	22.5
{150, 30}	67.5
{150, 50}	112.5
{250, 10}	37.5
{250, 30}	112.5
{250, 50}	187.5
{350, 10}	52.5
{350, 30}	157.5
{350, 50}	262.5

Tabla 6.2. Tiempo de parada para cada tamaño de instancia {*n, m*}

6.3 Indicadores de desempeño

Para poder evaluar y comparar la calidad de las soluciones obtenidas es necesario hacer uso de indicadores de desempeño.

Antes de exponer los indicadores utilizados en este trabajo, se presenta la notación empleada:

- I es el número de instancias ($i \in \{1, ..., I\}$)
- J es el número de objetivos/indicadores analizados ($j \in \{1, ..., J\}$). En total se estudian 12 objetivos/indicadores.
- K es el número de algoritmos o políticas de producción bajo comparación ($k \in \{1, ..., K\}$). En el primer análisis se comparan las dos variantes del AG mientras que en el segundo análisis realizado se comparan cuatro políticas de producción diferentes.

El indicador más frecuentemente utilizado en la literatura es la desviación porcentual relativa (*Relative Percentage Deviation*: RPD), el cual se calcula de la siguiente manera:

$$RPD_{ijk} = \frac{Valor\ obtenido_{ijk} - Mejor\ valor_{ij}}{Mejor\ valor_{ij}} * 100 \qquad (6\text{–}2)$$

Sin embargo, para varios de los objetivos estudiados en este trabajo, el mejor valor puede ser cero, generando en este caso una división entre cero, una indefinición, a la hora de calcular el RPD. Además, si el mejor valor resulta ser muy pequeño, este indicador subestima al resto de valores obtenidos, aunque estos sean sólo ligeramente peores.

Debido a ello y con el propósito de evitar estos problemas, en este trabajo se va a utilizar un indicador de desempeño diferente: el índice de desviación relativa (*Relative Deviation Index*: RDI):

$$RDI_{ijk} = \frac{Valor\ obtenido_{ijk} - Mejor\ valor_{ij}}{Peor\ valor_{ij} - Mejor\ valor_{ij}} * 100 \quad (6\text{–}3)$$

Donde:

- $Valor\ obtenido_{ijk}$: valor obtenido para la instancia *i* en el objetivo/indicador *j* mediante la variante del algoritmo o política de producción *k*
- $Mejor\ valor_{ij}$: el mejor valor obtenido entre ambas variantes o entre las cuatro políticas de producción para la instancia *i* en el objetivo/indicador *j*
- $Peor\ valor_{ij}$: el peor valor obtenido entre ambas variantes o entre las cuatro políticas de producción para la instancia *i* en el objetivo/indicador *j*

Mediante el indicador RDI se obtiene un valor entre cero y cien, de manera que un valor pequeño indica proximidad a la mejor solución y un valor grande proximidad a la peor. En concreto, si se obtiene el valor cero significa que se ha alcanzado la mejor mientras que si se obtiene el valor cien significa que se ha alcanzado la peor.

Cabe destacar que, si la mejor y la peor solución toman el mismo valor, entonces ambas variantes o las cuatro políticas de producción, según corresponda, proporcionan el mejor valor (el mismo), por ende, el valor del RDI para todas ellas será cero.

La media del RDI para todas las instancias evaluadas se denomina *Average Relative Deviation Index* (ARDI):

$$ARDI_{jk} = \sum_{i} \frac{RDI_{ijk}}{I} \quad (6\text{–}4)$$

Además de utilizar el indicador RDI en sí y su media ARDI, se va a calcular otra medida que también permite evaluar y comparar los resultados obtenidos: el número de instancias para las cuales se obtiene el mejor resultado. Esta medida se calcula teniendo en cuenta que, si $RDI_{ijk} = 0$, significa que la variante o política de producción *k* ha obtenido el mejor valor para la instancia *i* en el objetivo/indicador *j*. A diferencia del indicador RDI, esta medida no considera la desviación que se produce entre el valor obtenido en cuestión y el mejor valor, sencillamente tiene en cuenta si dicho valor obtenido coincide con el mejor o no.

Por último, cabe destacar que la ecuación 6-3 se utiliza para el caso de minimizar, sin embargo, existe una excepción, el objetivo *Number of JIT Jobs*, dónde el propósito es maximizar. Para este objetivo el indicador RDI se calcula de la siguiente manera:

$$RDI_{ijk} = \frac{Mejor\ valor_{ij} - Valor\ obtenido_{ijk}}{Mejor\ valor_{ij} - Peor\ valor_{ij}} * 100 \quad (6\text{–}5)$$

De esta manera, sea el caso que sea, maximizar o minimizar, el valor que se obtiene para el indicador RDI resulta ser siempre un valor positivo, permitiendo comparar sin ningún problema los resultados entre sí.

6.4 Análisis y comparación de las dos variantes del AG

Para analizar y comparar las dos variantes del AG se propone estudiar por separado dos funciones objetivo, es decir, dos políticas de producción diferentes. En concreto se escoge una política de producción tradicional (*Makespan*) y otra basada en la filosofía JIT (*Total Tardiness + Earliness*) con el propósito de determinar que variante proporciona mejores resultados en ambos casos.

6.4.1 Política de producción tradicional: *Makespan*

Los valores ARDI obtenidos para cada variante respecto a cada uno de los objetivos/indicadores estudiados se exponen en la Tabla 6.3:

Objetivos/indicadores	V1	V2
Total Tardiness	11,30	88,70
Total Earliness	65,00	33,52
Total Tardiness + Earliness	22,41	77,59
Number of Tardy Jobs	31,48	61,48
Number of Early Jobs	61,48	31,85
Number of Tardy + Early Jobs	1,48	1,11
Number of JIT Jobs	1,48	1,11
Makespan	7,04	92,41
Total Completion Time	12,41	87,59
Tiempo ocioso	8,89	91,11
Tiempo de espera	23,33	76,67
Tiempo medio en inventario (%)	29,44	70,56
Media	22,98	59,48

Tabla 6.3. Valores ARDI de cada variante para cada objetivo/indicador con política *Makespan*

Como se puede observar, la V1 proporciona valores bastante menores del ARDI que la V2 para la mayoría de los objetivos e indicadores, en concreto, para ocho de ellos. Por su parte, la V2 funciona notablemente mejor únicamente para dos objetivos, el *Total Earliness* y el *Number of Early Jobs*. Respecto al *Number of Tardy + Early Jobs* y al *Number of JIT Jobs*, aunque la segunda variante proporciona ligeramente un menor valor del ARDI, ambas variantes alcanzan resultados muy similares.

Esto significa que la V1, para la mayoría de los objetivos, se aproxima más a la mejor solución, es decir, obtiene mejores resultados que la V2. Si además se tiene en cuenta la media del ARDI, se reafirma que la primera variante funciona de manera general notablemente mejor que la segunda, ya que obtiene una media mucho menor. Esta media se obtiene haciendo el promedio del ARDI para los 12 objetivos/indicadores analizados.

Esto se puede observar gráficamente en la Figura 6.1 ya que la línea asociada a la V1, representada en azul, está por debajo de la asociada a la V2, en naranja, para la mayoría de los objetivos e indicadores.

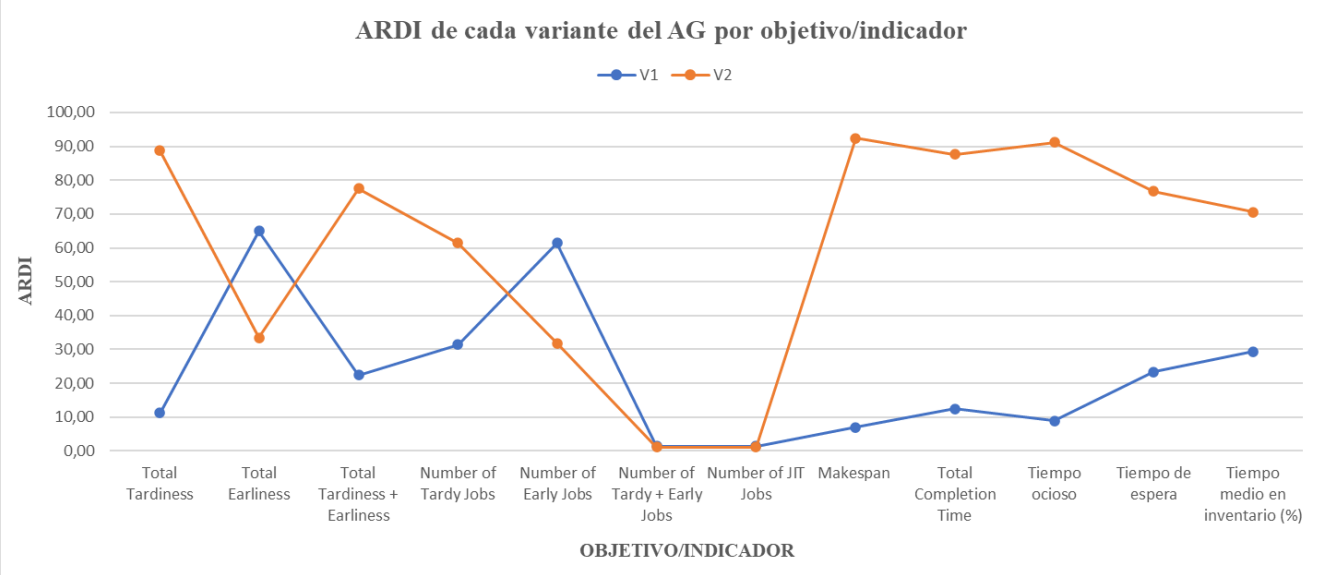


Figura 6.1. Gráfico ARDI de cada variante por objetivo/indicador con política *Makespan* [Fuente: Elaboración propia]

Además de los valores obtenidos para el indicador ARDI, se analiza también el número de instancias para las cuales cada una de las variantes proporciona la mejor solución respecto a cada objetivo/indicador analizado. Los resultados obtenidos se muestran a continuación en la Figura 6.2:

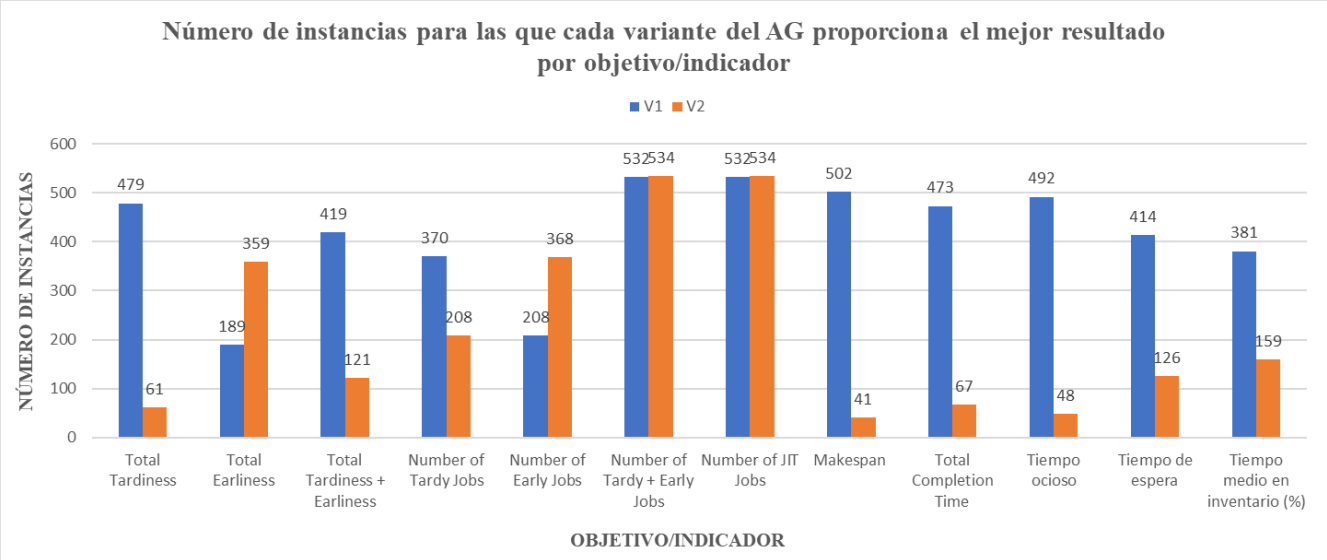


Figura 6.2. Gráfico número de instancias para las que cada variante proporciona el mejor resultado por objetivo/indicador con política *Makespan* [Fuente: Elaboración propia]

De nuevo se observa que la V1 alcanza, de manera general, el mejor resultado para un mayor número de instancias, es más, para muchos de los objetivos e indicadores esta variante llega a alcanzar la mejor solución para casi el 100%. Por el contrario, para aquellos casos en los que la V2 funciona mejor, este número de instancias no resulta ser tan dispar.

En total, la V1 proporciona el mejor para 4991 instancias mientras que la V2 lo hace sólo para 2626 instancias, aproximadamente la mitad.

6.4.2 Política de producción JIT: *Total Tardiness + Earliness*

Para esta segunda política de producción estudiada, los resultados obtenidos para el indicador ARDI se muestran en la Tabla 6.4:

Objetivos/indicadores	V1	V2
Total Tardiness	49,81	50,19
Total Earliness	85,56	13,15
Total Tardiness + Earliness	62,22	37,78
Number of Tardy Jobs	29,07	65,93
Number of Early Jobs	65,93	29,26
Number of Tardy + Early Jobs	1,48	3,15
Number of JIT Jobs	1,48	3,15
Makespan	11,48	88,33
Total Completion Time	17,59	82,41
Tiempo ocioso	16,48	83,52
Tiempo de espera	25,19	74,81
Tiempo medio en inventario (%)	34,44	65,56
Media	33,40	49,77

Tabla 6.4. Valores ARDI de cada variante para cada objetivo/indicador con política *Total Tardiness + Earliness*

Aunque los valores obtenidos ya no resultan ser tan dispares como en el caso anterior, la V1 sigue proporcionando valores considerablemente menores del ARDI para la mitad de los objetivos, es decir, para seis. Por su parte, la V2 sigue funcionando mejor que la V1 para el *Total Earliness* y el *Number of Early Jobs*, pero ahora además, a diferencia de lo que ocurría en el caso anterior, también lo hace para el *Total Tardiness + Earliness*. Ambas variantes alcanzan soluciones similares para tres de los objetivos: *Number of Tardy + Early Jobs, Number of JIT Jobs,* tal y como ocurría anteriormente, y *Total Tardiness.*

Respecto a las medias del ARDI, la V1 vuelve a obtener un valor menor, aunque la diferencia ya no es tan grande. Se llega por tanto a la misma conclusión: la V1, de media, se aproxima más a la mejor solución, es decir, obtiene mejores resultados que la V2. Aun así, cabe destacar que esta segunda variante funciona mejor para políticas de producción JIT que para políticas tradicionales.

Estos resultados se representan gráficamente a continuación, en la Figura 6.3.

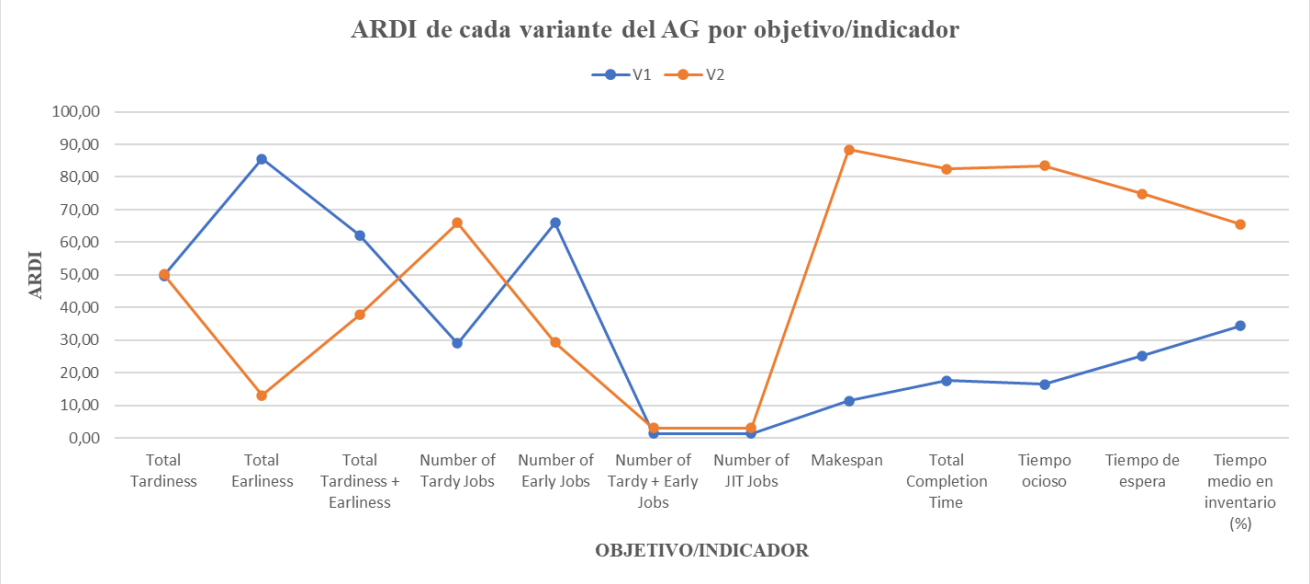


Figura 6.3. Gráfico ARDI de cada variante por objetivo/indicador con política *Total Tardiness + Earliness* [Fuente: Elaboración propia]

Asimismo, se exponen también gráficamente en la Figura 6.4 el número de instancias para las cuales cada variante alcanza la mejor solución respecto a cada objetivo/indicador estudiado.

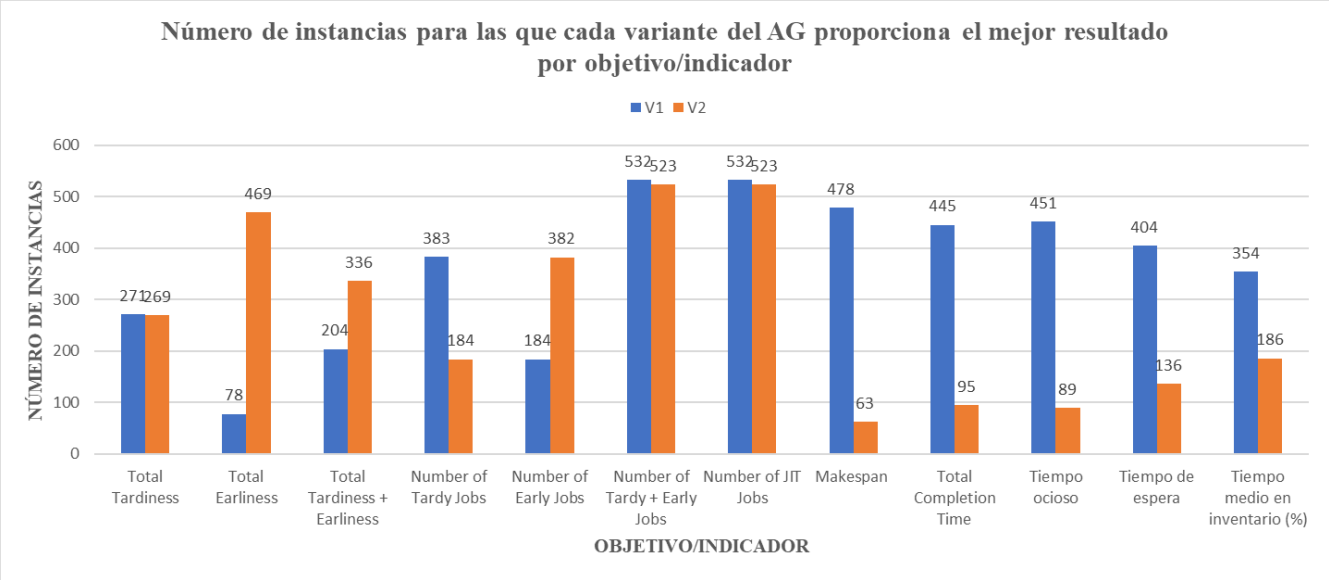


Figura 6.4. Gráfico número de instancias para las que cada variante proporciona el mejor resultado por objetivo/indicador con política *Total Tardiness + Earliness* [Fuente: Elaboración propia]

Si se realiza una suma teniendo en cuenta todos los objetivos e indicadores, la V1 consigue obtener la mejor solución para un total de 4316 instancias mientras que la V2 la obtiene para 3255. De nuevo se alcanza la misma conclusión: para esta política de producción la V1 vuelve a funcionar de manera general mejor que la V2, ya que alcanza la mejor solución para un mayor número de instancias, aunque la diferencia ya no es tan notable.

6.5 Análisis y comparación de las diferentes políticas de producción

Gracias a la comparación realizada en el apartado anterior, se llega a la conclusión de que la V1 del AG proporciona de manera general mejores resultados que la V2. Debido a ello, se utiliza esta primera variante para llevar a cabo el segundo análisis planteado, en el cual se comparan cuatro políticas de producción diferentes: *Makespan, Total Completion Time, Total Tardiness + Earliness y Number of Tardy + Early Jobs*.

En la Tabla 6.5 se exponen los valores obtenidos del ARDI para cada política respecto a cada objetivo/indicador:

Objetivos/indicadores	Makespan	Total Completion Time	Total Tardiness + Earliness	Number of Tardy + Early Jobs
Total Tardiness	48,54	22,15	6,70	99,97
Total Earliness	70,86	96,31	1,45	57,46
Total Tardiness + Earliness	66,93	59,05	0,88	97,09
Number of Tardy Jobs	42,00	4,79	86,38	63,98
Number of Early Jobs	58,84	94,83	14,79	28,18
Number of Tardy + Early Jobs	94,44	93,75	92,41	0,19
Number of JIT Jobs	94,44	93,75	92,41	0,19
Makespan	0,10	29,18	56,63	98,75
Total Completion Time	41,54	0,07	57,24	96,50
Tiempo ocioso	2,94	35,76	52,37	95,25
Tiempo de espera	50,10	0,65	54,95	92,54
Tiempo medio en inventario (%)	60,41	0,86	61,51	89,07
Media	52,60	44,26	48,14	68,26

Tabla 6.5. Valores ARDI de cada política de producción para cada objetivo/indicador

Como se puede observar, la política *Makespan* alcanza valores menores de ARDI para dos objetivos/indicadores, la política *Total Completion Time* para cuatro, la política *Total Tardiness + Earliness* también para cuatro y, por último, la política *Number of Tardy + Early Jobs* para dos.

En relación a estos resultados, cabe resaltar que ninguna política destaca entre las demás, sino que el funcionamiento de cada una depende del objetivo/indicador que se esté considerando. Si el propósito es cumplir de la mejor manera posible con los objetivos de producción tradicionales, entonces sin duda alguna las políticas que mejor funcionan, como era de esperar, son el *Makespan* y el *Total Completion Time*. Por el contrario, si la prioridad es implementar la filosofía JIT, entonces las políticas que se aproximan más a la mejor solución son el *Total Tardiness + Earliness* y el *Number of Tardy + Early Jobs.*

Aun así, dentro de las políticas tradicionales se obtiene que la de *Total Completion Time* funciona mejor para un mayor número de objetivos que la de *Makespan.* De manera similar, respecto a las políticas JIT, la de *Total Tardiness + Earliness* proporciona generalmente mejores resultados que la de *Number of Tardy + Early Jobs.*

Finalmente destacar que, si el propósito se centra en minimizar los tres indicadores propuestos (tiempo ocioso, tiempo de espera y tiempo medio en inventario) entonces las políticas de producción tradicionales funcionan bastante mejor que las basadas en la filosofía JIT, sobre todo la de *Total Completion Time.*

Los valores ARDI se representan gráficamente a continuación, en la Figura 6.5.

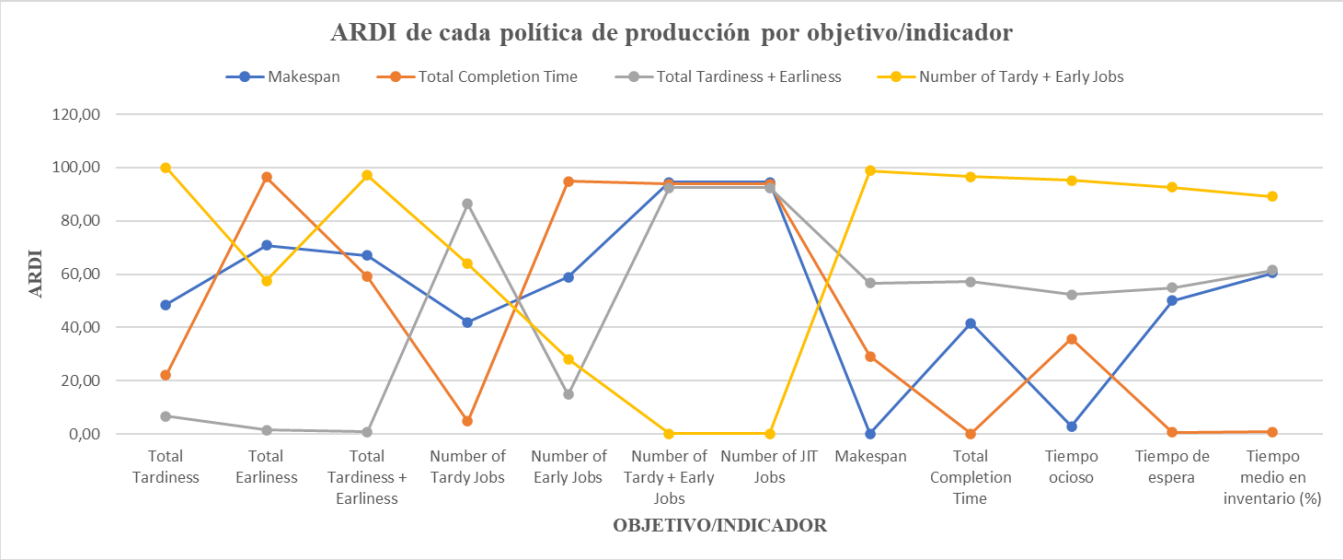


Figura 6.5. Gráfico ARDI de cada política de producción por objetivo/indicador [Fuente: Elaboración propia]

Asimismo, se representa en la Figura 6.6 la media del ARDI obtenida para cada política de producción, la cual se calcula realizando el promedio para todos los objetivos e indicadores.

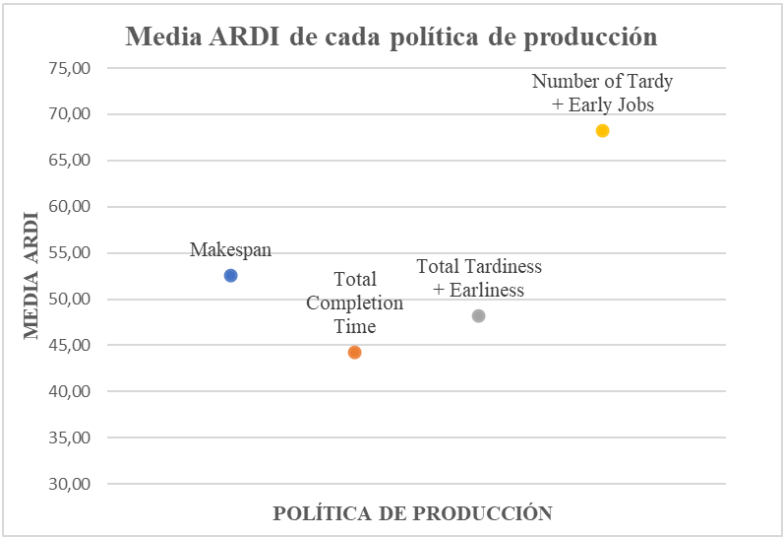


Figura 6.6. Gráfico media ARDI de cada política de producción [Fuente: Elaboración propia]

Si en vez de centrarse en cumplir unos objetivos u otros, el propósito consiste en buscar aquella política de producción que funcione mejor teniendo en cuenta todos ellos, tanto los tradicionales como los JIT, entonces se llega a la conclusión de que la política que se aproxima de manera general más a la mejor solución es la de *Total Completion Time*, ya que obtiene la menor media ARDI. Seguida de ella, con un valor similar de media, se encuentra la de *Total Tardiness + Earliness*. En tercer lugar, está la de *Makespan* y, por último, con una media bastante mayor que el resto, lo que implica que se aleja mucho de la mejor solución, está la de *Number of Tardy + Early Jobs.*

A continuación, se muestra en la Figura 6.7 el número de instancias para las que cada política obtiene la mejor solución respecto a cada objetivo e indicador.

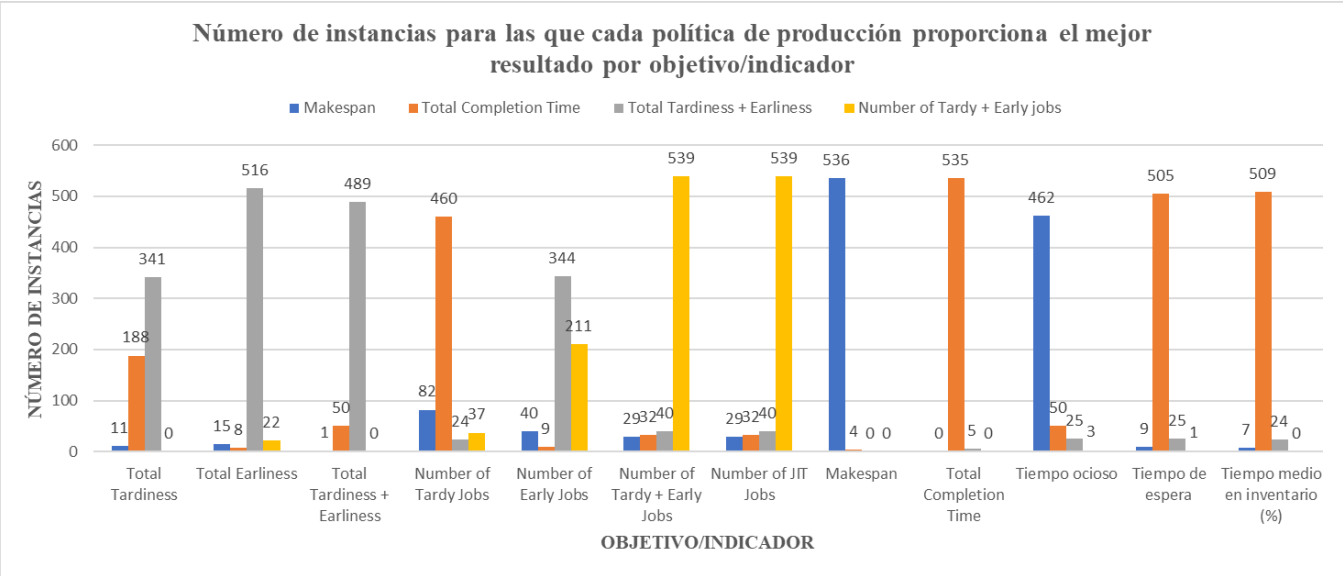


Figura 6.7. Gráfico número de instancias para las que cada política de producción proporciona el mejor resultado por objetivo/indicador [Fuente: Elaboración propia]

En cuanto al número de instancias, resaltar los valores tan dispares que se obtienen. Esto significa que, para cada objetivo/indicador existe una política de producción que funciona notablemente mejor que el resto, obteniendo la mejor solución para casi el 100% de las instancias. Sin embargo, una política puede conseguir destacar frente al resto para un objetivo/indicador en concreto, pero luego para otro no conseguir obtener la mejor solución para ninguna, o casi ninguna, de las instancias.

Con este gráfico se reafirma la conclusión alcanzada anteriormente: ninguna política funciona considerablemente mejor que el resto de manera general, sino que cada política destaca para uno o varios objetivos/indicadores específicos.

Aun así, si se realiza la suma para los doce objetivos/indicadores estudiados, se obtienen los siguientes resultados:

Makespan	Total Completion Time	Total Tardiness + Earliness	Number of Tardy + Early Jobs
1221	2382	1873	1352

Tabla 6.6. Número total de instancias para las que cada política de producción proporciona el mejor resultado

Al igual que ocurría para los valores de la media del ARDI, la política *Total Completion Time* resulta ser la que funciona mejor de manera general ya que alcanza el mejor resultado para un mayor número de instancias y, seguida de ella, se encuentra la de *Total Tardiness + Earliness*.

Sin embargo, ahora la política *Number of Tardy + Early Jobs* es la que está en tercer lugar y, la última es la de *Makespan*. Esto implica que la política *Number of Tardy + Early Jobs* alcanza la mejor solución para un mayor número de instancias que la de *Makespan*, pero que, cuando no lo alcanza, se aleja más del mejor valor que esta.

6.5.1 Análisis y comparación en función del tamaño de instancia

Además de comparar las políticas de producción de manera general, también se ha llevado a cabo un análisis en profundidad para cada tamaño de instancia *{n, m}* evaluado.

El anexo A contiene los gráficos de los valores de ARDI, así como los gráficos que muestran el número de instancias para las cuales se alcanza la mejor solución, de cada tamaño de instancia por separado. A continuación, se realiza un breve resumen de los resultados obtenidos y las conclusiones alcanzadas de manera general.

En primer lugar, en la Tabla 6.7 se exponen las medias del ARDI obtenidas para cada política de producción en función del tamaño de instancia. Estas medias se realizan promediando para los doce objetivos/indicadores estudiados.

Tamaño de instancia {*n, m*}	Makespan	Total Completion Time	Total Tardiness + Earliness	Number of Tardy + Early Jobs
50_10	53,68	46,06	48,60	68,03
50_30	46,51	40,07	38,80	69,02
50_50	41,41	33,92	32,81	66,96
150_10	55,45	46,25	52,31	66,30
150_30	53,51	44,26	47,75	70,04
150_50	53,60	45,03	45,45	70,35
250_10	55,57	46,50	54,66	65,13
250_30	53,20	45,88	50,42	69,66
250_50	54,65	45,33	48,64	69,53
350_10	55,85	46,53	55,32	65,47
350_30	53,52	46,34	52,05	69,27
350_50	54,20	44,96	50,92	69,39

Tabla 6.7. Valores media ARDI de cada política de producción en función del tamaño de instancia {*n, m*}

En relación con estos resultados, cabe destacar el hecho de que para cada política de producción se obtienen valores parecidos para todos los tamaños de instancia. Esto significa que las políticas funcionan de manera similar para todos los tamaños, es decir, se aproximan más o lo menos lo mismo a la mejor solución independientemente del número de trabajos *n* y del número de máquinas *m*.

De manera general, para casi todos los tamaños de instancia, la política que proporciona menores valores de media del ARDI resulta ser el *Total Completion Time*. En segundo lugar, se encuentra la política del *Total Tardiness + Earliness*, seguida del *Makespan* y, por último, el *Number of Tardy + Early Jobs*. Se alcanzan por tanto los mismos resultados que cuando se analizaron las políticas sin separar por tamaño de instancia.

Una última conclusión a la que se llega es que todas las políticas, excepto la de *Number of Tardy + Early Jobs,* alcanzan valores menores de media del ARDI, es decir, funcionan mejor, para el tamaño de instancia compuesto por 50 trabajos y 50 máquinas.

Estos valores se representan gráficamente en la Figura 6.8. Se puede observar que la línea asociada a la política *Total Completion Time* es la que se sitúa por debajo para todos los tamaños mientras que la asociada al *Number of Tardy + Early Jobs* está siempre por encima del todo.

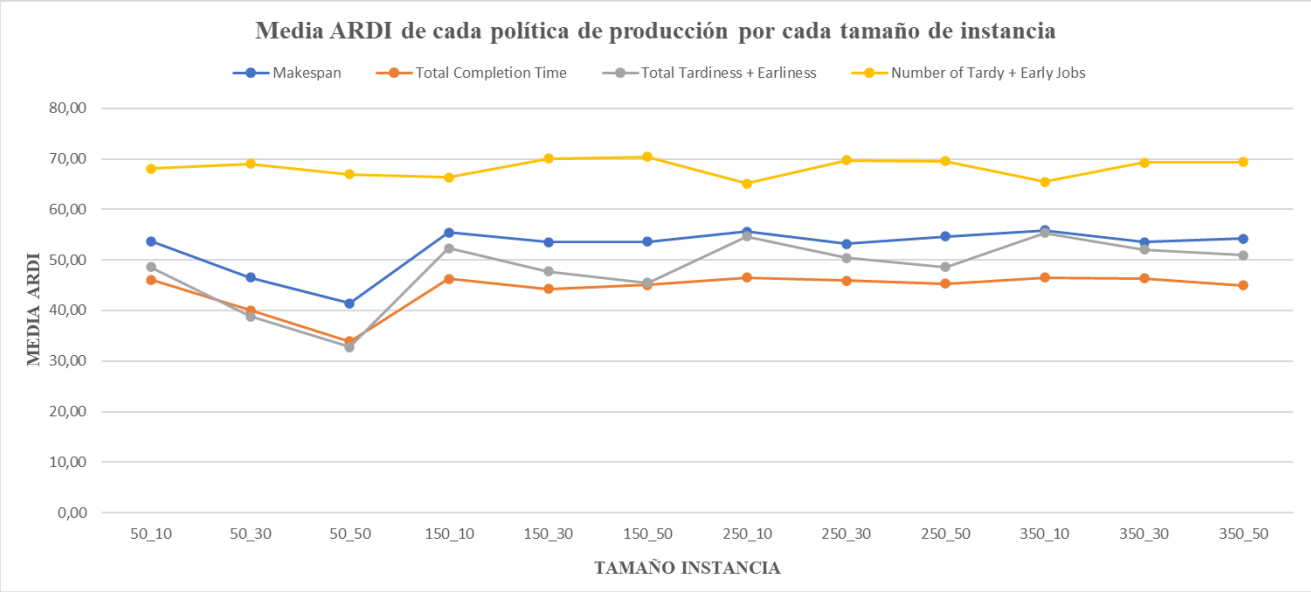


Figura 6.8. Gráfico media ARDI de cada política de producción por cada tamaño de instancia {*n, m*} [Fuente: Elaboración propia]

Respecto al número de instancias para las cuales cada política obtiene la mejor solución, en función de cada tamaño de instancia, se muestran en la Figura 6.9:

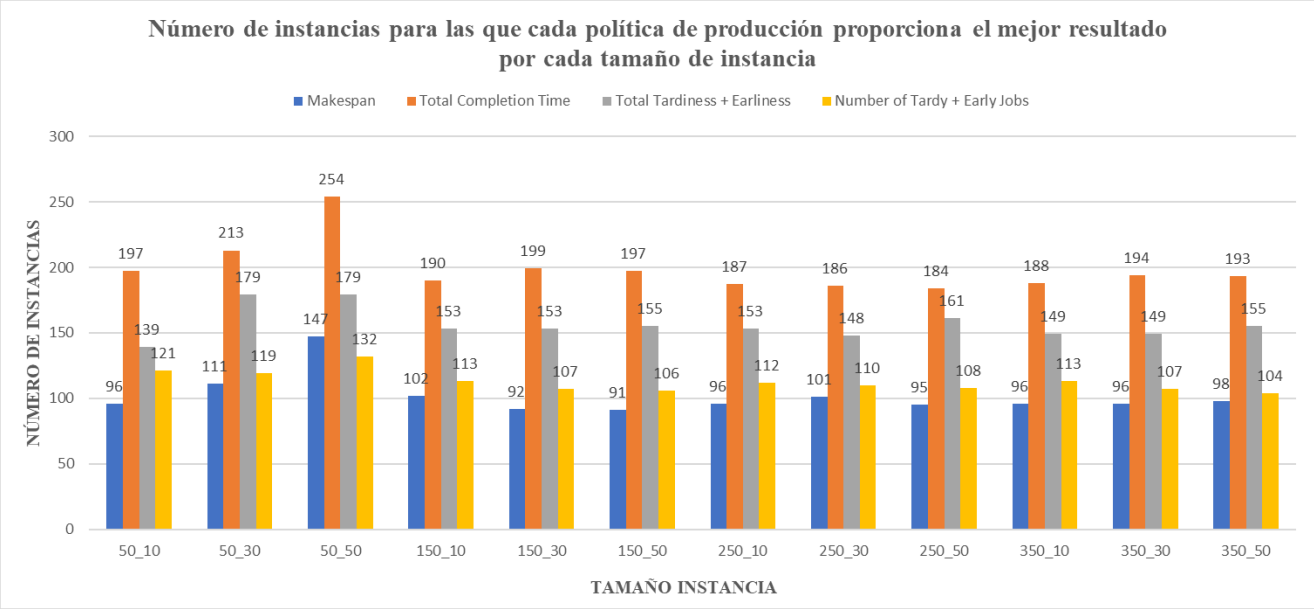


Figura 6.9. Gráfico número total de instancias para las que cada política de producción proporciona el mejor resultado por cada tamaño de instancia {*n, m*} [Fuente: Elaboración propia]

Observando este gráfico se llega a la misma conclusión que anteriormente: las políticas de producción funcionan de manera similar para todos los tamaños de instancia, siendo aquella cuyo objetivo es minimizar el *Total Completion Time* la que funciona mejor, independientemente del tamaño de la instancia, seguida de la de *Total Tardiness + Earliness*.

Sin embargo, resaltar que la política del *Number of Tardy + Early Jobs* alcanza para muchos tamaños de instancia el mismo o incluso mayor número de veces la mejor solución que la política del *Makespan*. Aún así, si se tiene en cuenta la media del ARDI, esta política proporciona mayores valores en todos los casos. Esto implica que, a pesar de obtener la mejor solución un número igual o mayor de veces, cuando no la obtiene su desviación respecto a esta es muy grande, es decir, se aleja mucho.

7 CONCLUSIONES

En el presente capítulo se exponen las conclusiones alcanzadas tras la realización del trabajo y obtención de los resultados, así como las futuras líneas de investigación posibles.

A la hora de abordar el problema objeto de estudio de este trabajo (*Permutation Flowshop Scheduling Problem*) se considera que, debido a su complejidad computacional, lo mejor es emplear métodos aproximados. En concreto se propone utilizar una metaheurística evolutiva, un algoritmo genético.

Respecto a la comparación entre ambas variantes, a la vista de los resultados obtenidos, se concluye que la que proporciona de manera general mejores resultados es la variante 1. En concreto para políticas tradicionales esta variante resulta funcionar considerablemente mejor que la segunda. Para políticas basadas en la filosofía JIT los resultados obtenidos no son tan dispares, aun así, la variante 1 sigue destacando frente a la 2. Una de las razones por la cual se sospecha que esto ocurre es por el mecanismo de búsqueda local introducido en la segunda variante. Dicho mecanismo requiere de un gran esfuerzo computacional, mayor cuánto mayor es el tamaño de la instancia. Por lo tanto, para instancias con un gran número de trabajos y máquinas, el algoritmo dedica una considerable cantidad de tiempo en llevar a cabo dicha búsqueda local, resultando en una disminución significativa de las iteraciones efectuadas por el algoritmo genético. Debido a esto, a medida que el tamaño de la instancia aumenta, el algoritmo tiende a convertirse en una búsqueda local en lugar de un verdadero algoritmo genético.

Teniendo esto en cuenta, se decide utilizar la primera variante para obtener los resultados del segundo análisis, dónde se comparan cuatro políticas de producción diferentes. En cuanto a este segundo análisis, se alcanza la conclusión de que ninguna política destaca entre las demás, sino que el funcionamiento de cada una depende del objetivo/indicador que se esté considerando. Si el propósito es cumplir de la mejor manera posible con los objetivos de producción tradicionales, entonces la política que mejor funciona es la de *Total Completion Time.* Por el contrario, si la prioridad es implementar la filosofía JIT, entonces la política que más se aproxima a la mejor solución es la de *Total Tardiness + Earliness.* Las políticas de *Makespan* y *Number of Tardy + Early Jobs* son las que peor funcionan de manera general.

También cabe destacar que, a la hora de minimizar los tres indicadores propuestos (tiempo ocioso, tiempo de espera y tiempo medio en inventario) las políticas de producción tradicionales funcionan bastante mejor que las basadas en la filosofía JIT. En concreto, la política de *Makespan* da lugar a una minimización del tiempo ocioso mientras que la política de *Total Completion Time* minimiza el tiempo de espera y el tiempo medio en inventario.

Por otra parte, si en vez de centrarse en unos objetivos u otros, se busca determinar la política de producción que funcione mejor teniendo en cuenta todos ellos, tanto los tradicionales como los JIT, entonces se llega a la conclusión de que la política que proporciona de manera general mejores resultados es la de *Total Completion Time.*

Otra conclusión alcanzada en este segundo análisis es que, para cada objetivo e indicador existe una política de producción que funciona notablemente mejor que el resto, obteniendo la mejor solución para casi el 100% de las instancias. Por tanto, si se busca cumplir con un solo objetivo o indicador, la elección es clara.

Se lleva también a cabo una comparativa de las distintas políticas de producción en función del tamaño de instancia. Se concluye que las políticas funcionan de manera similar para todos los tamaños, es decir, se aproximan más o lo menos lo mismo a la mejor solución independientemente del número de trabajos n y del número de máquinas m. Además, se obtiene que todas las políticas, excepto la de *Number of Tardy + Early Jobs,* funcionan mejor para el tamaño de instancia compuesto por 50 trabajos y 50 máquinas.

Finalmente, respecto a las posibles líneas de investigación, se podrían desarrollar mejoras en ambas variantes que permitiesen alcanzar mejores soluciones y/o disminuir el esfuerzo computacional requerido. Existen diversas modificaciones posibles que se podrían realizar, empezando por variar los operadores genéticos, ya que existe un gran número de ellos en la literatura. También se pueden añadir mecanismos como el *Path relinking*, que es una técnica de búsqueda en la cual el objetivo es explorar el espacio de búsqueda o "ruta" entre un conjunto dado de buenas soluciones, o incluso incluir procedimientos que aceleren el algoritmo. Otra posible línea de investigación interesante consistiría en estudiar funciones *fitness* multivariables, es decir, que tengan en cuenta al mismo tiempo varios objetivos e incluso adaptarlas a restricciones específicas del problema.

REFERENCIAS

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

Pérez González, Framiñán Torres, J. M., & Fernández-Viagas Escudero, V. (2021). Programación de operaciones : 4º del Grado de Ingeniería de Tecnologías Industriales. Universidad de Sevilla. Escuela Técnica Superior de Ingeniería. Sección de Publicaciones ETSI.

Navarro García, & Fernández-Viagas Escudero, V. (2020). Taller de flujo regular con tiempos de cambio dependientes de la secuencia: heurísticas constructivas basadas en la memoria Trabajo Fin de Grado. El autor.

Fernández-Viagas Escudero, & Framiñán Torres, J. M. (2016). The permutation flowshop scheduling problem analysis, solution procedures and problem extensions. [s.n.].

Pinedo. (2016). Scheduling Theory, Algorithms, and Systems (5th ed. 2016.). Springer International Publishing. https://doi.org/10.1007/978-3-319-26580-3

Lopez, & Roubellat, F. (2008). Production scheduling. ISTE.

Baker, & Trietsch, D. (2019). Principles of sequencing and scheduling (Second edition.). Wiley.

Jarboui, Siarry, P., Teghem, J., & Bourrieres, J.-P. (2013). Metaheuristics for production scheduling. ISTE.

Rios, & Ríos-Solís, Y. A. (Eds.). (2012). Just-in-Time Systems (1st ed. 2012.). Springer New York. https://doi.org/10.1007/978-1-4614-1123-9

Jozefowska, & Józefowska, J. (2007). Just-in-Time Scheduling Models and Algorithms for Computer and Manufacturing Systems (1st ed. 2007.). Springer US. https://doi.org/10.1007/978-0-387-71718-0

Rabadi (Ed.). (2016). Heuristics, Metaheuristics and Approximate Methods in Planning and Scheduling (1st ed. 2016.). Springer International Publishing. https://doi.org/10.1007/978-3-319-26024-2

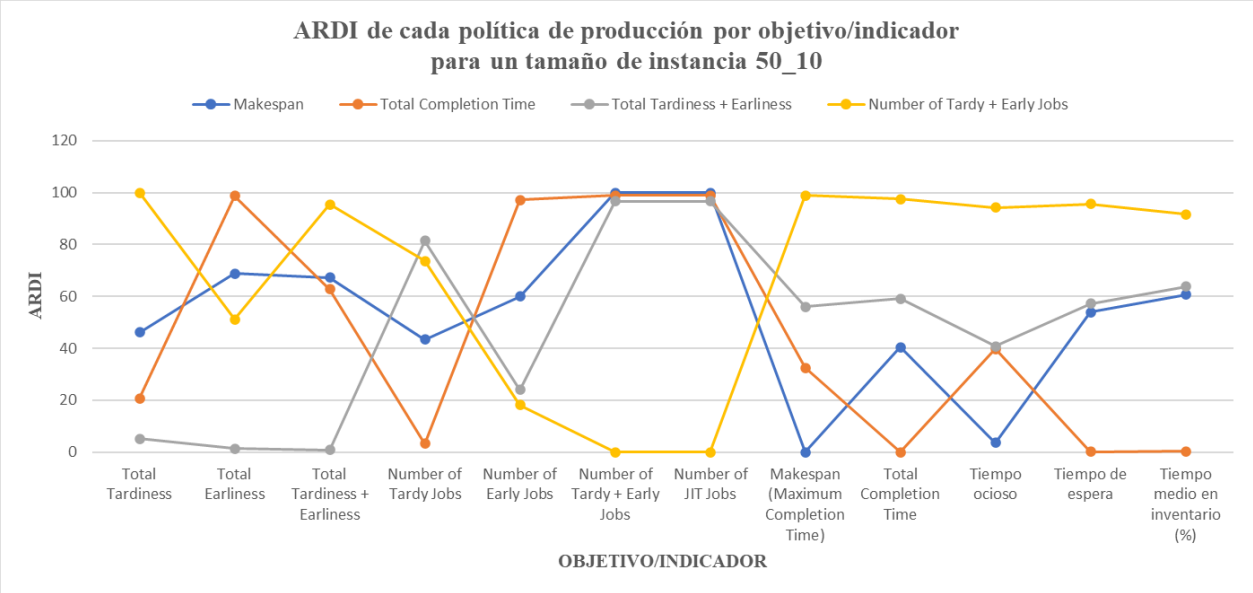
Sivanandam, & Deepa, S. N. (2008). Introduction to Genetic Algorithms. Springer. https://doi.org/10.1007/978-3-540-73190-0

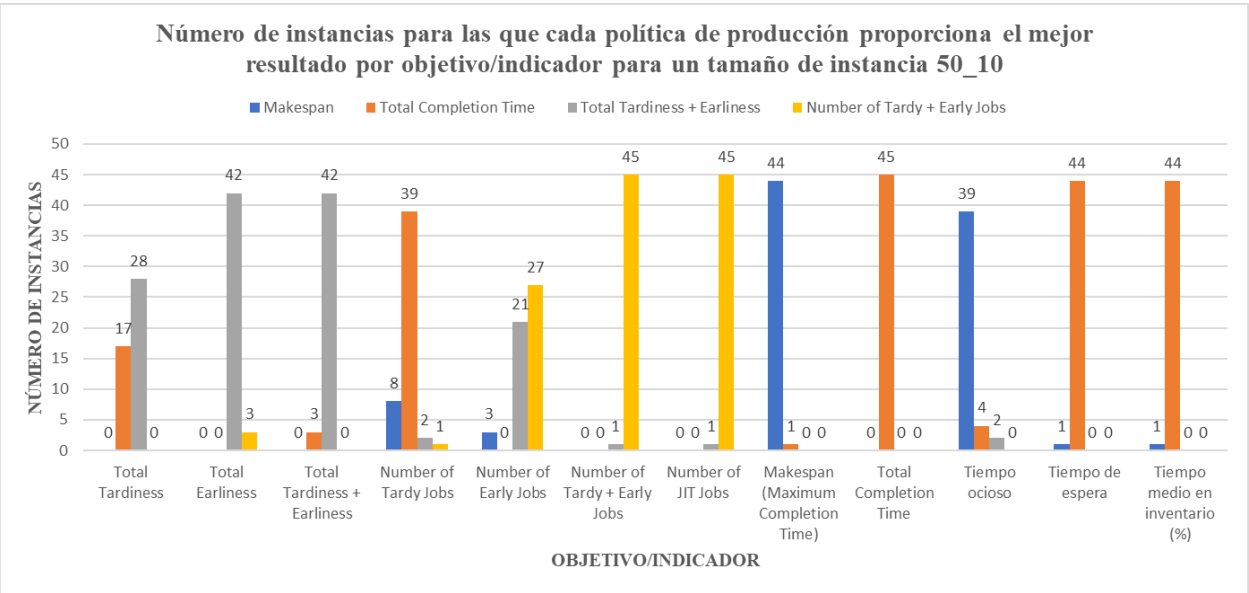
Eva Vallada, Rubén Ruiz. (2010). Genetic algorithms with path relinking for the minimum tardiness permutation flowshop problem, Omega, Volume 38, Issues 1–2, Pages 57-67, ISSN 0305-0483. https://doi.org/10.1016/j.omega.2009.04.002.

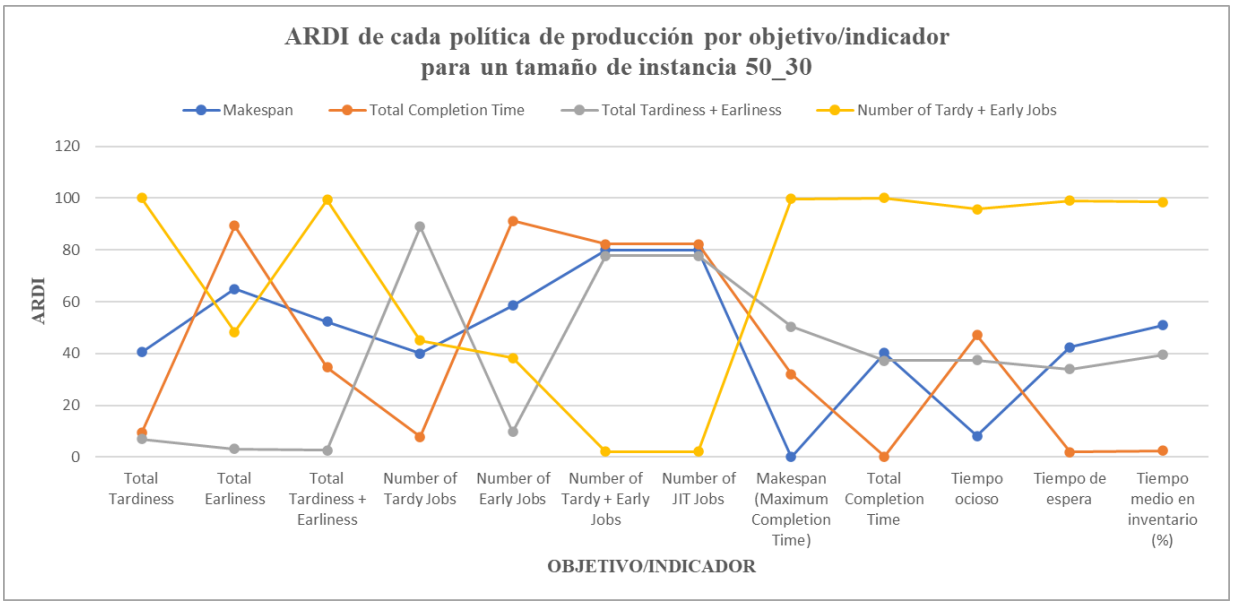
Eva Vallada, Rubén Ruiz, Gerardo Minella. (2008). Minimising total tardiness in the m-machine flowshop problem: A review and evaluation of heuristics and metaheuristics, Computers & Operations Research, Volume 35, Issue 4, Pages 1350-1373, ISSN 0305-0548. https://doi.org/10.1016/j.cor.2006.08.016.

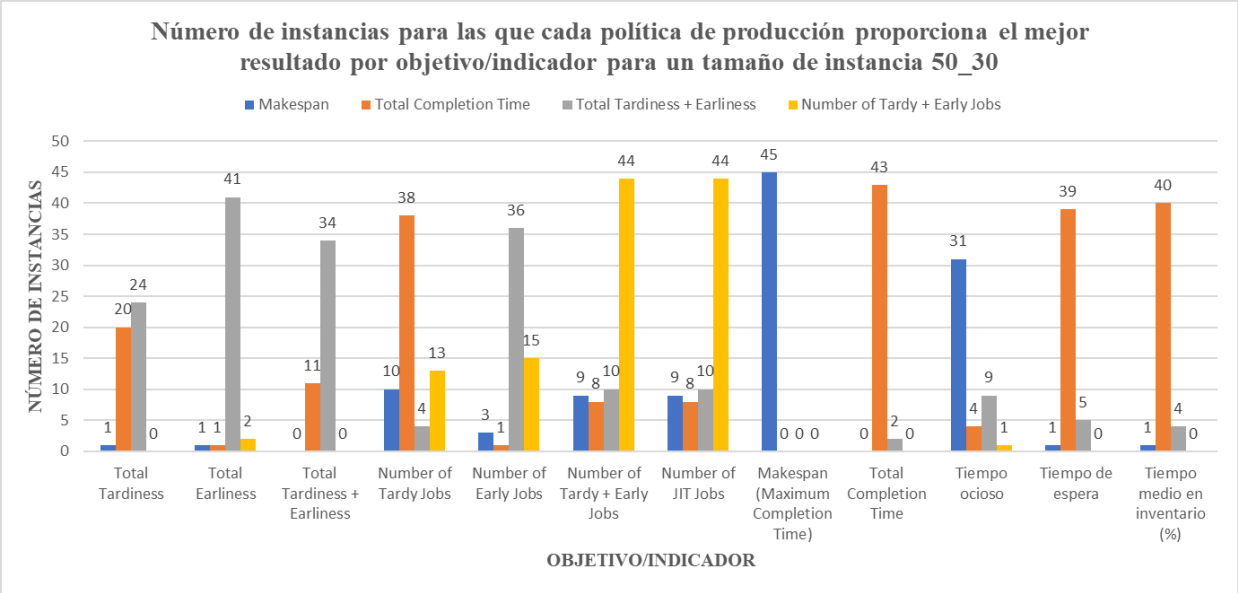
Cairó Battistutti. (2006). Metodología de la programación : algoritmos, diagramas de flujo y programas / (3 ed., [3 reimp.]). Alfaomega,.

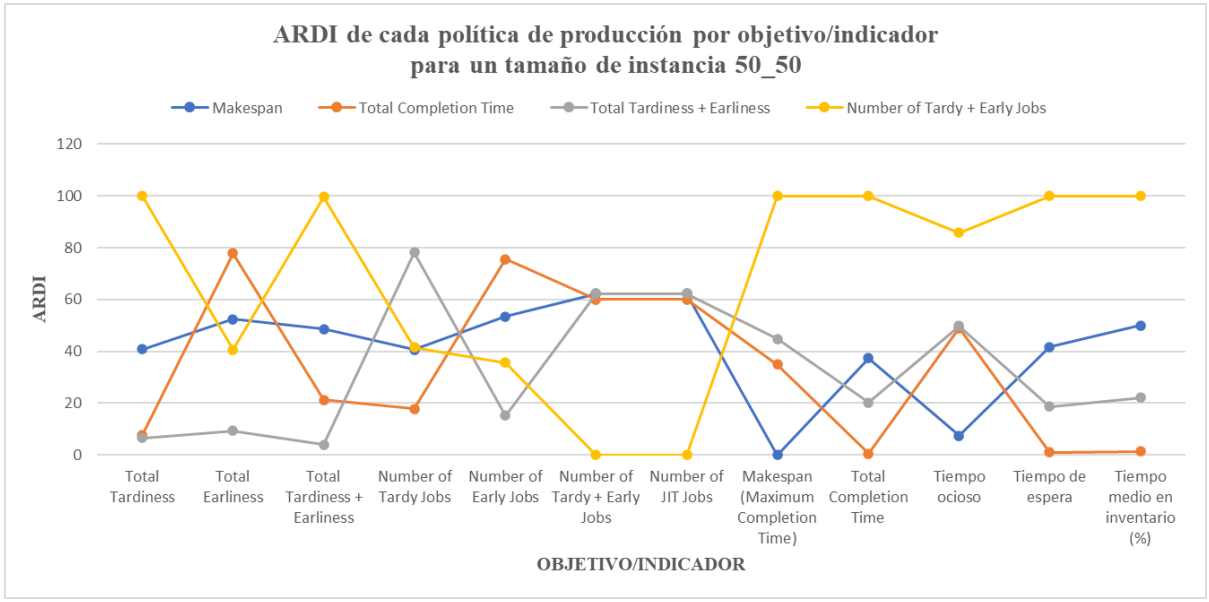
ANEXO A: Gráficas con los resultados obtenidos para cada tamaño de instancia

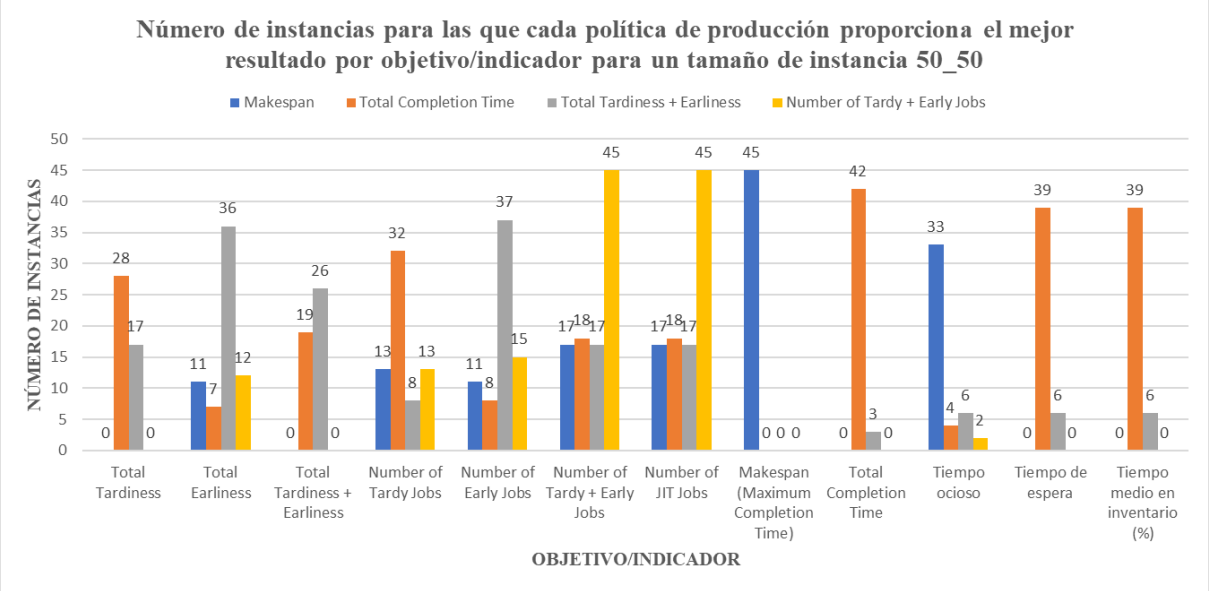


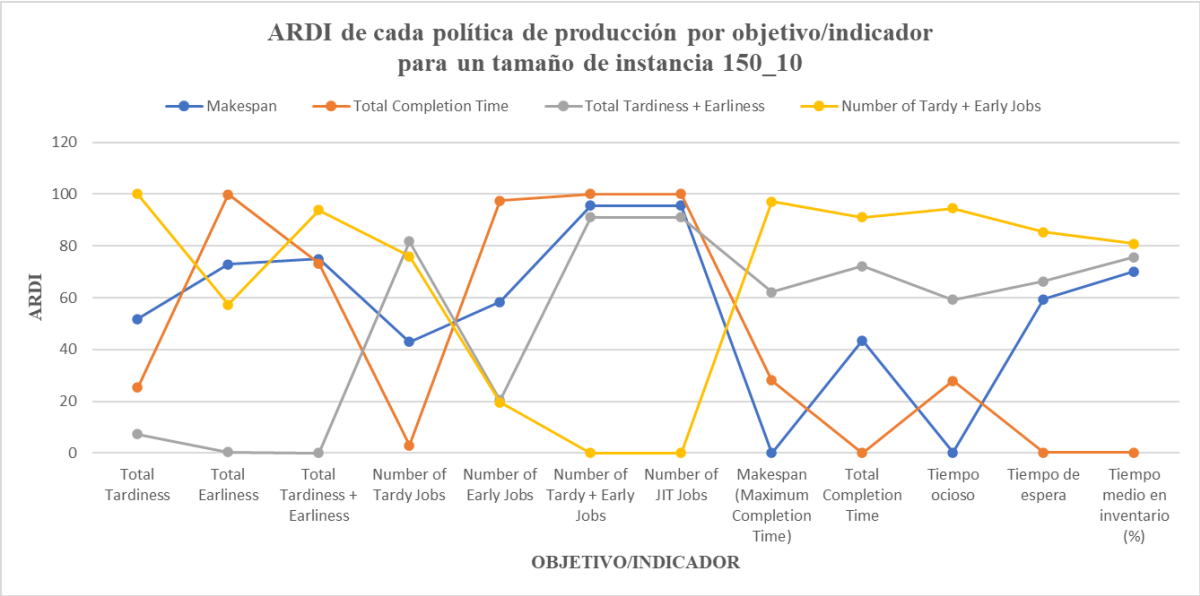


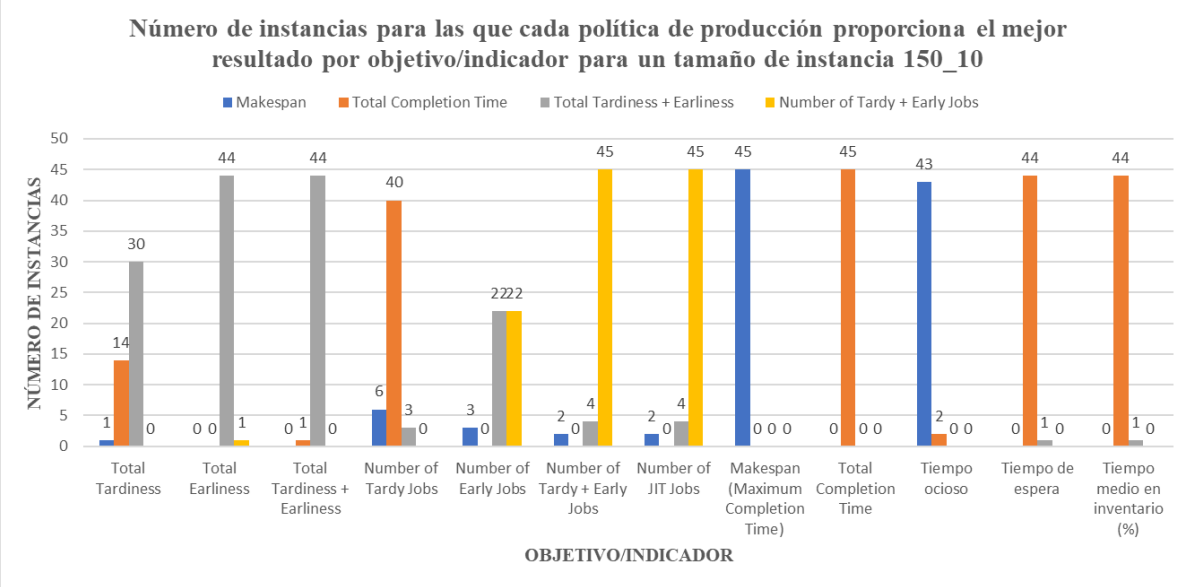

ARDI de cada política de producción por objetivo/indicador para un tamaño de instancia 50_30
Makespan
Total Completion Time
Total Tardiness + Earliness
Number of Tardy + Early Jobs
ARDI
120
100
80
60
40
20
0
Total Tardiness
Total Earliness
Total Tardiness + Earliness
Number of Tardy Jobs
Number of Early Jobs
Number of Tardy + Early Jobs
Number of JIT Jobs
Makespan (Maximum Completion Time)
Total Completion Time
Tiempo ocioso
Tiempo de espera
Tiempo medio en inventario (%)
OBJETIVO/INDICADOR

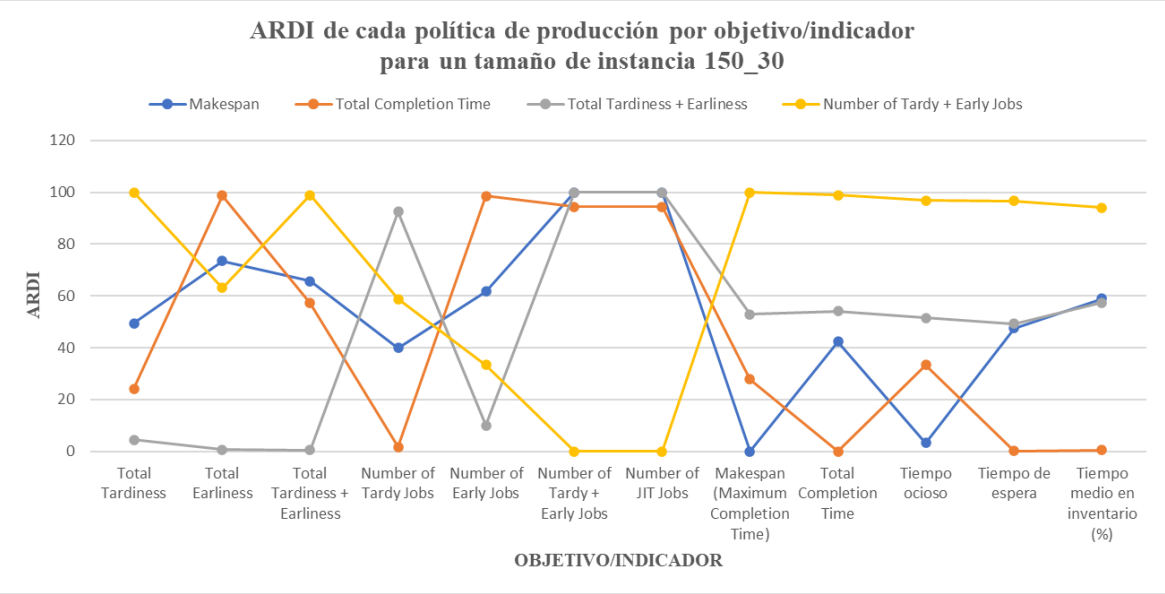

Número de instancias para las que cada política de producción proporciona el mejor resultado por objetivo/indicador para un tamaño de instancia 50_30
Makespan
Total Completion Time
Total Tardiness + Earliness
Number of Tardy + Early Jobs
NÚMERO DE INSTANCIAS
50
45
40
35
30
25
20
15
10
5
0
1 20 24
1 1 41 2
0 11 34 0
10 38 4 13
3 1 36 15
9 8 10 44
9 8 10 44
45 0 0 0
0 43 2 0
31 4 9 1
1 39 5 0
1 40 4 0
Total Tardiness
Total Earliness
Total Tardiness + Earliness
Number of Tardy Jobs
Number of Early Jobs
Number of Tardy + Early Jobs
Number of JIT Jobs
Makespan (Maximum Completion Time)
Total Completion Time
Tiempo ocioso
Tiempo de espera
Tiempo medio en inventario (%)
OBJETIVO/INDICADOR

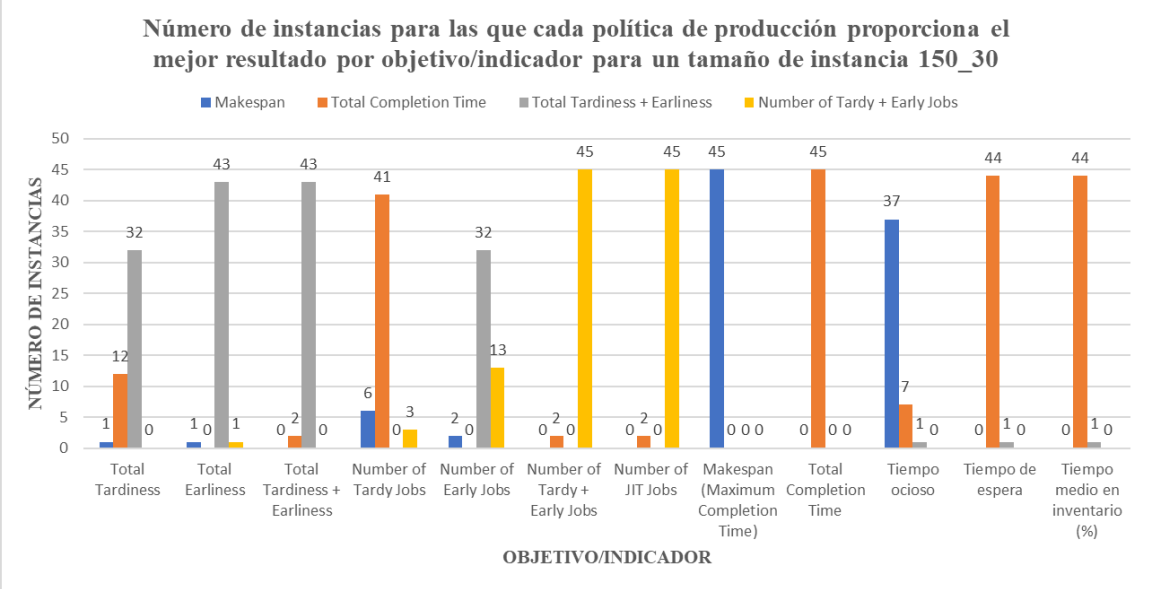

ARDI de cada política de producción por objetivo/indicador para un tamaño de instancia 50_50
Makespan
Total Completion Time
Total Tardiness + Earliness
Number of Tardy + Early Jobs
ARDI
120
100
80
60
40
20
0
Total Tardiness
Total Earliness
Total Tardiness + Earliness
Number of Tardy Jobs
Number of Early Jobs
Number of Tardy + Early Jobs
Number of JIT Jobs
Makespan (Maximum Completion Time)
Total Completion Time
Tiempo ocioso
Tiempo de espera
Tiempo medio en inventario (%)
OBJETIVO/INDICADOR

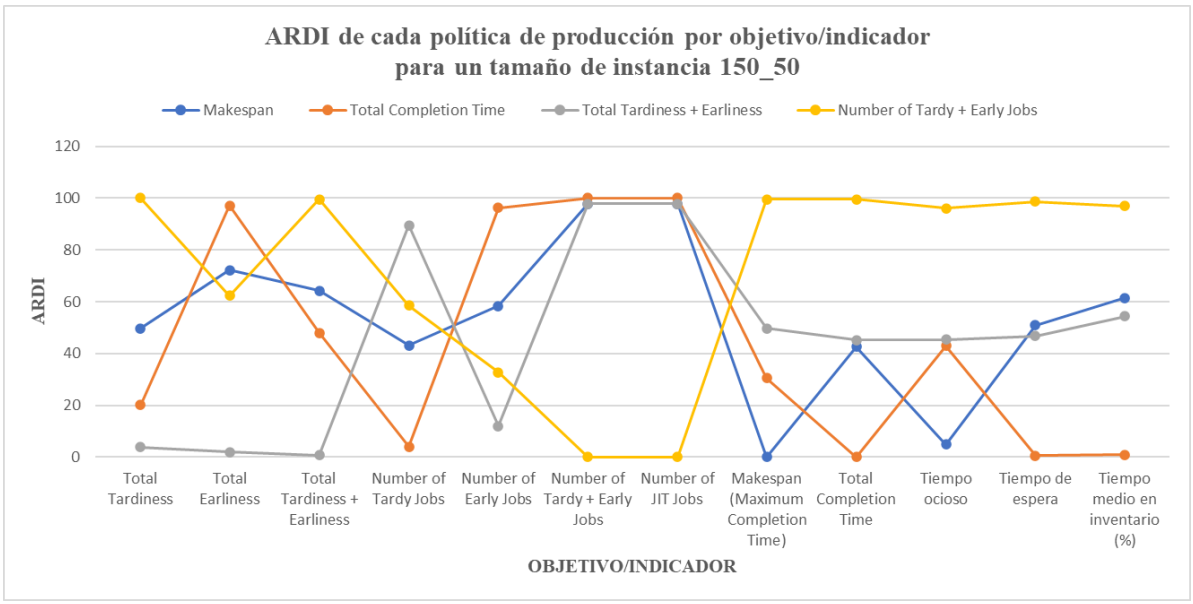

Número de instancias para las que cada política de producción proporciona el mejor resultado por objetivo/indicador para un tamaño de instancia 50_50
Makespan
Total Completion Time
Total Tardiness + Earliness
Number of Tardy + Early Jobs
NÚMERO DE INSTANCIAS
50
45
40
35
30
25
20
15
10
5
0
Total Tardiness
Total Earliness
Total Tardiness + Earliness
Number of Tardy Jobs
Number of Early Jobs
Number of Tardy + Early Jobs
Number of JIT Jobs
Makespan (Maximum Completion Time)
Total Completion Time
Tiempo ocioso
Tiempo de espera
Tiempo medio en inventario (%)
OBJETIVO/INDICADOR

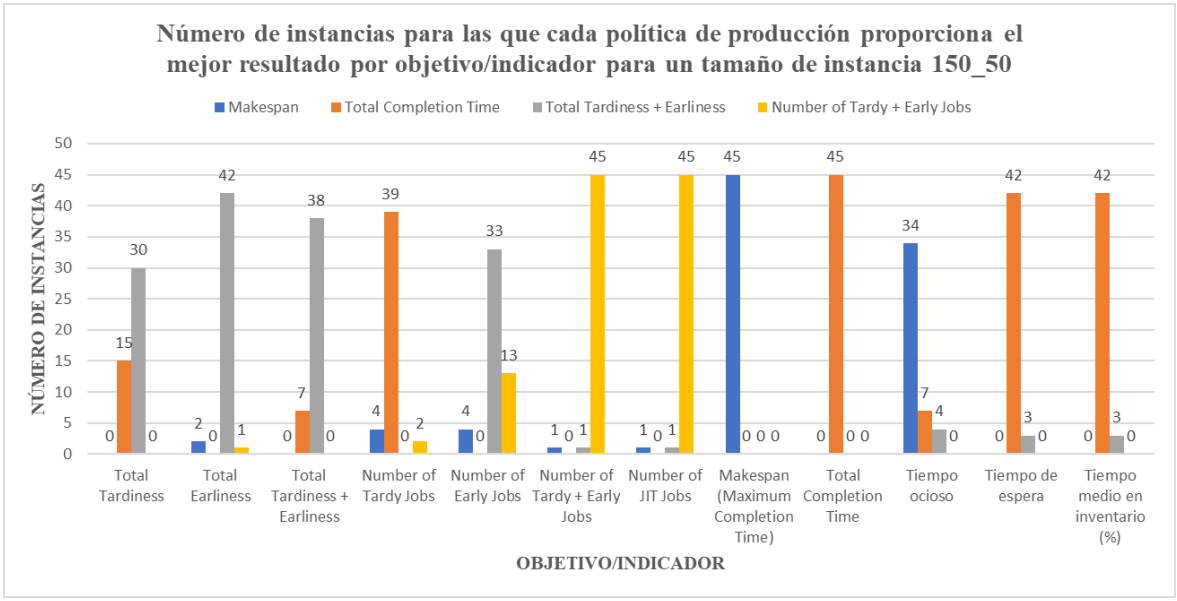

ARDI de cada política de producción por objetivo/indicador para un tamaño de instancia 150_10
Makespan
Total Completion Time
Total Tardiness + Earliness
Number of Tardy + Early Jobs
ARDI
120
100
80
60
40
20
0
Total Tardiness
Total Earliness
Total Tardiness + Earliness
Number of Tardy Jobs
Number of Early Jobs
Number of Tardy + Early Jobs
Number of JIT Jobs
Makespan (Maximum Completion Time)
Total Completion Time
Tiempo ocioso
Tiempo de espera
Tiempo medio en inventario (%)
OBJETIVO/INDICADOR

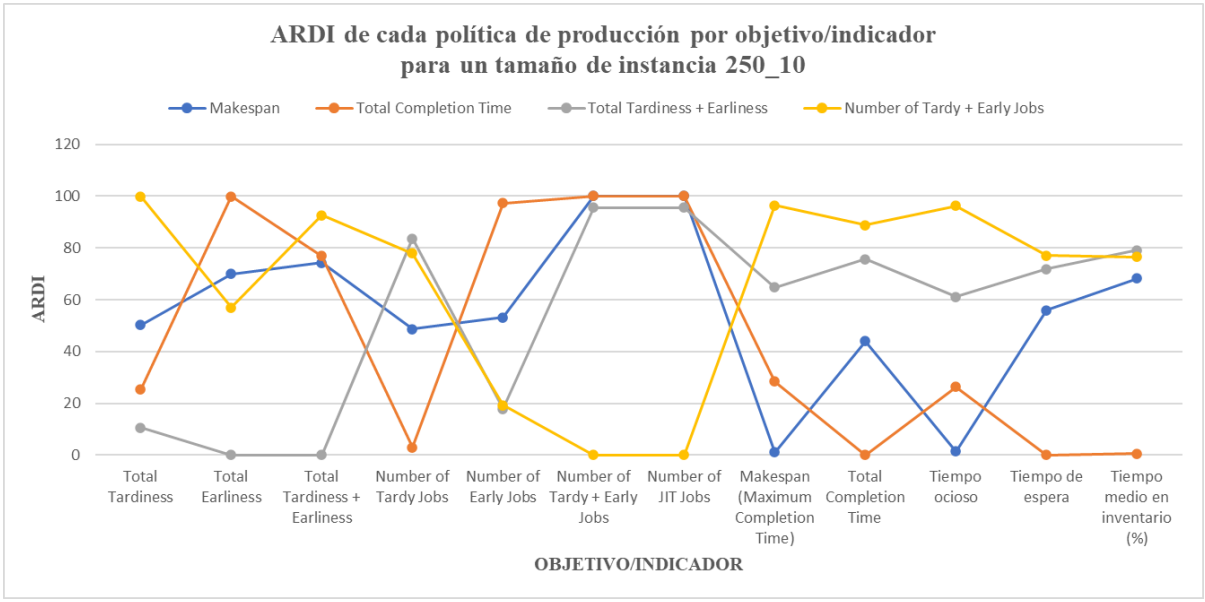

Número de instancias para las que cada política de producción proporciona el mejor resultado por objetivo/indicador para un tamaño de instancia 150_10
Makespan
Total Completion Time
Total Tardiness + Earliness
Number of Tardy + Early Jobs
NÚMERO DE INSTANCIAS
50
45
40
35
30
25
20
15
10
5
0
Total Tardiness
Total Earliness
Total Tardiness + Earliness
Number of Tardy Jobs
Number of Early Jobs
Number of Tardy + Early Jobs
Number of JIT Jobs
Makespan (Maximum Completion Time)
Total Completion Time
Tiempo ocioso
Tiempo de espera
Tiempo medio en inventario (%)
OBJETIVO/INDICADOR

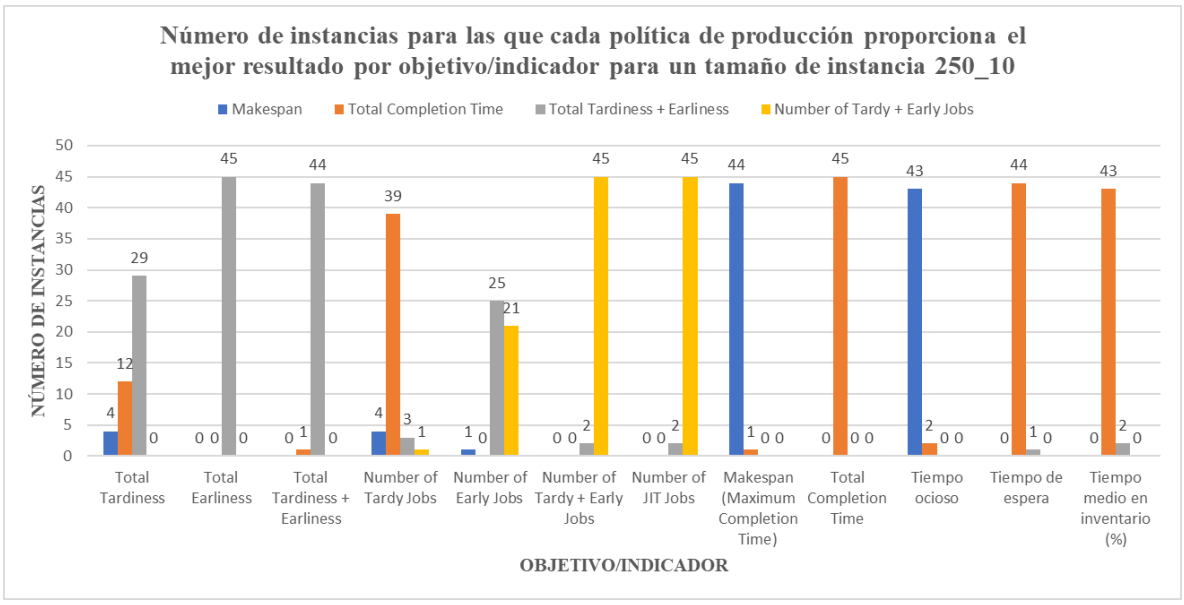

ARDI de cada política de producción por objetivo/indicador para un tamaño de instancia 150_30
Makespan
Total Completion Time
Total Tardiness + Earliness
Number of Tardy + Early Jobs
ARDI
120
100
80
60
40
20
0
Total Tardiness
Total Earliness
Total Tardiness + Earliness
Number of Tardy Jobs
Number of Early Jobs
Number of Tardy + Early Jobs
Number of JIT Jobs
Makespan (Maximum Completion Time)
Total Completion Time
Tiempo ocioso
Tiempo de espera
Tiempo medio en inventario (%)
OBJETIVO/INDICADOR

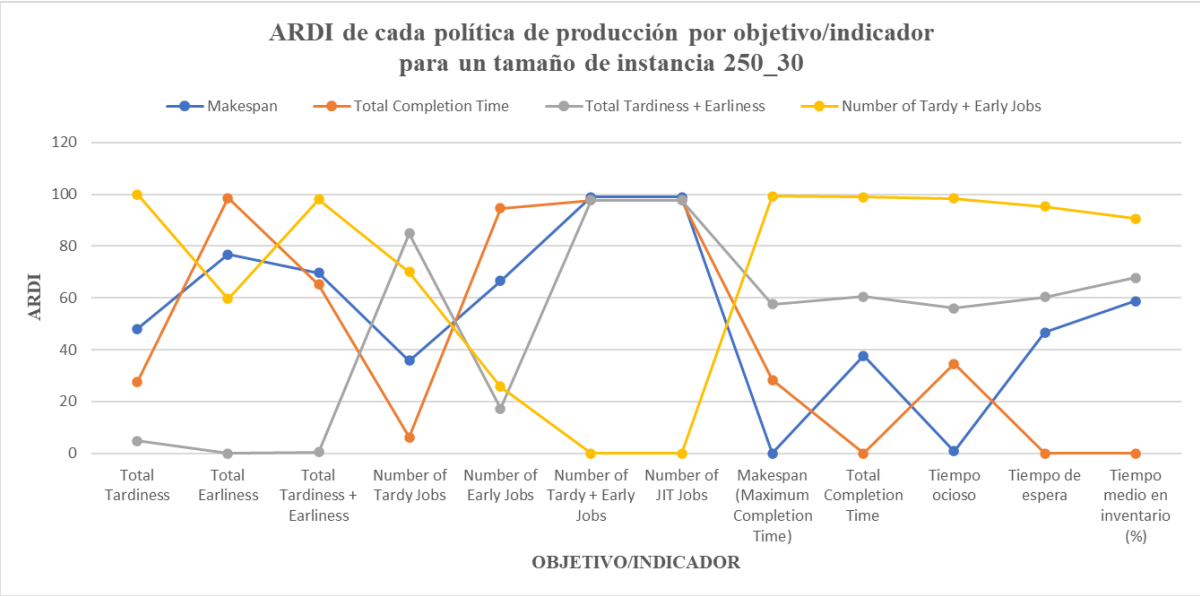

Número de instancias para las que cada política de producción proporciona el mejor resultado por objetivo/indicador para un tamaño de instancia 150_30
Makespan
Total Completion Time
Total Tardiness + Earliness
Number of Tardy + Early Jobs
NÚMERO DE INSTANCIAS
50
45
40
35
30
25
20
15
10
5
0
Total Tardiness
Total Earliness
Total Tardiness + Earliness
Number of Tardy Jobs
Number of Early Jobs
Number of Tardy + Early Jobs
Number of JIT Jobs
Makespan (Maximum Completion Time)
Total Completion Time
Tiempo ocioso
Tiempo de espera
Tiempo medio en inventario (%)
OBJETIVO/INDICADOR


ARDI de cada política de producción por objetivo/indicador para un tamaño de instancia 150_50
Makespan
Total Completion Time
Total Tardiness + Earliness
Number of Tardy + Early Jobs
ARDI
120
100
80
60
40
20
0
Total Tardiness
Total Earliness
Total Tardiness + Earliness
Number of Tardy Jobs
Number of Early Jobs
Number of Tardy + Early Jobs
Number of JIT Jobs
Makespan (Maximum Completion Time)
Total Completion Time
Tiempo ocioso
Tiempo de espera
Tiempo medio en inventario (%)
OBJETIVO/INDICADOR

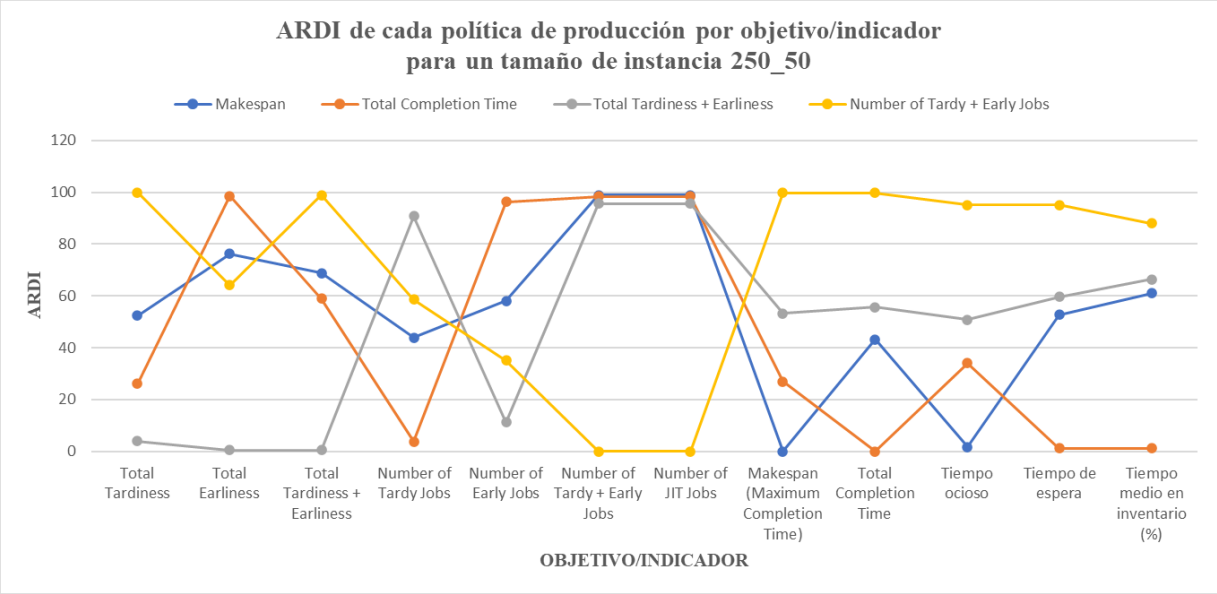

Número de instancias para las que cada política de producción proporciona el mejor resultado por objetivo/indicador para un tamaño de instancia 150_50
Makespan
Total Completion Time
Total Tardiness + Earliness
Number of Tardy + Early Jobs
NÚMERO DE INSTANCIAS
50
45
40
35
30
25
20
15
10
5
0
Total Tardiness
Total Earliness
Total Tardiness + Earliness
Number of Tardy Jobs
Number of Early Jobs
Number of Tardy + Early Jobs
Number of JIT Jobs
Makespan (Maximum Completion Time)
Total Completion Time
Tiempo ocioso
Tiempo de espera
Tiempo medio en inventario (%)
OBJETIVO/INDICADOR

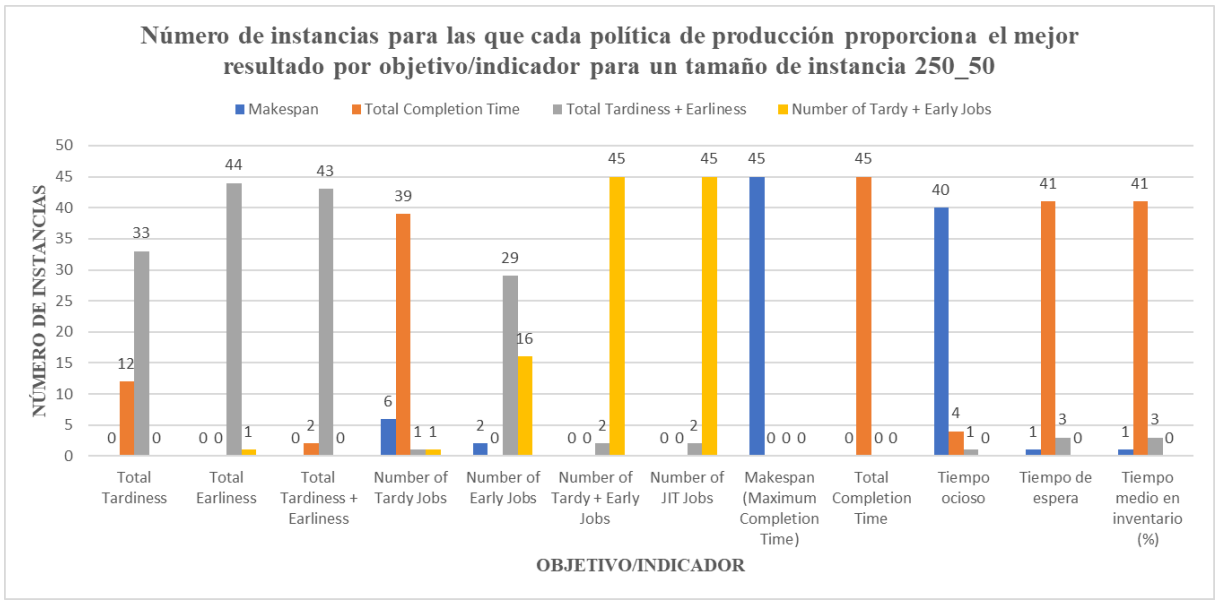

ARDI de cada política de producción por objetivo/indicador para un tamaño de instancia 250_10
Makespan
Total Completion Time
Total Tardiness + Earliness
Number of Tardy + Early Jobs
ARDI
120
100
80
60
40
20
0
Total Tardiness
Total Earliness
Total Tardiness + Earliness
Number of Tardy Jobs
Number of Early Jobs
Number of Tardy + Early Jobs
Number of JIT Jobs
Makespan (Maximum Completion Time)
Total Completion Time
Tiempo ocioso
Tiempo de espera
Tiempo medio en inventario (%)
OBJETIVO/INDICADOR

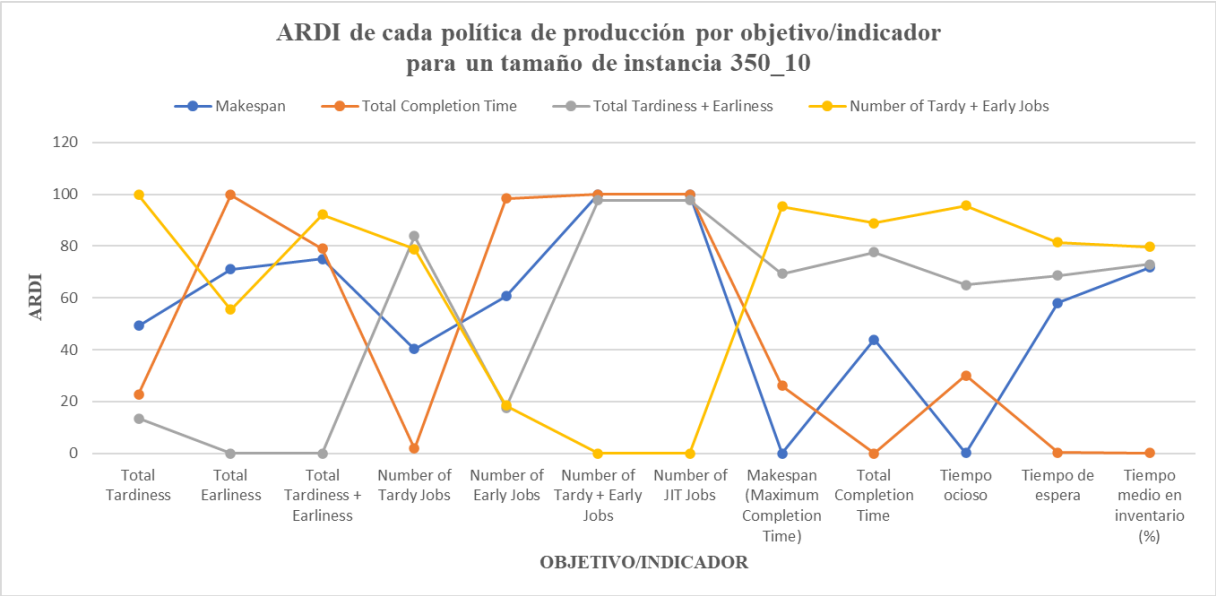

Número de instancias para las que cada política de producción proporciona el mejor resultado por objetivo/indicador para un tamaño de instancia 250_10
Makespan
Total Completion Time
Total Tardiness + Earliness
Number of Tardy + Early Jobs
NÚMERO DE INSTANCIAS
50
45
40
35
30
25
20
15
10
5
0
Total Tardiness
Total Earliness
Total Tardiness + Earliness
Number of Tardy Jobs
Number of Early Jobs
Number of Tardy + Early Jobs
Number of JIT Jobs
Makespan (Maximum Completion Time)
Total Completion Time
Tiempo ocioso
Tiempo de espera
Tiempo medio en inventario (%)
OBJETIVO/INDICADOR

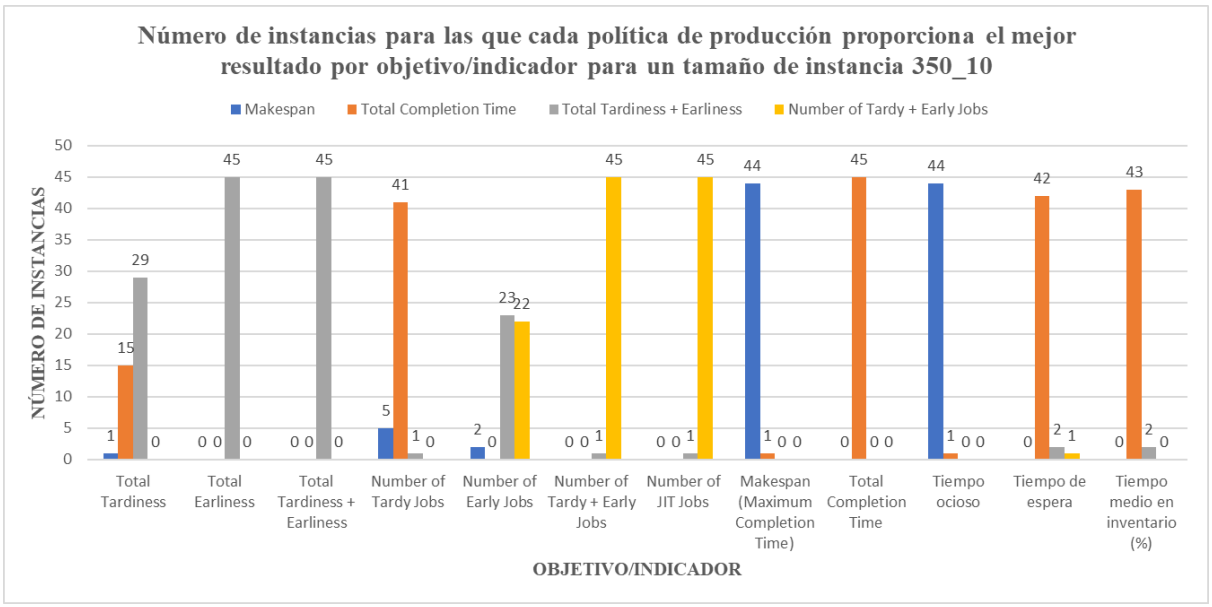

ARDI de cada política de producción por objetivo/indicador para un tamaño de instancia 250_30
Makespan
Total Completion Time
Total Tardiness + Earliness
Number of Tardy + Early Jobs
ARDI
120
100
80
60
40
20
0
Total Tardiness
Total Earliness
Total Tardiness + Earliness
Number of Tardy Jobs
Number of Early Jobs
Number of Tardy + Early Jobs
Number of JIT Jobs
Makespan (Maximum Completion Time)
Total Completion Time
Tiempo ocioso
Tiempo de espera
Tiempo medio en inventario (%)
OBJETIVO/INDICADOR

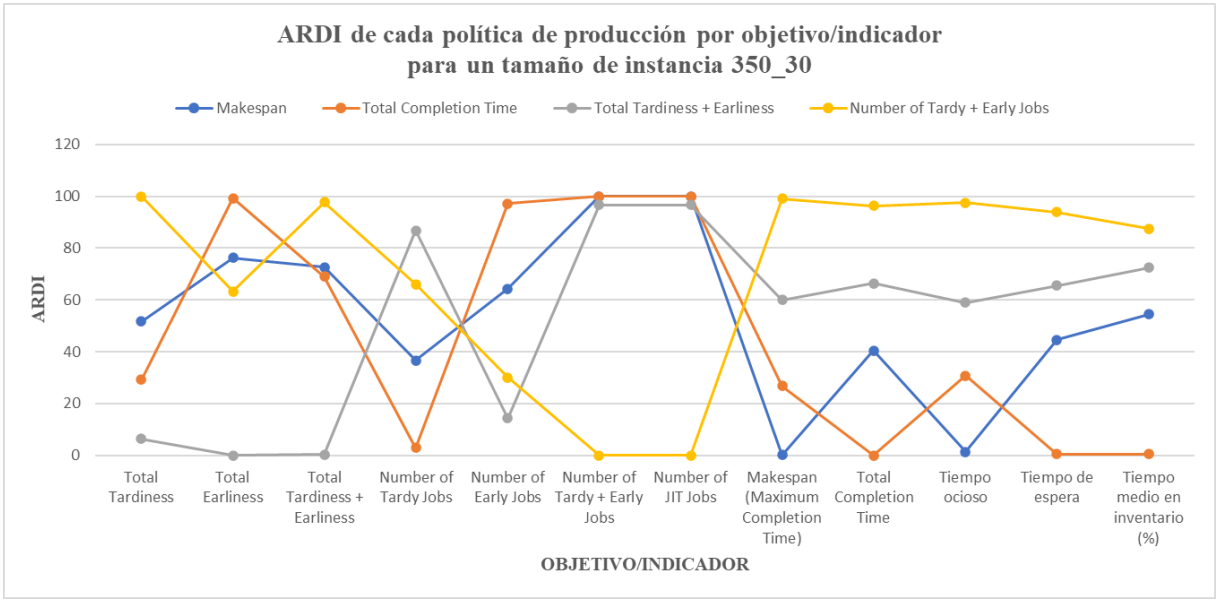

Número de instancias para las que cada política de producción proporciona el mejor resultado por objetivo/indicador para un tamaño de instancia 250_30
Makespan
Total Completion Time
Total Tardiness + Earliness
Number of Tardy + Early Jobs
NÚMERO DE INSTANCIAS
50
45
40
35
30
25
20
15
10
5
0
Total Tardiness
Total Earliness
Total Tardiness + Earliness
Number of Tardy Jobs
Number of Early Jobs
Number of Tardy + Early Jobs
Number of JIT Jobs
Makespan (Maximum Completion Time)
Total Completion Time
Tiempo ocioso
Tiempo de espera
Tiempo medio en inventario (%)
OBJETIVO/INDICADOR

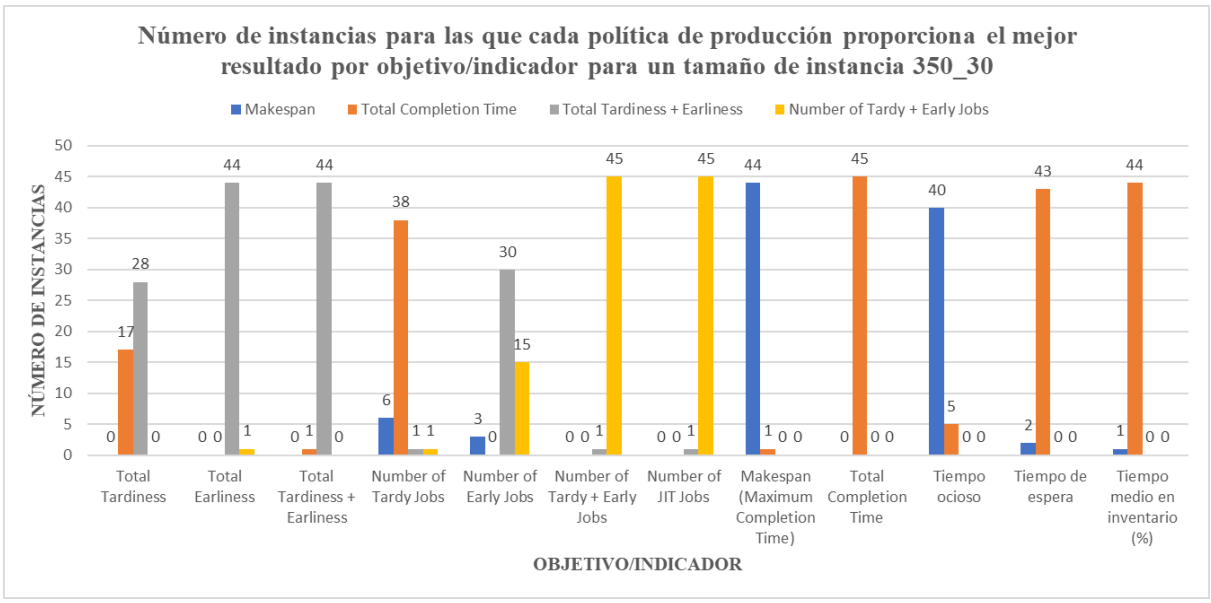

ARDI de cada política de producción por objetivo/indicador para un tamaño de instancia 250_50
Makespan
Total Completion Time
Total Tardiness + Earliness
Number of Tardy + Early Jobs
ARDI
120
100
80
60
40
20
0
Total Tardiness
Total Earliness
Total Tardiness + Earliness
Number of Tardy Jobs
Number of Early Jobs
Number of Tardy + Early Jobs
Number of JIT Jobs
Makespan (Maximum Completion Time)
Total Completion Time
Tiempo ocioso
Tiempo de espera
Tiempo medio en inventario (%)
OBJETIVO/INDICADOR

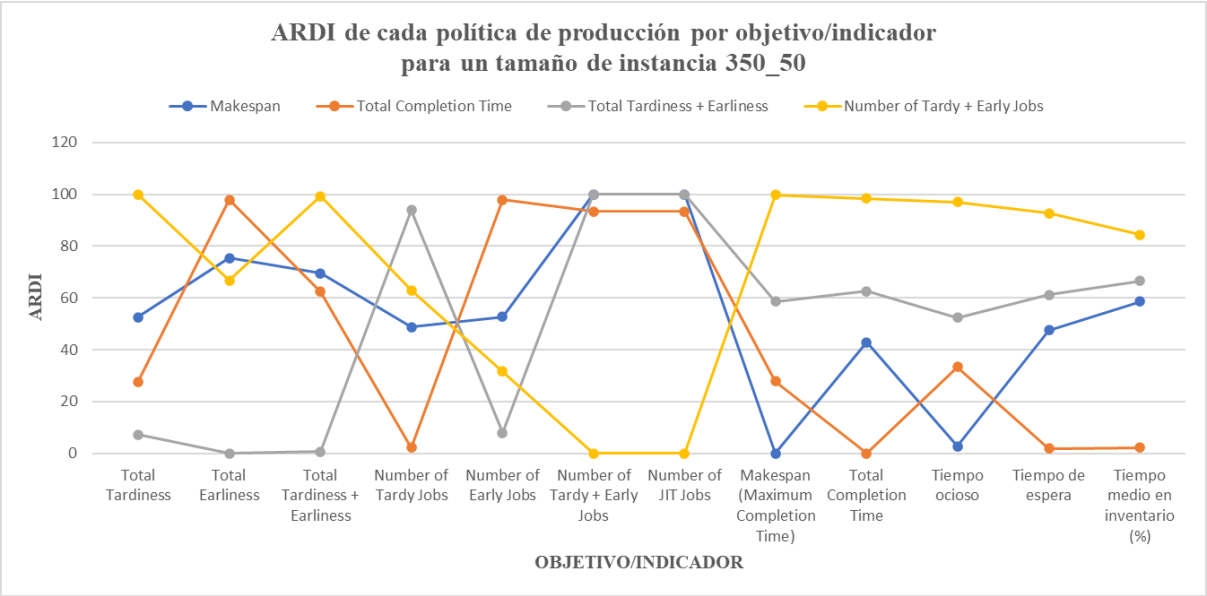

Número de instancias para las que cada política de producción proporciona el mejor resultado por objetivo/indicador para un tamaño de instancia 250_50
Makespan
Total Completion Time
Total Tardiness + Earliness
Number of Tardy + Early Jobs
NÚMERO DE INSTANCIAS
50
45
40
35
30
25
20
15
10
5
0
Total Tardiness
Total Earliness
Total Tardiness + Earliness
Number of Tardy Jobs
Number of Early Jobs
Number of Tardy + Early Jobs
Number of JIT Jobs
Makespan (Maximum Completion Time)
Total Completion Time
Tiempo ocioso
Tiempo de espera
Tiempo medio en inventario (%)
OBJETIVO/INDICADOR

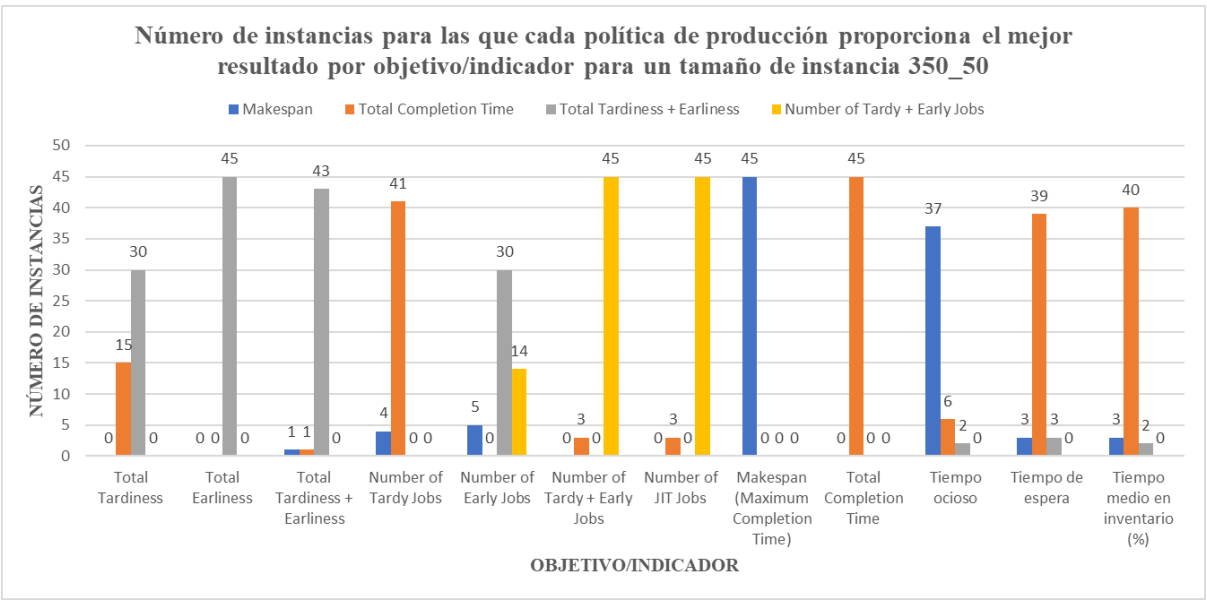

ARDI de cada política de producción por objetivo/indicador para un tamaño de instancia 350_10
Makespan
Total Completion Time
Total Tardiness + Earliness
Number of Tardy + Early Jobs
ARDI
120
100
80
60
40
20
0
Total Tardiness
Total Earliness
Total Tardiness + Earliness
Number of Tardy Jobs
Number of Early Jobs
Number of Tardy + Early Jobs
Number of JIT Jobs
Makespan (Maximum Completion Time)
Total Completion Time
Tiempo ocioso
Tiempo de espera
Tiempo medio en inventario (%)
OBJETIVO/INDICADOR


Número de instancias para las que cada política de producción proporciona el mejor resultado por objetivo/indicador para un tamaño de instancia 350_10
Makespan
Total Completion Time
Total Tardiness + Earliness
Number of Tardy + Early Jobs
NÚMERO DE INSTANCIAS
50
45
40
35
30
25
20
15
10
5
0
1
15
29
0
0 0
45
0
0 0
45
0
5
41
1
0
2
0
23
22
0 0
1
45
0 0
1
45
44
1
0 0
0
0 0
45
44
1
0 0
0
2
1
42
0
2
0
43
Total Tardiness
Total Earliness
Total Tardiness + Earliness
Number of Tardy Jobs
Number of Early Jobs
Number of Tardy + Early Jobs
Number of JIT Jobs
Makespan (Maximum Completion Time)
Total Completion Time
Tiempo ocioso
Tiempo de espera
Tiempo medio en inventario (%)
OBJETIVO/INDICADOR


ARDI de cada política de producción por objetivo/indicador para un tamaño de instancia 350_30
Makespan
Total Completion Time
Total Tardiness + Earliness
Number of Tardy + Early Jobs
ARDI
120
100
80
60
40
20
0
Total Tardiness
Total Earliness
Total Tardiness + Earliness
Number of Tardy Jobs
Number of Early Jobs
Number of Tardy + Early Jobs
Number of JIT Jobs
Makespan (Maximum Completion Time)
Total Completion Time
Tiempo ocioso
Tiempo de espera
Tiempo medio en inventario (%)
OBJETIVO/INDICADOR


Número de instancias para las que cada política de producción proporciona el mejor resultado por objetivo/indicador para un tamaño de instancia 350_30
Makespan
Total Completion Time
Total Tardiness + Earliness
Number of Tardy + Early Jobs
NÚMERO DE INSTANCIAS
50
45
40
35
30
25
20
15
10
5
0
Total Tardiness
Total Earliness
Total Tardiness + Earliness
Number of Tardy Jobs
Number of Early Jobs
Number of Tardy + Early Jobs
Number of JIT Jobs
Makespan (Maximum Completion Time)
Total Completion Time
Tiempo ocioso
Tiempo de espera
Tiempo medio en inventario (%)
OBJETIVO/INDICADOR


ARDI de cada política de producción por objetivo/indicador para un tamaño de instancia 350_50
Makespan
Total Completion Time
Total Tardiness + Earliness
Number of Tardy + Early Jobs
ARDI
120
100
80
60
40
20
0
Total Tardiness
Total Earliness
Total Tardiness + Earliness
Number of Tardy Jobs
Number of Early Jobs
Number of Tardy + Early Jobs
Number of JIT Jobs
Makespan (Maximum Completion Time)
Total Completion Time
Tiempo ocioso
Tiempo de espera
Tiempo medio en inventario (%)
OBJETIVO/INDICADOR


Número de instancias para las que cada política de producción proporciona el mejor resultado por objetivo/indicador para un tamaño de instancia 350_50
Makespan
Total Completion Time
Total Tardiness + Earliness
Number of Tardy + Early Jobs
NÚMERO DE INSTANCIAS
50
45
40
35
30
25
20
15
10
5
0
Total Tardiness
Total Earliness
Total Tardiness + Earliness
Number of Tardy Jobs
Number of Early Jobs
Number of Tardy + Early Jobs
Number of JIT Jobs
Makespan (Maximum Completion Time)
Total Completion Time
Tiempo ocioso
Tiempo de espera
Tiempo medio en inventario (%)
OBJETIVO/INDICADOR

ANEXO B: Código Python

```
from random import randint

#---------------------------------------- COMPLETION TIMES ----------------------------------------
# Función que calcula los completion times de cada trabajo (j) en cada máquina (i) --> en el orden de la secuencia
def cij(machines, jobs, pij, dj, sec):
    completion_times = [[0 for j in range(len(sec))] for i in range(machines)]
    completion_times[0][0] = pij[0][sec[0]]
    for i in range(1, machines):
        completion_times[i][0] = completion_times[i-1][0] + pij[i][sec[0]]
    for j in range(1, len(sec)):
        completion_times[0][j] = completion_times[0][j-1] + pij[0][sec[j]]
    for i in range(1, machines):
        for j in range(1, len(sec)):
            completion_times[i][j] = max(completion_times[i-1][j], completion_times[i][j-1]) + pij[i][sec[j]]
    return completion_times

#Función que calcula los completion times de cada trabajo (j) en cada máquina (i) --> en el orden de los trabajos
def cij_order(machines, jobs, pij, dj, sec):
    cij_f = cij(machines, jobs, pij, dj, sec)
    completion_times = [[0 for j in range(len(sec))] for i in range(machines)]
    for i in range(machines):
        for j in range(len(sec)):
            completion_times[i][j] = cij_f[i][sec.index(j)]
    return completion_times

#Función que calcula los completion times de cada trabajo (j) --> en el orden de la secuencia
def cj(machines, jobs, pij, dj, sec):
    cij_f = cij(machines, jobs, pij, dj, sec)
    completion_times = [0 for j in range(len(sec))]
    for j in range(len(sec)):
        completion_times[j] = cij_f[-1][j]
    return completion_times
```

```
#Función que calcula los completion times de cada trabajo (j) --> en el orden de los trabajos
def cj_order(machines, jobs, pij, dj, sec):
    cij_f = cij(machines, jobs, pij, dj, sec)
    completion_times = [0 for j in range(len(sec))]
    for j in range(len(sec)):
        completion_times[j] = cij_f[-1][sec.index(j)]
    return completion_times

#---------------------------------------- OBJETIVOS JIT ----------------------------------------------------
#Función que calcula el Total Tardiness
def Total_Tj(machines, jobs, pij, dj, sec):
    cj_order_f = cj_order(machines, jobs, pij, dj, sec)
    Tj = [0 for j in range(len(sec))]
    for j in range(len(sec)):
        Tj[j] = max (cj_order_f[j]-dj[j],0)
    Total_Tj = sum(Tj)
    return Total_Tj

#Función que calcula el Total Earliness
def Total_Ej(machines, jobs, pij, dj, sec):
    cj_order_f = cj_order(machines, jobs, pij, dj, sec)
    Ej = [0 for j in range(len(sec))]
    for j in range(len(sec)):
        Ej[j] = max (dj[j]-cj_order_f[j],0)
    Total_Ej = sum(Ej)
    return Total_Ej

#Función que calcula el Total Tardiness + Earliness
def Total_Tj_Ej(machines, jobs, pij, dj, sec):
    Total_Tj_f = Total_Tj(machines, jobs, pij, dj, sec)
    Total_Ej_f = Total_Ej(machines, jobs, pij, dj, sec)
    Total_Tj_Ej = Total_Tj_f + Total_Ej_f
    return Total_Tj_Ej
```

```
#Función que calcula el Number of Tardy Jobs
def Number_Tardy(machines, jobs, pij, dj, sec):
    cj_order_f = cj_order(machines, jobs, pij, dj, sec)
    Number_Tardy = 0
    for j in range(len(sec)):
        if cj_order_f[j] > dj[j]:
            Number_Tardy += 1
    return Number_Tardy

#Función que calcula el Number of Early Jobs
def Number_Early(machines, jobs, pij, dj, sec):
    cj_order_f = cj_order(machines, jobs, pij, dj, sec)
    Number_Early = 0
    for j in range(len(sec)):
        if dj[j] > cj_order_f[j]:
            Number_Early += 1
    return Number_Early

#Función que calcula el Number of Tardy + Early Jobs
def Number_Tardy_Early(machines, jobs, pij, dj, sec):
    Number_Tardy_f = Number_Tardy(machines, jobs, pij, dj, sec)
    Number_Early_f = Number_Early(machines, jobs, pij, dj, sec)
    Number_Tardy_Early = Number_Tardy_f + Number_Early_f
    return Number_Tardy_Early

#Función que calcula el Number of JIT Jobs
def Number_JIT(machines, jobs, pij, dj, sec):
    cj_order_f = cj_order(machines, jobs, pij, dj, sec)
    Number_JIT = 0
    for j in range(len(sec)):
        if dj[j] == cj_order_f[j]:
            Number_JIT += 1
    return Number_JIT
```

```
#---------------------------------- OBJETIVOS DE PRODUCCIÓN TRADICIONALES -------------------------
#Función que calcula el Makespan (Maximum Completion Time)
def Max_Cj(machines, jobs, pij, dj, sec):
    cj_order_f = cj_order(machines, jobs, pij, dj, sec)
    makespan = max(cj_order_f)
    return makespan

#Función que calcula el Total Completion Time
def Total_Cj(machines, jobs, pij, dj, sec):
    cj_order_f = cj_order(machines, jobs, pij, dj, sec)
    Total_Cj = sum(cj_order_f)
    return Total_Cj

#---------------------------------------- INDICADORES ----------------------------------------------
#Función que calcula el Tiempo ocioso --> tiempo que las máquinas están paradas entre trabajo y trabajo
def t_ocioso(machines, jobs, pij, dj, sec):
    cij_f = cij(machines, jobs, pij, dj, sec)
    t_ocioso = 0
    for i in range(1, machines):
        for j in range(1, len(sec)):
            t_ocioso += max(cij_f[i-1][j] - cij_f[i][j-1] , 0)
    return t_ocioso

#Función que calcula el Tiempo de espera --> tiempo que los trabajos están esperando a ser procesados entre máquina y máquina
def t_espera(machines, jobs, pij, dj, sec):
    cij_f = cij(machines, jobs, pij, dj, sec)
    t_espera = 0
    for i in range(1, machines):
        for j in range(1, len(sec)):
            t_espera += max(cij_f[i][j-1] - cij_f[i-1][j] , 0)
    return t_espera
```

```
#Función que calcula el Tiempo medio en inventario
def t_medio_inventario(machines, jobs, pij, dj, sec):
    cij_f = cij(machines, jobs, pij, dj, sec)
    cj_f = cj(machines, jobs, pij, dj, sec)
    t_espera_j = [0 for j in range(len(sec))]
    for j in range(1, len(sec)):
        for i in range(1, machines):
            t_espera_j[j] += max(cij_f[i][j-1] - cij_f[i-1][j] , 0)
    t_total_j = [0 for j in range(len(sec))]
    for j in range(len(sec)):
        t_total_j[j] = cj_f[j] - (cij_f[0][j] - pij[0][sec[j]])
    t_inventario_j = [0 for j in range(len(sec))]
    for j in range(len(sec)):
        t_inventario_j[j] = t_espera_j[j] / t_total_j[j]
    t_medio_inventario = sum(t_inventario_j) / len(t_inventario_j)
        return round(t_medio_inventario, 4)
```

```
from Funciones import *
import random
import time
from copy import deepcopy
import operator

#-------------------------------- FUNCIONES NECESARIAS PARA AG TANTO V1 COMO V2 ------------------
#Función objetivo (fitness) que se va a utilizar para evaluar los individuos --> objetivo/política de producción
def fitness(machines, jobs, pij, dj, individuo):
    return Max_Cj(machines, jobs, pij, dj, individuo)

#Generar un individuo (secuencia) aleatoriamente
def genera_individuo(longitud):
    individuo = []
    for i in range(longitud):
        x = randint(0, longitud-1)
        while(individuo.count(x)!= 0):
            x = randint(0, longitud-1)
        individuo.append(x)
    return individuo
```

```
#Generar población aleatoriamente
def genera_poblacion(tamano, longitud):
    return [genera_individuo(longitud) for _ in range(tamano)]

#Seleccionar individuos para la reproducción (padres) --> Torneo (n-tournament)
def tournament_selection(machines, jobs, pij, dj, poblacion, pressure):
    cjto_individuos = random.sample(poblacion, round(len(poblacion) * pressure))
    best_individuo = cjto_individuos[0]
    best_fitness = fitness(machines, jobs, pij, dj, cjto_individuos[0])
    for i in range(1, len(cjto_individuos)):
        fitness_i = fitness(machines, jobs, pij, dj, cjto_individuos[i])
        if fitness_i < best_fitness:
            best_individuo = cjto_individuos[i]
            best_fitness = fitness_i
    return best_individuo

#Cruzar dos individuos/padres (generación de dos hijos/offsprings) --> Order one crossover
def order_one_crossover(padre1, padre2):
    crossover_point = random.randint(1, len(padre1) - 1)
    hijo1 = padre1[:crossover_point]
    for j in range (len(padre2)):
        if padre2[j] not in hijo1:
            hijo1.append(padre2[j])
    hijo2 = padre2[:crossover_point]
    for j in range (len(padre1)):
        if padre1[j] not in hijo2:
            hijo2.append(padre1[j])
    return hijo1, hijo2

#Mutar un individuo --> Shift mutation
def shift_mutation(individuo, probabilidad_mutacion):
    for j in individuo:
        if random.random() < probabilidad_mutacion:
            nueva_pos = random.randint(0, len(individuo) - 1)
            individuo.insert(nueva_pos, individuo.pop(j))
    return individuo
```

```
#Función que elimina el peor individuo de una población e inserta un nuevo individuo dado --> SÓLO SI EL NUEVO INDIVIDUO ES MEJOR QUE EL PEOR INDIVIDUO DE LA POBLACIÓN Y SI ES ÚNICO
def insercion_individuo(machines, jobs, pij, dj, poblacion, nuevo_individuo):
    worst_individuo = poblacion[0]
    worst_fitness = fitness(machines, jobs, pij, dj, poblacion[0])
    for j in range (1, len(poblacion)):
        fitness_j = fitness(machines, jobs, pij, dj, poblacion[j])
        if fitness_j > worst_fitness:
            worst_individuo = poblacion[j]
            worst_fitness = fitness_j
    nuevo_fitness = fitness(machines, jobs, pij, dj, nuevo_individuo)
    if (nuevo_fitness < worst_fitness) and (nuevo_individuo not in poblacion):
        poblacion.remove(worst_individuo)
        poblacion.append(nuevo_individuo)


#---------------------------------------- ALGORITMO GENÉTICO V1 ----------------------------------------

def algoritmo_genetico_V1(machines, jobs, pij, dj, tamano_poblacion, pressure, probabilidad_mutacion):
    poblacion = genera_poblacion(tamano_poblacion, jobs)
    best_individuo = poblacion[0]
    best_fitness = fitness(machines, jobs, pij, dj, poblacion[0])
    for j in range (1, tamano_poblacion):
        fitness_j = fitness(machines, jobs, pij, dj, poblacion[j])
        if fitness_j < best_fitness:
            best_individuo = poblacion[j]
            best_fitness = fitness_j

    start = time.time()
    while time.time()-start < (30*jobs*machines / 2000):
        padre1 = tournament_selection(machines, jobs, pij, dj, poblacion, pressure)
        padre2 = tournament_selection(machines, jobs, pij, dj, poblacion, pressure)

        hijo1, hijo2 = order_one_crossover(padre1, padre2)
        hijo1_mutado = shift_mutation(hijo1, probabilidad_mutacion)
        hijo2_mutado = shift_mutation(hijo2, probabilidad_mutacion)
        fitness_hijo1_mutado = fitness(machines, jobs, pij, dj, hijo1_mutado)
        fitness_hijo2_mutado = fitness(machines, jobs, pij, dj, hijo2_mutado)
```

```
        insercion_individuo(machines, jobs, pij, dj, poblacion, hijo1_mutado)
        insercion_individuo(machines, jobs, pij, dj, poblacion, hijo2_mutado)
        if fitness_hijo1_mutado < best_fitness:
            best_individuo = hijo1_mutado
            best_fitness = fitness_hijo1_mutado
        if fitness_hijo2_mutado < best_fitness:
            best_individuo = hijo2_mutado
            best_fitness = fitness_hijo2_mutado

    return poblacion, best_individuo, best_fitness


#---------------------------------- FUNCIONES NECESARIAS PARA AG V2 ----------------------------------

#Regla de despacho utilizada al inicializar la población --> cuando política de producción tradicional
def regla_despacho_SPT(machines, jobs, pij, dj):
    sum_pij = [0 for j in range(jobs)]
    for j in range(jobs):
        sum_pij[j] = sum(pij[i][j] for i in range(machines))
    enum = enumerate(sum_pij)
    sorted_enum = sorted(enum, key=operator.itemgetter(1))
    secuencia = [index for index,item in sorted_enum]
    return secuencia

#Regla de despacho utilizada al inicializar la población --> cuando política de producción JIT
def regla_despacho_EDD(machines, jobs, pij, dj):
    enum = enumerate(dj)
    sorted_enum = sorted(enum, key=operator.itemgetter(1))
    secuencia = [index for index,item in sorted_enum]
    return secuencia

#Generar población aleatoriamente excepto un individuo obtenido mediante regla de despacho (SPT o EDD)
def genera_poblacion_SPT(tamano, longitud, machines, jobs, pij, dj):
    poblacion = [genera_individuo(longitud) for _ in range(tamano-1)]
    poblacion.append(regla_despacho_SPT(machines, jobs, pij, dj))
    return poblacion
```

```
def genera_poblacion_EDD(tamano, longitud, machines, jobs, pij, dj):
    poblacion = [genera_individuo(longitud) for _ in range(tamano-1)]
    poblacion.append(regla_despacho_EDD(machines, jobs, pij, dj))
    return poblacion


#Procedimiento de búsqueda local (Local Search) --> INSERTION neighbourhood
def local_search(machines, jobs, pij, dj, individuo, probabilidad_local_search, start):
    best_vecino = deepcopy(individuo)
    best_vecino_fitness = fitness(machines, jobs, pij, dj, individuo)
    if random.random() < probabilidad_local_search:
        for j in range (jobs):
            for k in range (jobs):
                if (j!=k) and (k!=j-1) and time.time()-start < (30*jobs*machines / 2000):
                    individuo_vecino = deepcopy(individuo)
                    individuo_vecino.insert(k, individuo_vecino.pop(j))
                    fitness_vecino = fitness(machines, jobs, pij, dj, individuo_vecino)
                    if (fitness_vecino < best_vecino_fitness):
                        best_vecino = deepcopy(individuo_vecino)
                        best_vecino_fitness = deepcopy(fitness_vecino)
    return best_vecino, best_vecino_fitness


#Diversidad de una población --> para el REESTART MECHANISM --> se busca asegurar que siempre exista cierta diversidad en la población
def diversidad(poblacion):
    GeneCount = [[0 for alfa in range(len(poblacion[0]))] for k in range(len(poblacion[0]))]
    for k in range(len(poblacion[0])):
        for alfa in range(len(poblacion[0])):
            for i in range(len(poblacion)):
                if poblacion[i][k] == alfa:
                    GeneCount[k][alfa] += 1
    GeneFrecuency = [[0 for alfa in range(len(poblacion[0]))] for k in range(len(poblacion[0]))]
    for k in range(len(poblacion[0])):
        for alfa in range(len(poblacion[0])):
            GeneFrecuency[k][alfa] = GeneCount[k][alfa] / len(poblacion)
    Div = (1 / (len(poblacion[0]) - 1)) * (sum(GeneFrecuency[k][alfa] * (1 - GeneFrecuency[k][alfa]) for alfa in range(len(poblacion[0])) for k in range(len(poblacion[0]))))
    return Div
```

```
#------------------------------------ ALGORITMO GENÉTICO V2 --------------------------------

def algoritmo_genetico_V2(machines, jobs, pij, dj, tamano_poblacion, pressure, probabilidad_mutacion, probabilidad_local_search, umbral_diversidad):
    poblacion = genera_poblacion_SPT(tamano_poblacion, jobs, machines, jobs, pij, dj)
    best_individuo = poblacion[0]
    best_fitness = fitness(machines, jobs, pij, dj, poblacion[0])
    for j in range (1, tamano_poblacion):
        fitness_j = fitness(machines, jobs, pij, dj, poblacion[j])
        if fitness_j < best_fitness:
            best_individuo = poblacion[j]
            best_fitness = fitness_j


    start = time.time()
    while time.time()-start < (30*jobs*machines / 2000):
        if diversidad(poblacion) < umbral_diversidad:
            poblacion = genera_poblacion_SPT(tamano_poblacion, jobs, machines, jobs, pij, dj)
        padre1 = tournament_selection(machines, jobs, pij, dj, poblacion, pressure)
        padre2 = tournament_selection(machines, jobs, pij, dj, poblacion, pressure)
        hijo1, hijo2 = order_one_crossover(padre1, padre2)
        hijo1_mutado = shift_mutation(hijo1, probabilidad_mutacion)
        hijo2_mutado = shift_mutation(hijo2, probabilidad_mutacion)
        mejor_vecino_hijo1_mutado, fitness_mejor_vecino_hijo1_mutado = local_search(machines, jobs, pij, dj, hijo1_mutado, probabilidad_local_search, start)
        mejor_vecino_hijo2_mutado, fitness_mejor_vecino_hijo2_mutado = local_search(machines, jobs, pij, dj, hijo2_mutado, probabilidad_local_search, start)
        insercion_individuo(machines, jobs, pij, dj, poblacion, mejor_vecino_hijo1_mutado)
        insercion_individuo(machines, jobs, pij, dj, poblacion, mejor_vecino_hijo2_mutado)
        if fitness_mejor_vecino_hijo1_mutado < best_fitness:
            best_individuo = mejor_vecino_hijo1_mutado
            best_fitness = fitness_mejor_vecino_hijo1_mutado
        if fitness_mejor_vecino_hijo2_mutado < best_fitness:
            best_individuo = mejor_vecino_hijo2_mutado
            best_fitness = fitness_mejor_vecino_hijo2_mutado


    return poblacion, best_individuo, best_fitness
```

```
from Funciones import *

#---------------------------------------- INSTANCIA y SECUENCIA DE EJEMPLO ---------------------------

def instancia_ejemplo():
    machines =  3
    jobs = 6
    pij = [[2,4,3,15,1,5],[7,3,5,1,3,6],[4,7,2,1,6,9]]
    dj = [15,35,12,32,45,30]
    return machines, jobs, pij, dj

def secuencia_ejemplo():
    secuencia = [2,0,5,3,1,4]
    return secuencia

#---------------------------------------------------- PRUEBA DE FUNCIONES -----------------------------------

machines, jobs, pij, dj = instancia_ejemplo()
print("Instancia de prueba:")
print("Número de máquinas:", machines)
print("Número de trabajos:", jobs)
print("Tiempos de proceso:", pij)
print("Fechas de entrega:", dj)

sec = secuencia_ejemplo()
print("\nSecuencia de prueba:", sec)

#Cálculo de los completion times de cada trabajo en cada máquina --> en el orden de la secuencia
cij = cij(machines, jobs, pij, dj, sec)
print("\nCompletion times de cada trabajo en cada máquina en el orden de la secuencia:", cij)

#Cálculo de los completion times de cada trabajo en cada máquina --> en el orden de los trabajos
cij_order = cij_order(machines, jobs, pij, dj, sec)
print("\nCompletion times de cada trabajo en cada máquina en el orden de los trabajos:", cij_order)

#Cálculo de los completion times de cada trabajo --> en el orden de la secuencia
cj = cj(machines, jobs, pij, dj, sec)
print("\nCompletion times de cada trabajo en el orden de la secuencia:", cj)
```

```
#Cálculo de los completion times de cada trabajo --> en el orden de los trabajos
cj_order = cj_order(machines, jobs, pij, dj, sec)
print("\nCompletion times de cada trabajo en el orden de los trabajos:", cj_order)

#Cálculo del Total Tardiness
Total_Tj = Total_Tj(machines, jobs, pij, dj, sec)
print("\nTotal Tardiness:", Total_Tj)

#Cálculo del Total Earliness
Total_Ej = Total_Ej(machines, jobs, pij, dj, sec)
print("\nTotal Earliness:", Total_Ej)

#Cálculo del Total Tardiness + Earliness
Total_Tj_Ej = Total_Tj_Ej(machines, jobs, pij, dj, sec)
print("\nTotal Tardiness + Earliness:", Total_Tj_Ej)

#Cálculo del Number of Tardy Jobs
Number_Tardy = Number_Tardy(machines, jobs, pij, dj, sec)
print("\nNumber of Tardy Jobs:", Number_Tardy)

#Cálculo del Number of Early Jobs
Number_Early = Number_Early(machines, jobs, pij, dj, sec)
print("\nNumber of Early Jobs:", Number_Early)

#Cálculo del Number of Tardy + Early Jobs
Number_Tardy_Early = Number_Tardy_Early(machines, jobs, pij, dj, sec)
print("\nNumber of Tardy + Early Jobs:", Number_Tardy_Early)

#Cálculo del Number of JIT Jobs
Number_JIT = Number_JIT(machines, jobs, pij, dj, sec)
print("\nNumber of JIT Jobs:", Number_JIT)

#Cálculo del Makespan (Maximum Completion Time)
Max_Cj = Max_Cj(machines, jobs, pij, dj, sec)
print("\nMakespan (Maximum Completion Time):", Max_Cj)
```

```
#Cálculo del Total Completion Time
Total_Cj = Total_Cj(machines, jobs, pij, dj, sec)
print("\nTotal Completion Time:", Total_Cj)

#Cálculo del Tiempo ocioso
t_ocioso = t_ocioso(machines, jobs, pij, dj, sec)
print("\nTiempo ocioso:", t_ocioso)

#Cálculo del Tiempo de espera
t_espera = t_espera(machines, jobs, pij, dj, sec)
print("\nTiempo de espera:", t_espera)

#Cálculo del Tiempo medio en inventario
t_medio_inventario = t_medio_inventario(machines, jobs, pij, dj, sec)
print("\nTiempo medio en inventario:", t_medio_inventario)
```

```
from Metaheuristica import *
import time

#-------------------- PRUEBA DE FUNCIONES NECESARIAS TANTO PARA V1 COMO V2 ----------------

print("\n---------- FUNCIONES NECESARIAS PARA AG TANTO V1 COMO V2----------")

#Parámetros función fitness
machines_ejemplo =  3
jobs_ejemplo = 6
pij_ejemplo = [[2,4,3,15,1,5],[7,3,5,1,3,6],[4,7,2,1,6,9]]
dj_ejemplo = [15,35,12,32,45,30]
individuo_ejemplo = [2,0,5,3,1,4]

#Cálculo función objetivo (fitness)
fitness_ejemplo = fitness(machines_ejemplo, jobs_ejemplo, pij_ejemplo, dj_ejemplo, individuo_ejemplo)
print("\nFitness:", fitness_ejemplo)

#Parámetros funciones genera_individuo y genera_población
longitud_ejemplo = 6
tamano_ejemplo = 10
```

```
#Generación de un individuo de manera aleatoria
individuo = genera_individuo(longitud_ejemplo)
print("\nIndividuo:", individuo)

#Generación de una población de manera aleatoria
poblacion = genera_poblacion(tamano_ejemplo, longitud_ejemplo)
print("\nPoblación:", poblacion)

#Fitness de cada individuo de la población:
fitness_poblacion = []
for j in range (len(poblacion)):
    fitness_poblacion.append(fitness(machines_ejemplo, jobs_ejemplo, pij_ejemplo, dj_ejemplo, poblacion[j]))
print("\nFitness población:", fitness_poblacion)

#Parámetros función de selección --> TORNEO
machines_ejemplo =  3
jobs_ejemplo = 6
pij_ejemplo = [[2,4,3,15,1,5],[7,3,5,1,3,6],[4,7,2,1,6,9]]
dj_ejemplo = [15,35,12,32,45,30]
poblacion_ejemplo = poblacion[:]
pressure_ejemplo = 0.4 #40%

#Selección de un individuo/padre --> TORNEO
padre = tournament_selection(machines_ejemplo, jobs_ejemplo, pij_ejemplo, dj_ejemplo, poblacion_ejemplo,
pressure_ejemplo)
print("\nIndividuo/padre seleccionado por torneo:", padre)

#Parámetros función de cruce --> ORDER ONE CROSSOVER
padre1_ejemplo = [3,17,9,15,8,14,11,7,13,19,6,5,1,18,4,2,16,10,20,12]
padre2_ejemplo = [3,9,14,15,11,19,6,18,5,8,7,17,1,16,4,2,10,13,20,12]

#Cruce de dos individuos/padres --> generación de dos nuevos individuos/hijos --> ORDER ONE
CROSSOVER
hijo1, hijo2 = order_one_crossover(padre1_ejemplo, padre2_ejemplo)
print("\nIndividuo/hijo 1 generado mediante order one crossover:", hijo1)
print("Individuo/hijo 2 generado mediante order one crossover:", hijo2)
```

```
#Parámetros función de mutación --> SHIFT MUTATION
individuo_ejemplo = [2,0,5,3,1,4]
prob_mutacion_ejemplo = 0.2 #20%

#Mutación de un individuo --> SHIFT MUTATION
individuo_mutado = shift_mutation(individuo_ejemplo, prob_mutacion_ejemplo)
print("\nIndividuo mutado mediante shift mutation:", individuo_mutado)

#Parámetros función inserción_individuo
nuevo_individuo_ejemplo = [3,4,5,0,1,2] #Fitness de 115

#Función inserción_individuo
insercion_individuo(machines_ejemplo, jobs_ejemplo, pij_ejemplo, dj_ejemplo, poblacion_ejemplo, nuevo_individuo_ejemplo)
print("\nPoblación nueva:", poblacion_ejemplo)



#---------------------------------- IMPLEMENTACIÓN ALGORITMO GENÉTICO V1 ----------------------

#Datos instancia
machines =  3
jobs = 6
pij = [[2,4,3,15,1,5],[7,3,5,1,3,6],[4,7,2,1,6,9]]
dj = [15,35,12,32,45,30]

#Parámetros del algoritmo genético
tamano_poblacion = 10
pressure = 0.5
probabilidad_mutacion = 0.3

#Ejecución del algoritmo genético
poblacion_algoritmo_V1, best_individuo_algoritmo_V1, best_fitness_algoritmo_V1 = algoritmo_genetico_V1(machines, jobs, pij, dj, tamano_poblacion, pressure, probabilidad_mutacion)

#Resultados obtenidos
print("\n---------- RESULTADOS ALGORITMO GENÉTICO V1 ----------")
print("\nPoblación final obtenida con algoritmo V1:", poblacion_algoritmo_V1)
print("\nMejor individuo obtenido con algoritmo V1:", best_individuo_algoritmo_V1)
print("\nFitness del mejor individuo obtenido con algoritmo V1:", best_fitness_algoritmo_V1)
```

```
#---------------------------------------- PRUEBA DE FUNCIONES NECESARIAS PARA V2 ------------------

print("\n---------- FUNCIONES NECESARIAS PARA AG V2 ----------")

#Datos de ejemplo
machines_ejemplo =  3
jobs_ejemplo = 6
pij_ejemplo = [[2,4,3,15,1,5],[7,3,5,1,3,6],[4,7,2,1,6,9]]
dj_ejemplo = [15,35,12,32,45,30]
longitud_ejemplo = 6
tamano_ejemplo = 10
individuo_ejemplo_localsearch = [4,2,1,0,5,3]
probabilidad_ejemplo_localsearch = 0.6 #60%
start_ejemplo = time.time()

#Funciones de mejora
sec_SPT = regla_despacho_SPT(machines_ejemplo, jobs_ejemplo, pij_ejemplo, dj_ejemplo)
print("\nIndividuo generado con regla de despacho SPT:", sec_SPT)

sec_EDD = regla_despacho_EDD(machines_ejemplo, jobs_ejemplo, pij_ejemplo, dj_ejemplo)
print("\nIndividuo generado con regla de despacho EDD:", sec_EDD)

poblacion_SPT = genera_poblacion_SPT(tamano_ejemplo, longitud_ejemplo, machines_ejemplo, jobs_ejemplo, pij_ejemplo, dj_ejemplo)
print("\nPoblación generada aleatoriamente + regla de despacho SPT:", poblacion_SPT)

poblacion_EDD = genera_poblacion_EDD(tamano_ejemplo, longitud_ejemplo, machines_ejemplo, jobs_ejemplo, pij_ejemplo, dj_ejemplo)
print("\nPoblación generada aleatoriamente + regla de despacho EDD:", poblacion_EDD)

best_vecino, best_vecino_fitness = local_search(machines_ejemplo, jobs_ejemplo, pij_ejemplo, dj_ejemplo, individuo_ejemplo_localsearch, probabilidad_ejemplo_localsearch, start_ejemplo)
print("\nMejor vecino encontrado:", best_vecino)
print("Fitness del mejor vecino:", best_vecino_fitness)

diversidad = diversidad([[0,1,2,3],[1,2,3,0],[0,3,1,2]])
print("\nValor de la diversidad (entre 0 y 1):", diversidad)
```

```
#---------------------------------- IMPLEMENTACIÓN ALGORITMO GENÉTICO V2 ----------------------

#Datos instancia
machines =  3
jobs = 6
pij = [[2,4,3,15,1,5],[7,3,5,1,3,6],[4,7,2,1,6,9]]
dj = [15,35,12,32,45,30]

#Parámetros del algoritmo genético mejorado
tamano_poblacion = 10
pressure = 0.5
probabilidad_mutacion = 0.3
probabilidad_local_search = 0.2
umbral_diversidad = 0.4

#Ejecución del algoritmo genético mejorado
poblacion_algoritmo_V2, best_individuo_algoritmo_V2, best_fitness_algoritmo_V2 = algoritmo_genetico_V2(machines, jobs, pij, dj, tamano_poblacion, pressure, probabilidad_mutacion, probabilidad_local_search, umbral_diversidad)

#Resultados obtenidos
print("\n---------- RESULTADOS ALGORITMO GENÉTICO V2 ----------")
print("\nPoblación final obtenida con algoritmo V2:", poblacion_algoritmo_V2)
print("\nMejor individuo obtenido con algoritmo V2:", best_individuo_algoritmo_V2)
print("\nFitness del mejor individuo obtenido con algoritmo V2:", best_fitness_algoritmo_V2)
```

```
from Metaheuristica import *
from Funciones import *
from os import path, remove
from copy import copy

#---------------------------------------------- LEER INSTANCIA ----------------------------------------

def lee_instancia(a, b, c, d, e):
    datos = []
    with open(f'Instancias/I_{a}_{b}_{c}_{d}_{e}.txt') as instancia:
```

```
for lineas in instancia:
    datos.append(lineas.split())
    datos_correctos = copy(datos)
    for i in range (len(datos)):
        for j in range (len(datos[i])):
            try:
                datos_correctos[i][j] = int(datos[i][j])
            except ValueError:
                try:
                    datos_correctos[i][j] = float(datos[i][j])
                except ValueError:
                    datos_correctos[i][j] = datos[i][j]
    jobs = datos_correctos[0][0]
    machines = datos_correctos[0][1]
    pij = [[0 for j in range(jobs)] for i in range(machines)]
    for i in range(machines):
        for j in range (jobs):
            pij[i][j] = datos_correctos[j+1][i*2+1]
    dj = []
    for j in range(jobs+2, 2*jobs+2):
        dj.append(datos_correctos[j][1])
    return machines, jobs, pij, dj

#---------------------------------------- EVALUAR INSTANCIA ----------------------------------------

def evalua_instancia(machines, jobs, pij, dj, a, b, c, d, e):
    tamano_poblacion = 30
    pressure = 0.3 #30%
    probabilidad_mutacion = 0.02
    probabilidad_local_search = 0.15
    umbral_diversidad = 0.4

    poblacion, best_individuo, best_fitness = algoritmo_genetico_V2(machines, jobs, pij, dj, tamano_poblacion,
pressure, probabilidad_mutacion, probabilidad_local_search, umbral_diversidad)

    Total_Tj_f = Total_Tj(machines, jobs, pij, dj, best_individuo) #Total Tardiness
    Total_Ej_f = Total_Ej(machines, jobs, pij, dj, best_individuo) #Total Earliness
    Total_Tj_Ej_f = Total_Tj_Ej(machines, jobs, pij, dj, best_individuo) #Total Tardiness + Earliness
```

```
    Number_Tardy_f = Number_Tardy(machines, jobs, pij, dj, best_individuo) #Number of Tardy Jobs
    Number_Early_f = Number_Early(machines, jobs, pij, dj, best_individuo) #Number of Early Jobs
    Number_Tardy_Early_f = Number_Tardy_Early(machines, jobs, pij, dj, best_individuo) #Number of Tardy + Early Jobs
    Number_JIT_f = Number_JIT(machines, jobs, pij, dj, best_individuo) #Number of JIT Jobs
    Max_Cj_f = Max_Cj(machines, jobs, pij, dj, best_individuo) #Makespan (Maximum Completion Time)
    Total_Cj_f = Total_Cj(machines, jobs, pij, dj, best_individuo) #Total Completion Time

    t_ocioso_f = t_ocioso(machines, jobs, pij, dj, best_individuo) #Tiempo ocioso
    t_espera_f = t_espera(machines, jobs, pij, dj, best_individuo) #Tiempo de espera
    t_medio_inventario_f = t_medio_inventario(machines, jobs, pij, dj, best_individuo) #Tiempo medio en inventario

    archivo = f"Resultados/Algoritmo Genetico V2/Makespan/R_{a}_{b}_{c}_{d}_{e}.txt"
    if path.exists(str(archivo)):
        remove(str(archivo))

    with open(f'Resultados/Algoritmo Genetico V2/Makespan/R_{a}_{b}_{c}_{d}_{e}.txt','a') as archivo_resultados:
        archivo_resultados.write('Solución obtenida: ' + str(best_individuo) + '\n' +
                                 '\nOBJETIVOS JIT' +
                                 '\nTotal Tardiness: ' + str(Total_Tj_f) +
                                 '\nTotal Earliness: ' + str(Total_Ej_f) +
                                 '\nTotal Tardiness + Earliness: ' + str(Total_Tj_Ej_f) +
                                 '\nNumber of Tardy Jobs: ' + str(Number_Tardy_f) +
                                 '\nNumber of Early Jobs: ' + str(Number_Early_f) +
                                 '\nNumber of Tardy + Early Jobs: ' + str(Number_Tardy_Early_f) +
                                 '\nNumber of JIT Jobs: ' + str(Number_JIT_f) + '\n' +
                                 '\nOBJETIVOS DE PRODUCCIÓN TRADICIONALES' +
                                 '\nMakespan (Maximum Completion Time): ' + str(Max_Cj_f) +
                                 '\nTotal Completion Time: ' + str(Total_Cj_f) + '\n' +
                                 '\nINDICADORES' +
                                 '\nTiempo ocioso: ' + str(t_ocioso_f) +
                                 '\nTiempo de espera: ' + str(t_espera_f) +
                                 '\nTiempo medio en inventario: ' + str(round(t_medio_inventario_f * 100, 2)) + ' %')
```

```
#------------------------- PRUEBA FUNCIONES LEER Y EVALUAR INSTANCIA----------------------

#Para la prueba utilizamos la instancia "I_0,2_0,2_50_10_1.txt"

#Leemos la instancia
machines_1, jobs_1, pij_1, dj_1 = lee_instancia('0,2', '0,2', 50, 10, 1)

print("\nMachines:", machines_1)
print("\nJobs:", jobs_1)
print("\npij:", pij_1)
print("\ndj:", dj_1)

#Evaluamos la instancia
evalua_instancia(machines_1, jobs_1, pij_1, dj_1, '0,2', '0,2', 50, 10, '1_prueba'
```

```
from Leer_Evaluar_Instancia import *

#------------------ LEER Y EVALUAR TODAS LAS INSTANCIAS (540 instancias en total) ---------

for a in ('0,2', '0,4', '0,6'):
    for b in ('0,2', '0,6', '1'):
        for c in (50, 150, 250, 350):
            for d in (10, 30, 50):
                for e in (1, 2, 3, 4, 5):
                    #Leemos la instancia
                    machines, jobs, pij, dj = lee_instancia(a, b, c, d, e)
                    #Evaluamos la instancia
                    evalua_instancia(machines, jobs, pij, dj, a, b, c, d, e)
```

```
from os import path, remove

#------------------------- RESULTADOS ACUMULADOS (540 instancias en total) --------------------

i = 1

#Generación resultados ACUMULADOS
archivo = f"Resultados/Algoritmo Genetico V2/Makespan acumulado V2.txt"
if path.exists(str(archivo)):
   remove(str(archivo))

with open(f'Resultados/Algoritmo Genetico V2/Makespan acumulado V2.txt','a') as archivo_resultados:

   for a in ('0,2', '0,4', '0,6'):
      for b in ('0,2', '0,6', '1'):
         for c in (50, 150, 250, 350):
            for d in (10, 30, 50):
               for e in (1, 2, 3, 4, 5):
                  archivo_resultados.write('Instancia: ' + str(i) + '\n')
                  with open(f'Resultados/Algoritmo Genetico V2/Makespan/R_{a}_{b}_{c}_{d}_{e}.txt','r') as
instancia:
                     for line in instancia:
                        archivo_resultados.write(line)
                     archivo_resultados.write('\n\n')
                  i += 1
```