UNIVERSITAT POLITÉCNICA DE VALENCIA

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Departamento de Estadística e Investigación Operativa Aplicadas y Calidad

Máster Universitario en Ingeniería de Análisis de Datos, Mejora de Procesos y Toma de Decisiones

Trabajo Final de Máster

Programación de la Producción en Máquinas Paralelas sujeto a Adelantos, Retrasos y Fechas Límite

Autor: Jeffry Miguel Marte Collado

Tutor: Rubén Ruiz García

Curso: 2016-2017

Resumen

Este Trabajo Final de Máster trata sobre la programación de la producción en máquinas en paralelo no relacionadas, tomando en cuenta fechas de lanzamiento, fechas límite, tiempos de preparación dependientes de la secuencia y de la máquina con el objetivo de minimizar los adelantos y retrasos ponderados. Se propone un método heurístico/metaheurístico para resolverlo. Se plantea un modelo matemático de programación lineal entera mixta y la implementación del modelo en Lingo versión 17. El método propuesto es un algoritmo genético cuya población inicial está compuesta por soluciones obtenidas de la regla de despacho SPT y soluciones aleatorias. El algoritmo se ha implementado en C#. Se han llevado a cabo las pruebas computacionales necesarias para evaluar los métodos propuestos. Se han realizado evaluaciones estadísticas para calibrar el algoritmo propuesto. Los análisis estadísticos realizados indican que la versión del algoritmo que incorpora operadores que tienen la posibilidad de incluir tiempos ociosos es la que mejor funciona.

Resum

Este Treball Final de Màster tracta sobre la programació de la producció en màquines en paral·lel no relacionades, tenint en compte dates de llançament, dates límit, temps de preparació dependents de la seqüència i de la màquina amb l'objectiu de minimitzar els avanços i retards ponderats. Es proposa un mètode heurístico/metaheurístico per a resoldre-ho. Es planteja un model matemàtic de programació lineal sencera mixta i la implementació del model en Lingo versió 17. El mètode proposat és un algoritme genètic la població inicial del qual està composta per solucions obtingudes de la regla de despatx SPT i solucions aleatòries. L'algoritme s'ha implementat en C#. S'han dut a terme les proves computacionals necessàries per a avaluar els mètodes proposats. S'han realitzat avaluacions estadístiques per a calibrar l'algoritme proposat. Les anàlisis estadístiques realitzats indiquen que la versió de l'algoritme que incorpora operadors que tenen la possibilitat d'incloure temps ociosos és la que millor funciona.

Abstract

This Final Master's Work deals with the scheduling of production on unrelated parallel machines, taking into account launch dates, deadlines, sequence and machine-dependent preparation times in order to minimize weighted earliness and weighted tardiness. A heuristic / metaheuristic method is proposed to solve it.
We propose a mathematical model of mixed integer linear programming and the implementation of the model in Lingo version 17. The proposed method is a genetic algorithm whose initial population consists of solutions obtained from the dispatch rule SPT and random solutions. The algorithm has been implemented in C #. The necessary computational tests have been carried out to evaluate the proposed methods. Statistical evaluations have been performed to calibrate the proposed algorithm. Statistical analyzes indicate that the version of the algorithm that incorporates operators that have the possibility of including idle times is the one that works best.

Índice General

Índice de Figuras

Índice de Tablas

Capítulo 1. Introducción, Motivación y Objetivos

1.1 Introducción

La programación de la producción es un proceso de toma de decisiones cuyo objetivo es optimizar uno o más objetivos sujetos a restricciones o limitaciones. Este proceso desempeña un papel de gran importancia en la fabricación, así como en los sistemas de servicios e industrias, debido a que las empresas deben mantenerse competitivas siendo capaces de cumplir con los compromisos acordados con los clientes y los plazos de producción. En términos generales la programación de la producción se refiere a la asignación de recursos, usualmente limitados, a unas tareas en un período de tiempo. Los recursos y las tareas pueden ser de diferentes tipos. Por ejemplo, asignación de trabajadores en los turnos de la empresa, programación de los aviones en los aeropuertos y unidades de procesamiento a través de una computadora. Previo a realizar las fases de programación de la producción, se debe haber determinado cuales productos se han de realizar y en qué cantidades, la cantidad de recursos disponibles, configuraciones y como están dispuestos, lo que se denomina como planificación de la producción.

La programación de la producción en talleres con máquinas en paralelo comúnmente presenta una estructura especial, ya que estos pueden tener elementos de talleres de flujo, así como de talleres de flujos híbridos. Estos pueden modelizar las etapas de cuello de botella de un proceso en donde se replican varias máquinas para aumentar la capacidad. En el problema de programación de máquinas en paralelo hay un conjunto de $N = \{1,\ldots,n\}$ trabajos que deben ser procesados en una máquina de un conjunto $M = \{1,\ldots,m\}$ de m máquinas en paralelo. Cada trabajo debe visitar una máquina y se asume que todas las máquinas son capaces de procesar todos los trabajos. Cada máquina no puede procesar más de un trabajo al mismo tiempo y una vez comenzados los trabajos deben ser procesados hasta completarse, no pueden ser interrumpidos. El caso más general de máquinas en paralelo es máquinas no relacionadas que incluyen máquinas idénticas y uniformes, donde los tiempos

de procesamiento de cada trabajo dependen de la máquina donde sea asignado el trabajo. Por lo que se tendrá un matriz de tiempos de procesamiento p_{ij} ,siendo los tiempos de procesamiento p_{ij} no negativos, fijos, conocidos con antelación y se asumen que son enteros.

El criterio de optimización más estudiado es la minimización del máximo tiempo de finalización denominado (Cmáx), pero a pesar de esto ha sido criticado debido a que la satisfacción del cliente no es la prioridad de este objetivo. Usualmente los trabajos ayudan a modelar las órdenes que los clientes realizan y que deben ser entregadas en fechas específicas. Estas fechas de entrega al igual que los tiempos de procesamiento son no negativas, fijas, deterministas, conocidas con antelación y se conocen como d_j. Por lo tanto, terminar antes $C_j < d_j$ o después $C_j > d_j$ de la fecha de entrega tiene consecuencias negativas.

La filosofía JIT busca identificar y eliminar los desperdicios como la: sobreproducción, tiempos de espera, transporte, inventario, movimiento y productos defectuoso. Debido a la gran aceptación de esta filosofía a lo largo de los años es que el objetivo de minimizar el Adelanto/Retraso $\sum_{j=1}^{n}(E_j + T_j)$, ha ido tomando gran importancia y convirtiéndose en objeto de estudio, donde el adelanto de un trabajo j se calcula $E_j = \text{máx}\{d_j - C_j, 0\}$. Y el retraso de un trabajo j es definida como $T_j = máx\{C_j - d_j, 0\}$ se desea completar los trabajos lo más cerca posible de sus fechas de entrega, ya que si se terminan los trabajos por adelantado se incurre en costos de inventario si el cliente no acepta el pedido hasta la fecha de entrega y si se termina con retraso se incurre en compensaciones hacia el cliente por haber incumplido la fecha de entrega. Se debe tener en cuenta que un trabajo solo puede ir retrasado, adelantado o a tiempo. En este trabajo de fin de Máster se han considerado pesos para los adelantos/retrasos ya que no todos los trabajos tienen la misma importancia. Cada unidad de tiempo en el cual un trabajo está retrasado/adelantado se multiplica por una prioridad que es diferente para cada trabajo. Como resultado el objetivo a considerar es: $\sum_{j=1}^{n}(W_j'E_j + W_jT_j)$ donde W_j' es el peso para los adelantos y W_j es el peso para los retrasos.

Para hacer el problema más realista se han agregado fechas de lanzamiento que son los instantes de tiempo donde un trabajo puede comenzar o estar disponible, debido a que puede estar esperando materia prima o de acuerdo al tiempo en que se ha recibido la orden siendo estos solo dos ejemplos, se denota como r_j. Las fechas límite no pueden ser violadas al contrario de las fechas de

entrega, que se pueden no cumplir, so pena de incurrir en las penalizaciones arriba comentadas. Por lo tanto, todos los trabajos deben ser terminados antes o en la fecha límite. Cuando se cuenta con fechas de lanzamiento y fechas límite el resultado es llamado "ventana de procesamiento" ya que el tiempo de procesamiento de un trabajo debe ser completado en este intervalo de tiempo $\bar{d}_j - r_j$.

Las fechas límite en general y la ventana de procesamiento afectan a la obtención de soluciones factibles al problema. Ya que, no todas las secuencias de ordenamiento de los trabajos en las máquinas son posibles, cuando se está trabajando con adelantos/retrasos conviene incluir una fecha límite para que no retrase los trabajos más allá de esa fecha cuando este minimizando los retrasos. Los tiempos de preparación son tiempos no productivos que se necesitan en las máquinas para realizar configuraciones o preparaciones y limpiezas para la producción entre los trabajos. Estos tiempos han sido estudiados y revisados por Allahverdi, et al [1], [2], [3] así como en Allahverdi y Soroush [4], donde se resalta su importancia. Para este trabajo se utilizan tiempos de preparación dependientes de la secuencia de los trabajos y de las máquinas, donde S_{ijk} es el tiempo de preparación necesario en la maquina i cuando se ha procesado el trabajo j y le sigue un trabajo k en la secuencia. Estos tiempos de preparación son potencialmente asimétricos, es decir los tiempos de preparación entre los trabajos j y k en una máquina i pueden ser diferentes de tiempos de preparación entre k y j en la misma máquina.

Dado que se desea minimizar los pesos ponderados de adelantos/retrasos, con tiempos preparación dependientes de la secuencia y la máquina, el orden de asignación de los trabajos se convierte de vital importancia en la programación de máquinas paralelas no relacionadas, ya que dependiendo del orden que se asigna el trabajo, el tiempo de finalización cambiará por lo tanto cambiando el adelanto/retraso. El problema a tratar será:

$$\mathrm{R}/r_j, \bar{d}_j, S_{ijk}/\sum_{j=1}^{n}(W_j'E_j + W_jT_j)$$

Donde se insertarán tiempos ociosos para ayudar a la función objetivo principalmente a reducir los adelantos. Se resolverá un modelo MILP para este problema en un solver de optimización y se propondrá un algoritmo genético(GA) para resolver el problema de forma heurística, probando las variaciones necesarias del algoritmo que permitan hacer las comparaciones si se

introduce o no tiempos ociosos en las máquinas y los beneficios que estos pueden brindar principalmente a los adelantos.

1.2 Motivación

Al iniciar este trabajo final de máster se eligió un problema de programación de la producción de máquinas en paralelo no relacionadas, siendo este el más general de los problemas de máquinas en paralelo. Aunque este problema puede ser considerado teórico, tiene diversas aplicaciones prácticas siendo la modelización de cuellos de botella en las empresas productoras. La producción de la mayoría de bienes tiene varias etapas, pero es normal que una de ellas sea central en el proceso (decapación iónica en la fabricación de semiconductores, cocido de baldosas cerámicas, etc). Es habitual que las empresas añadan recursos productivos en las etapas que son cuello de botella para así aumentar la productividad. Es aquí donde la programación de la producción se vuelve de gran importancia para la toma de decisiones de asignar los trabajos a las máquinas y el orden de los trabajos asignados a cada máquina que permitan optimizar un objetivo.

El objetivo elegido para este trabajo ha sido la minimización de los adelantos y retrasos con pesos, debido a que el entorno competitivo de las empresas es cambiante de manera acelerada estas deben ser capaces de cumplir con las fechas de entrega acordadas con los clientes. Por lo que incumplir con las fechas de entrega ya sea adelantándose provocando aumento de costos de inventario y retraso en el pago de los clientes o retrasándose resultando en pérdidas de clientes, compensaciones hacia los clientes entre otras penalidades. El adelanto y el retraso han sido ponderados, es decir multiplicados por un peso debido a que no todos los trabajos tienen la misma importancia y se acostumbra a darles prioridad a los trabajos de clientes más importantes. Tanto el retraso como el adelanto no son deseados ni por la empresa ni por los clientes.

Una vez determinado el problema sobre el que se procederá a trabajar, para que este se asemeje más a la realidad, se incluyeron las siguientes restricciones adicionales: las fechas de lanzamiento son los tiempos donde los trabajos pueden comenzar o estar disponibles ya que no siempre los trabajos están listos para procesarse inmediatamente se reciben en el taller, debido a esperas de materia prima o el orden en que fueron recibidos. También se consideran,

tiempos de preparación de las máquinas que dependen de la secuencia y la máquina donde se realice el trabajo, estos tiempos no productivos de las máquinas representan limpiezas, configuraciones o preparaciones para la producción entre trabajos donde si no son programados de manera adecuada pueden significar una gran cantidad de tiempo en la realización de estos. Por último, se tratan también, fechas límite, que son las fechas que no pueden ser violadas, y ayudan a evitar que los trabajos se retrasen más allá de una fecha establecida donde se pueda incurrir en penalidades a la empresa. Con estas motivaciones, procedo a marcar una serie de objetivos a alcanzar en la realización de este trabajo final de máster, que serán detallados a continuación.

1.3 Objetivos

- Realizar un estado del arte que permita conocer distintos métodos existentes para resolver el problema de máquinas en paralelo no relacionadas.

- Resolver el modelo MIP en el solver de optimización Lingo 17.0, generando las instancias necesarias para realizar las evaluaciones.

- Proponer un algoritmo genético que sea simple y efectivo para resolver el problema de máquinas en paralelo no relacionadas con el objetivo de minimizar los adelantos y retrasos ponderados.

- Evaluar variaciones del algoritmo permitiendo o no insertar tiempos ociosos en las máquinas.

El resto de este trabajo final de Máster está estructurado de la siguiente manera:

- **Capítulo 2**: Descripción del problema, se presenta la clasificación $\alpha/\beta/\gamma$ de los problemas de programación de la producción donde se detallan las características consideradas para después describir el problema formalmente de manera más detallada.

- **Capítulo 3**: Estado del arte, se realiza un repaso sobre la literatura relacionada con el problema de máquinas en paralelo no relacionadas con el objetivo de minimizar los adelantos/retrasos ponderados, así como los métodos propuestos para su resolución.

- **Capítulo 4**: Metodología, se plantea un modelo de programación entera lineal entera mixta para el problema de máquinas en paralelo no relacionadas, con fechas de lanzamiento, fechas límite con el objetivo de minimizar los adelantos/retrasos ponderados. Se propone un método heurístico/metaheurístico a través de un algoritmo genético utilizando la regla de despacho (SPT) y soluciones aleatorias válidas para generar la población inicial.

- **Capítulo 5:** Validación, se realizará análisis estadístico para evaluar diferencias entre versiones del algoritmo para determinar cuál proporciona el mejor resultado.

- **Capítulo 6:** Conclusión, se comentan los resultados obtenidos en el trabajo, así como líneas de trabajo futuro.

Capítulo 2. Descripción del problema

2.1 Notación

Antes de describir el problema que se pretende resolver se debe destacar que los problemas de programación son determinados o definidos por características de los trabajos, características de las máquinas y el criterio de optimización. Para la clasificación de estos problemas la notación más frecuentemente utilizada es la propuesta en Graham et al. [15] basada en la tripleta α/β/γ, ver detalles a continuación:

El campo α se refiere a las características de las máquinas este se subdivide a su vez en α1 y α2.

α1 puede tener los valores los siguientes valores:
1 o ∅: una sola máquina.
P: máquinas en paralelo idénticas.
Q: máquinas en paralelo uniformes.
R: máquinas en paralelo no relacionadas.
F: taller de flujo.
J: taller de trabajo.
O: taller abierto.

α2 representa el número de máquinas fijado (si no figura quiere decir que el problema estudiado no está limitado para un numero de máquinas en específico).

El campo β contiene las características de los trabajos, este campo se subdivide en varios subcampos de acuerdo a las características que tenga el problema, debido que son variadas y numerosas, son separados por coma. En el caso de que no estén presente se asume que la característica no se da. Algunas de las más utilizadas son:

- Prec: relaciones de precedencia.

- S_{nsd}: tiempos de preparación no dependientes de la secuencia. Se denotan como S_{ij} y son conocidos y deterministas. Indican la cantidad de tiempo de preparación necesarios en una máquina i antes de procesar un trabajo j.

- S_{sd}: tiempos de preparación dependientes de la secuencia. Para cada máquina se tiene una matriz de tiempos de preparación, donde S_{ijk} es el tiempo de preparación necesario en una máquina i para procesar un trabajo k luego de haber procesado un trabajo j.

- Prmu: el orden de entrada de los trabajos es el mismo para todas las máquinas. Aplica solo para taller de flujo y a otras variantes específicas de distintos problemas de secuenciación.

- Brkdwn: las máquinas sufren averías o tiempos donde deben estar paradas.

- r_j : tiempo en el cual el trabajo j está listo para ser procesado (release date).

- Prmp: los trabajos pueden interrumpirse para continuar luego en la misma u otras máquinas.

- d_j : fecha de entrega de los trabajos (due date). Se permite que los trabajos terminen fuera de esta fecha, pero con penalización.

- $\bar{d}_j$: fecha límite de finalización de los trabajos (deadline). No se pueden terminar los trabajos después de esta fecha.

- M_j : restricción de uso de máquinas. Indica que algunos trabajos solo pueden ser procesados en determinadas máquinas.

- Recrc: indica que por lo menos un trabajo debe ser reprocesado por lo menos en una o varias máquinas.

- No-idle: ninguna máquina puede estar inactiva después que se empiezan a procesar los trabajos.

El campo γ contiene información respecto al criterio de optimización que se utiliza. Algunos de los criterios más comunes son:

$C_{máx}$: minimización del tiempo máximo de finalización de todos los trabajos o makespan.
$\bar{C}$: minimización del tiempo medio de finalización.
$F_{máx}$: minimización del tiempo máximo de flujo.
$\bar{F}$: minimización del tiempo medio de flujo.
$L_{máx}$: minimización de la máxima holgura.
$\bar{L}$: minimización de la holgura media.
$T_{máx}$: minimización de los retrasos máximos.
$\bar{T}$: minimización del retraso medio.
$E_{máx}$: minimización del adelanto máximo.
$\bar{E}$: minimización del adelanto medio.
N_T : minimización del número de trabajos retrasados.
$\sum_{j=1}^{n}(E_j + T_j)$: minimización de la suma de retrasos y adelantos.
$\sum_{i=1}^{m} I_i$: minimización de los tiempos ociosos.

En el problema de programación de máquinas en paralelo se quiere asignar y secuenciar un conjunto N de n trabajos a un conjunto M de m máquinas que están dispuestas en paralelo, donde los trabajos solo tienen una única operación. Se utilizarán los subíndices j y k para los trabajos y el subíndice i para las máquinas. Los problemas de máquinas en paralelo se dividen en:

- Máquinas en paralelo idénticas: significa que no existen diferencias entre el tiempo de procesamiento de un trabajo en una máquina y en otra. Donde los tiempos de procesamiento son denominados p_j.

- Máquinas en paralelo uniformes: donde el tiempo de procesamiento de un trabajo j en una máquina i sigue la relación de p_{ij}/s_i, donde s_i representa las diferentes velocidades para la máquina i cuando está procesando un trabajo, esto quiere decir que existen máquinas que procesan los trabajos más rápido o más lento que las demás.

- Máquinas en paralelo no relacionadas: en donde la velocidad de procesamiento depende de la máquina y del trabajo que se está realizando. Este

es el más general de los casos ya que incluye a los dos anteriores como casos particulares.

En este trabajo final de Máster se eligió el problema de máquinas en paralelo no relacionadas con el objetivo de minimizar los adelantos y retrasos con pesos ponderados. Este criterio de optimización está fuertemente relacionado con la satisfacción del cliente, donde el retraso es definido como $T_j = \text{máx}\{C_j - d_j, 0\}$ y el adelanto se define $E_j = \text{máx}\{d_j - C_j, 0\}$. La programación justo a tiempo (JIT) busca reducir los retrasos y adelantos, completando el trabajo en su fecha de entrega o lo más cerca de su fecha de entrega. Si se termina los trabajos por adelantado se incurre en costos de inventario si el cliente no acepta el pedido hasta la fecha de entrega, así como retraso en recibir el pago por el trabajo realizado y si se termina tarde se incurre en pérdidas de clientes y compensaciones hacia el cliente por haber incumplido la fecha de entrega. Se debe tener en cuenta que un trabajo solo puede ir tarde, adelantado o a tiempo. De acuerdo a la clasificación α/β/γ en [14], este objetivo se denota como $R//\sum_{j=1}^{n}(W_j'E_j + W_jT_j)$ donde se conoce con anterioridad la cantidad de máquinas, trabajos y también la matriz de tiempos de procesamiento de un trabajo j en una máquina i (p_{ij}). Los trabajos están disponibles a partir de su fecha de lanzamiento (r_j). Se asume que cada trabajo solo puede ser asignado a una sola máquina y no se podrá interrumpir una vez este se ha iniciado. Ninguna máquina puede procesar más de un trabajo a la vez, pueden procesar todos los trabajos y siempre están disponibles. Para que el problema sea más realista de acuerdo a lo planteado anteriormente se toman en consideración los siguientes aspectos:

Las fechas de lanzamiento de los trabajos (release dates) denominadas como r_j, es el tiempo en el que el trabajo puede comenzar o estar disponible para procesarse, estos ayudan a modelar las fechas más tempranas de inicio de los trabajos debido a disponibilidad de materia prima o el tiempo en que se puso la orden.

Los tiempos de preparación de las máquinas (setup times) son esos tiempos no productivos de las máquinas necesarios para realizar limpiezas, configuraciones o preparaciones para la producción entre trabajos. Para nuestro trabajo se consideran tiempos de preparación anticipatorios o separables, estos son aquellos que pueden ser aplicados en las máquinas inmediatamente termine el trabajo anterior en la secuencia sin necesidad de que el trabajo siguiente haya llegado a la máquina. Son dependientes de la secuencia y de la máquina, es decir

estos tiempos dependerán del trabajo procesado anterior en la máquina y el siguiente trabajo a procesar. Se denominan S_{ijk}, se tendrá una matriz con los tiempos de preparación que necesita una máquina i para procesar un trabajo k luego de haber procesado un trabajo j. Estos tiempos son asimétricos, o sea, los tiempos de preparación entre trabajos j y k en una máquina i, pueden ser diferentes de tiempos de preparación entre trabajos k y j en otra máquina y deben cumplir la desigualdad triangular $S_{ijk} \leq S_{ijl} + p_{il} + S_{ilk}$. Dada una máquina i, el tiempo de preparación entre los trabajos j y k es igual o menor que el tiempo de preparación entre un trabajo j y cualquier otro trabajo l, el tiempo de procesamiento del trabajo l y el tiempo de preparación entre trabajos l y k.

Las fechas límite (deadlines) que al contrario de las fechas de entrega no pueden ser violadas y todos los trabajos deben ser terminados antes o en esta fecha, cuando se cuenta con fechas de lanzamientos y fechas límite el resultado es llamado “ventana de procesamiento” ya que el tiempo de procesamiento de un trabajo debe ser completado en este intervalo de tiempo $\bar{d}_j - r_j$. Las fechas límite en general y la ventana de procesamiento afectan que se pueda obtener una programación factible, no todas las secuencias de ordenamiento de los trabajos en las máquinas son posibles, cuando se está trabajando con adelantos/retrasos conviene incluir una fecha límite para que no retrase los trabajos más allá de esa fecha cuando este minimizando los retrasos.

Con el objetivo de minimizar los adelantos y retrasos con pesos ponderados, los tiempos de preparación dependientes de la máquina y la secuencia, fechas de lanzamiento y fechas límites, el orden o la secuencia en los trabajos asignados a las máquinas en el problema de máquinas no relacionadas en paralelo se vuelve de gran importancia. Dependiendo de cuando un trabajo j se programe en una máquina, su tiempo de finalización C_j cambiará y por lo tanto los valores de adelantos y retrasos también. Al igual que la cantidad de tiempos de preparación en la máquina también depende de la secuencia. Un aspecto a tomar en consideración para mejorar el criterio de optimización elegido principalmente los adelantos es la inserción de tiempos de tiempos ociosos en las máquinas para retrasar los trabajos que vayan adelantados, cuando el peso de un adelanto es mayor que el peso de un retraso, estos tiempos benefician el objetivo, siempre tomando en cuenta que retrasar un trabajo en una máquina afecta los demás trabajos secuenciados en esta.

Este problema debido a sus características está definido como NP – Hard (difícil), esto significa que no existe un algoritmo en tiempo polinómico capaz de calcular la solución óptima. Para resolver estos problemas de forma habitual

se emplean algoritmos o heurísticas que permiten obtener buenas soluciones en tiempos computacionales razonables.

Un ejemplo de este problema para 6 trabajos en 2 máquinas es:

Tabla 2. 1 : Datos para el ejemplo de 6 trabajos en 2 máquinas.

ID	Tiempos de Procesamiento		Pesos para retrasos	Pesos para adelantos	Fechas de entrega	Fechas de Lanzamiento	Fechas límite
	1	2					
1	60	53	9	8	212	14	292
2	58	80	3	2	58	9	138
3	80	58	6	5	205	15	285
4	89	83	6	8	164	9	244
5	62	16	1	1	437	34	537
6	65	21	10	3	133	37	213

Tabla 2. 2 : Secuencia de los trabajos en las máquinas con su valor de la función objetivo.

ID	Máquina Asignada	Inicio	Cj	Sijk	Fechas de entrega	Fechas límite	Retraso Ponderado	Adelanto Ponderado
2	2	9	89	0	58	138	93	0
4	2	91	174	2	164	244	60	0
3	2	214	272	40	205	285	402	0
5	2	322	338	50	437	537	0	99
6	1	37	102	0	133	213	0	93
1	1	152	212	50	212	292	0	0

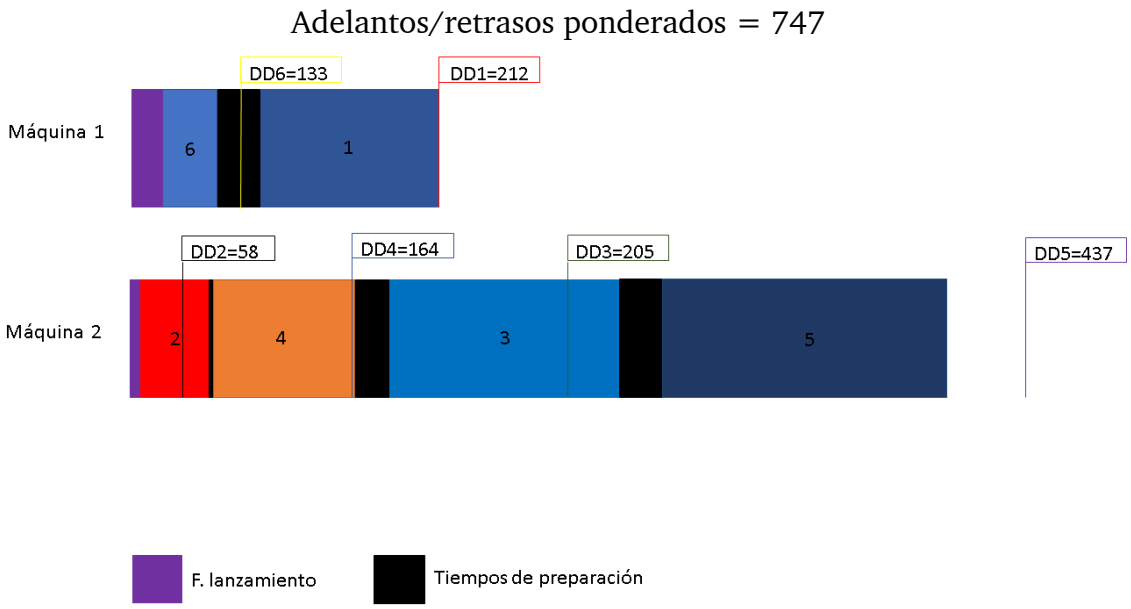

Figura 2. 1 : Secuencia de 6 trabajos y 2 máquinas.

Capítulo 3. Estado del arte

3.1 Estado del arte

El problema considerado en este trabajo es máquinas en paralelo no relacionadas con fechas de lanzamiento, fechas límite y tiempos de preparación dependiente de la secuencia y la máquina con el objetivo de minimizar los adelantos/retrasos ponderados se denota como $R/r_j, \bar{d}_j, S_{ijk}/\sum_{j=1}^{n}(W_j' E_j + W_j T_j)$, al realizar una búsqueda en la literatura sobre el problema de programación en máquinas en paralelo se puede constatar que tanto las máquinas idénticas y uniformes han sido más estudiadas a lo largo de los años en comparación con las no relacionadas. En algunas revisiones realizadas por autores como: Cheng y Sin [14] y Mokotoff [34] es posible apreciar esto. Habiendo constatado esto la literatura sobre programación en máquinas en paralelo es extensa, por lo que sí podemos comentar artículos que están más relacionados con el tipo de máquina y objetivo a optimizar en este trabajo.

Ya desde 1959, con McNaughton [33] se viene estudiando el problema de máquinas en paralelo no relacionadas. Los adelantos y retrasos o programación justo a tiempo(JIT) es un campo sujeto a diversas investigaciones con revisiones como la de Baker y Scudder [8], Cheng y Gupta [13], Zhu y Meredith [47]. En la revisión proporcionada por Baker y Scudder, la mayoría de los modelos que incorporan adelantos y retrasos con penalidades son de una sola máquina que contienen fechas de entrega (due dates) comunes para todos los trabajos. Los resultados indican que cuando la fecha de entrega es igual para todos los trabajos, la programación óptima es en forma de V sin insertar tiempos ociosos entre los trabajos. Un trabajo se completa en su fecha de entrega y otro inicia en esa fecha. Zhu y Meredith [46] resaltan los aspectos claves de la implementación de la filosofía justo a tiempo(JIT) en las industrias, siendo la programación justo a tiempo una de ellas. Kanet [25] examina el problema de los adelantos y retrasos, para una sola máquina, con iguales penalidades y fechas de entrega comunes no restringidas. Un problema es considerado no restringido cuando su fecha de entrega es lo suficientemente grande para no restringir el problema. Introdujo un algoritmo en tiempo polinomial para encontrar una solución óptima.

Otros artículos donde se estudia minimizar el objetivo igual o similares son Heady y Zhu [22] que buscan minimizar los adelantos/retrasos de los trabajos en máquinas idénticas con tiempos de preparación que se denota por P/ S_{jk}/ $\sum_{j=1}^{n}(E_j + T_j)$. También Balakrishnan et al. [9], Zhu y Heady [47], Omar y Teo [36] y Şerifoğlu y Ulusoy [41] estudian un problema de máquinas uniformes Q/r_j, S_{ijk}/ $\sum_{j=1}^{n}(W_j'E_j + W_jT_j)$ con ciertas diferencias. Balakrishnan et al. [9] toman en consideración tiempos de lanzamiento y tiempos de preparación dependientes de la máquina y la secuencia presentando un modelo matemático de programación lineal entera mixta (MIP). Zhu y Heady [46] no toman en consideración los tiempos de lanzamiento para este problema. Omar y Teo [36] comparan su modelo propuesto con el anterior de Zhu y Heady [46] demostrando que el modelo anterior estaba incompleto por lo que añadieron una restricción. Y por último Şerifoğlu y Ulusoy [41] a diferencia de Balakrishnan et al. [9] presentan dos algoritmos genéticos uno con un operador de cruce otro sin cruce, donde concluyen que el algoritmo genético con operador de cruce tiene mejor rendimiento cuando son problemas grandes y más difíciles.

Bank y Werner [10] proponen diversos algoritmos heurísticos para R/r_j, d_J/ $\sum_{j=1}^{n}(W_j'E_j + W_jT_j)$ llegan a la conclusión comparando las diferentes heurísticas que ninguna es superior. Rocha et al. [38] proponen un algoritmo utilizando la metaheurística GRASP para maquinas paralelas con tiempos de preparación dependientes de la secuencia y de la máquina. En Kim et al. [28] se estudia la minimización de los retrasos máximos en máquinas en paralelo no relacionadas con tiempos de preparación. Posteriormente en Kim et al. [29] eligen el objetivo de minimizar los retrasos ponderados. Vallada y Ruiz [42] proponen un algoritmo genético para máquinas en paralelo no relacionadas con tiempos de preparación dependiente de la secuencia y la máquina con el objetivo de minimizar el makespan luego Vallada y Ruiz [43] proponen el algoritmo genético para el problema R/ S_{ijk}/ $\sum_{j=1}^{n}(W_j'E_j + W_jT_j)$ donde consideran la inserción de tiempos ociosos en las máquinas (idle time) para beneficiar los adelantos.

Bajestani y Tavakkoli [7] proponen un nuevo algoritmo B&B para el problema de máquinas en paralelo no relacionadas con tiempos de preparación dependiente de la secuencia y concluyen que obtienen la solución óptima hasta 10 trabajos con 4 máquinas en todas las combinaciones de los factores de retraso y rangos de fecha de entrega. Avdeenko y Mesentsev [6] proponen acercamientos eficientes para el problema de máquinas en paralelo no relacionadas con fechas de lanzamiento con el objetivo de minimizar el

makespan. Liaw et Al. [30] proponen un algoritmo (B&B) para el problema de máquinas en paralelo no relacionadas minimizando los retrasos ponderados usando una cota inferior y otras heurísticas para la cota superior.

Weng et Al. [44] estudian el mismo problema que Zhu y Heady [46] pero con dos diferencias: el criterio estudiado es el tiempo de finalización medio y los tiempos de preparación solo dependen de la secuencia y no de la máquina. Cheng y Huang [12] proponen un algoritmo genético híbrido con tiempos de lanzamiento controlados para el problema de máquinas en paralelo no relacionadas para minimizar los adelantos y retrasos ponderados y concluyen que el algoritmo supera a un modelo MIP propuesto y un algoritmo genético clásico. Joo y Kim [24] proponen un algoritmo genético híbrido con reglas de despacho para máquinas en paralelo no relacionadas con tiempos de preparación dependientes de la secuencia y las máquinas. Kayvanfar et al. [27] estudian el problema de máquinas en paralelo no relacionadas con el objetivo de minimizar los retrasos y adelantos ponderados, pero con tiempos de procesamiento controlables. Longedran et al. [32] proponen diversos métodos de búsqueda tabú para el problema de máquinas paralelas no relacionadas con tardanza ponderada. Mokotoff y Jimeno [35] proponen una heurística basada en enumeraciones parciales para el problema de máquinas en paralelo no relacionadas, con esta novedosa metodología lograron excelentes resultados resolviendo hasta 200x20 en poco tiempo de CPU. Fanjul-Peyro y Ruiz [17] proponen una serie de metaheurísticas basadas en reducción del tamaño al problema original obteniendo buenos resultados comparados con otros métodos. Posteriormente Fanjul-Peyro y Ruiz [18] proponen 3 algoritmos para maquinas paralelas no relacionadas cuando no todas las maquinas están disponibles obteniendo buenos resultados comparándolos con un modelo MIP. Nogueira et al. [15] proponen un algoritmo GRASP con variaciones para resolver el problema de minimizar los adelantos/retrasos ponderados en máquinas en paralelo no relacionadas.

Avalos-Rosales, et al. [5] proponen un algoritmo metaheurístico basado en un algoritmo multi-inicio y vecindarios variables, así como nuevas formulaciones de programación entera lineal mixta para maquinas en paralelo con tiempos de preparación dependientes de la secuencia y máquina con el objetivo de minimizar el makespan. Diana et al. [16] proponen un algoritmo inmuno-inspirado cuya población inicial es generada a partir de un GRASP para el mismo problema que Avalos-Rosales. Fanjul-Peyro, et al. [19] estudian el problema de máquinas en paralelo donde el procesamiento de los trabajos en las máquinas requiere un número de unidades de un recurso escaso, proponen dos modelos

MILP y tres metaheurísticas para resolver el problema. Rodríguez, et al. [39] proponen un algoritmo iterativo voraz para máquinas no relacionadas con el objetivo de minimizar el tiempo de finalización total ponderado para grandes instancias.

Zheng y Wang [45] para el problema de máquinas no relacionadas con restricciones adicionales de recursos presentan un modelo MILP y un algoritmo de optimización de dos fases de mosca de la fruta. Sels, et al. [40] proponen un algoritmo genético, un algoritmo de búsqueda tabú para máquinas en paralelo con el objetivo de minimizar el makespan. Lin y Lin [31] proponen algoritmo de búsqueda tabú para el problema de máquinas no relacionadas tomando en cuenta fechas de lanzamiento con dos objetivos de minimizar el makespan y los retrasos ponderados totales.

Tras revisar y comentar parte de la literatura relacionada con el problema de máquinas paralelas no relacionadas si se toma en cuenta las restricciones de release date, deadlines y setups con el objetivo de minimizar los adelantos/retrasos ponderados, se puede decir que es escasa la literatura que combina estas restricciones, aun así se pueden encontrar una gran diversidad de trabajos que propuestos para maquinas paralelas no relacionadas con diferentes objetivos a optimizar que proporcionan una buena idea para su solución.

Capítulo 4. Metodología

4.1 Modelo Matemático (MILP)

Se plantea un modelo matemático de programación entera lineal mixta para máquinas paralelas no relacionadas con tiempos de preparación dependiente de la máquina y la secuencia de los trabajos, fechas de lanzamiento y fechas límite de entrega. Este modelo es una adaptación del propuesto por Vallada and Ruiz [43].

Para el problema:

$$\mathrm{R}/r_j, \bar{d}_j, S_{ijk}/\sum_{j=1}^{n}(W_j' E_j + W_j T_j)$$

Variables

$$X_{ijk} = \begin{cases} 1, \text{si el trabajo j precede al trabajo k en la máquina i} \\ 0, \text{sino} \end{cases}$$

C_j = Tiempo de finalización del trabajo j.
E_j = Adelanto del trabajo j.
T_j = Retraso del trabajo j.

Parámetros

p_{ij} = Tiempo de procesamiento del trabajo j en la máquina i.
r_j = fecha de lanzamiento del trabajo j.
$\bar{d}_j$ = fecha límite del trabajo j.
d_j = fecha de entrega del trabajo j.
W_j' = Peso del Adelanto.
W_j = Peso del retraso.
S_{ijk} = la cantidad de tiempo de preparación que necesita una máquina i luego de haber procesado un trabajo j y el siguiente trabajo en la secuencia es el trabajo k.

Función Objetivo

$$mín \sum_{j=1}^{n} (W_j' E_j + W_j T_j)$$

Restricciones

$$\sum_{i \in M} \sum_{\substack{j \in \{0\} \cup \{N\} \\ j \neq k}} X_{ijk} = 1, \qquad \forall k \in N, \qquad (1.1)$$

$$\sum_{i \in M} \sum_{\substack{k \in N \\ j \neq k}} X_{ijk} \leq 1, \qquad \forall j \in N, \qquad (1.2)$$

$$\sum_{k \in N} X_{i0k} \leq 1, \qquad \forall i \in M, \qquad (1.3)$$

$$\sum_{\substack{h \in \{0\} \cup \{N\} \\ h \neq k, h \neq j}} X_{ihj} \geq X_{ijk}, \qquad \forall j, k \in N, j \neq k, \forall i \in M, \qquad (1.4)$$

$$C_k + V(1 - X_{ijk}) \geq C_j + S_{ijk} + p_{ik}, \qquad \forall j \in \{0\} \cup \{N\}, \forall k \in N, \ j \neq k, \forall i \in M, \qquad (1.5)$$

$$C_k + V(1 - X_{ijk}) \geq r_k + p_{ik}, \qquad \forall j \in \{0\} \cup \{N\}, \forall k \in N, \ j \neq k, \forall i \in M, \qquad (1.6)$$

$$C_j + E_j - T_j = d_j \qquad \forall j \in N, \qquad (1.7)$$

$$C_j \leq \overline{d}_j \qquad \forall j \in N, \qquad (1.8)$$

$$C_j \geq 0 \qquad \forall j \in N, \qquad (1.9)$$

$$E_j \geq 0 \qquad \forall j \in N, \qquad (1.10)$$

$$T_j \geq 0 \qquad \forall j \in N, \qquad (1.11)$$

$$X_{ijk} \in \{0,1\}, \qquad \forall j \in \{0\} \cup \{N\}, \forall k \in N, \ j \neq k, \forall i \in M, \qquad (1.12)$$

Donde $C_0 = S_{i0k} = 0 \ \forall i \in M, \forall k \in N$. El objetivo es minimizar la suma de los adelantos/retrasos ponderados.

1.1. Asegura que cada trabajo tenga exactamente un predecesor en una máquina, incluyendo el trabajo inicial dummy "0" que puede estar en cada máquina.

1.2. Establece el máximo número de sucesores de cada trabajo a máximo uno (la excepción de un trabajo sin sucesor sería el último trabajo asignado a una máquina).

1.3. Limita el número de sucesores de los trabajos dummy X_{i0k} a 1 en cada máquina.

1.4. Garantiza que los trabajos estén relacionados adecuadamente en la máquina, es decir si un trabajo j es procesado en una máquina i, un predecesor valido h debe existir en la misma máquina.

1.5. Controla el tiempo de finalización de los trabajos en las máquinas, si un trabajo k es asignado a una máquina i después de un trabajo j ($X_{ijk} = 1$), su tiempo de finalización C_k tiene que ser mayor que el tiempo de finalización j, C_j, más el tiempo de preparación entre j y k, el tiempo de procesamiento de k. V es una constante grande.

1.6. Toma en consideración las fechas de lanzamiento de los trabajos k.

1.7. Define el adelanto y el retraso, por lo que un trabajo solo puede ir o adelantado o retrasado.

1.8. Controla los tiempos de finalización de los trabajos j que deben ser menor o igual que las fechas límite.

1.9, 1.10 y 1.11. Establecen la no negatividad de las variables.

1.12. Define los valores que toman las variables binarias.

4.1.1 Implementación del modelo MILP

Para resolver el modelo MILP se ha utilizado un solver de optimización llamado Lingo. Este al igual que otros solvers de optimización utilizan una estrategia de Ramificación y Acotación (Branch *&* Bound) para resolver problemas de programación entera lineal y programación entera lineal mixta. La Ramificación y acotación es un método sistemático que realiza la enumeración de forma implícita de modo que es necesario examinar todas las combinaciones de valores de las variables. Cuando se está minimizando una función objetivo, las ramas o combinaciones que son descartadas tienen un valor de la función objetivo mayor que el límite inferior calculado.

Los algoritmos de ramificación y acotación comienzan resolviendo el problema original como un problema de programación lineal, relajando las condiciones de que las variables deben tomar valores discretos.

Si no se obtiene solución entera, una de las variables enteras que todavía tiene valor continuo, denotada por xs=a.b se selecciona y se crean dos descendientes del problema original, uno en el que xs≥a+1 y otro en el que xs≤a. Esta operación se denomina Ramificación. El proceso se repite seleccionando uno de los problemas como el actual problema de programación entera y tratándolo exactamente igual que el original. Esto a su vez genera dos nuevos problemas que reemplazan al anterior a menos que por ejemplo el problema en curso no tenga solución posible.

Repitiendo iterativamente el proceso se alcanza una solución entera (si existe) para uno de los actuales problemas que se convierte en candidata para ser la solución óptima del problema original. La mejor de estas soluciones candidatas es una forma de eliminar descendientes del problema original que es inútil explorar porque no conducirían a la solución óptima. La mejor solución entera candidata se llama incúmbente y el proceso de eliminación se denomina Poda.

El código para la implementación de $R/r_j, \bar{d}_j, S_{ijk}/\sum_{j=1}^{n}(W_j'E_j + W_jT_j)$. En Lingo 17.0 se muestra en el listado 1. En el capítulo 5 Validación se encuentran los detalles de los datos y parámetros usados para resolver el modelo.

```
DATA:
num_jobs = 6;
num_machines = 2;
ENDDATA

SETS:
JOBS /0,1,2,3,4,5,6/:C, Earliness, Tardiness, dd, dl, rd, we, wt; !Completion time, !dd=due dates,
dl=deadlines, rd=release dates, wa= weight earliness, wr = weight tardiness;
MACHINE /1..num_machines/:V; !V=upper bound that represents the occupation of a given
machine i;
JobMachine (MACHINE, JOBS): Proc_time; !Processing time;
MachineJobJob (MACHINE, JOBS, JOBS): X, S; ! X variable, S=setup time for machine i for
processing a job j when job k is next in sequence;
ENDSETS

DATA:
Proc_time = @OLE('C:\Users\jeffr\Desktop\TFM\TestyResultados.xlsx', 'Proc_time' ); !loads
Processing times(i,j);
dd = @OLE('C:\Users\jeffr\Desktop\TFM\TestyResultados.xlsx', 'dd' ); !loads due dates;
dl = @OLE('C:\Users\jeffr\Desktop\TFM\TestyResultados.xlsx', 'dl' );   !loads deadlines;
rd = @OLE('C:\Users\jeffr\Desktop\TFM\TestyResultados.xlsx', 'rd' );   !loads release dates;
we= @OLE('C:\Users\jeffr\Desktop\TFM\TestyResultados.xlsx', 'we' );  !loads weight for
earliness;
wt= @OLE('C:\Users\jeffr\Desktop\TFM\TestyResultados.xlsx', 'wt' );   !loads weight for
tardiness;
S=   @OLE('C:\Users\jeffr\Desktop\TFM\TestyResultados.xlsx', 'S' );   !loads setup times;

ENDDATA

@FOR(MACHINE(i) :V=@SUM(JOBS(J) | J#GT#1:Proc_time(i, j) + @MAX(JOBS(K) | K#GT#1:S(i, j,
k)))); !upper bound that represents the occupation of a  given machine i.

!Objective function;
MIN = @SUM(JOBS( j ):we( j )*Earliness( j ) + wt( j )*Tardiness( j ));

!Subject to;

@FOR(JOBS( k ) | k#GT#1 :[Const1]@SUM(MACHINE(i):@SUM(JOBS(J) |  j#NE#k :X(i, j,k)))=1);
!Set(1.1);

@FOR(JOBS( j ) | j#GT#1:[Const2]@SUM(MACHINE(i):@SUM(JOBS(k) | k#GT#1 #AND#  j#NE#k
:X(i, j ,k)))<=1); !Set(1.2);

@FOR(MACHINE( i ):[Const3]@SUM(JOBS(K)| k#GT#1 :X(i,1,k))<=1); !Set(1.3);

@FOR(JOBS(j):@FOR(JOBS(k)| J#GT#1 #AND# K#GT#1 #AND#
J#NE#k:@FOR(MACHINE(i):[Const4] @SUM(JOBS(h) | h#NE#j #AND# h#NE#k: X(i, h , j)) >= X(i , j
,k)))); !Set(1.4);

@FOR(JOBS(J):@FOR(JOBS(K)| j#NE#k #AND# k#GT#1:@FOR(MACHINE(i): [Const5] C(k) + V * (1-
X(i , j ,k))  >= C(j) + S(i,j,k) + Proc_time(i,k)  ) ) ); !Set(1.5);

@FOR(JOBS( j ):@FOR(JOBS( k ) | j#NE#k #AND# k#GT#1:@FOR(MACHINE(i):[Const6] C(k) + V *
(1-X(i , j ,k)) >= rd(k) + Proc_time(i,k) ) ) ); !Set(1.6);
```

```
@FOR(JOBS( j ) | j#GT#1:[Const7] C( j) + Earliness( j ) - Tardiness( j ) = dd( j )); !Set(1.7);
@FOR(JOBS( j ) | j#GT#1:[Const8] C( j) <= dl( j )); !Set(1.8);
@FOR(JOBS( j ) | j#GT#1:[Const9] C( j) >=0); !Set(1.9);

@FOR(JOBS( j ) | j#GT#1:[Const10] Earliness( j ) >= 0); !Set(1.10);

@FOR(JOBS( j ) | j#GT#1:[Const11] Tardiness( j ) >= 0); !Set(1.11);

@FOR(MACHINE(i ):@FOR(JOBS(k) | K#GT#1 :[Const12] C(1) = 0)); !completion time of job 0=0;

@FOR(JOBS(j):@FOR(JOBS(k):@FOR(MACHINE(i) | k#GT#1 #AND# J#NE#K:@BIN(X(i , j ,k)))));
!binary variable if job 1 precedes job k in machine i, otherwise
0;
 @FOR(JOBS(J):@GIN(Earliness(j)));
 @FOR(JOBS(J):@GIN(Tardiness(j)));
```

Listado 1: Código en Lingo 17.0 para: R/$r_j, \bar{d}$j , $S_{ijk}/\sum_{j=1} n$ $(W_j'E_j + W_jT_j)$.

4.2. Propuesta Heurística/Meta Heurística

Una heurística es un procedimiento que proporciona una buena solución de manera intuitiva a un problema matemático bien definido sin garantizar que esta sea óptima. La pérdida de optimalidad se compensa con un tiempo de ejecución razonable, cosa que en un modelo matemático puede no ser el caso, especialmente en problemas de tamaño medio o grande.

Los procedimientos metaheurísticos son una clase de métodos aproximados que están diseñados para resolver problemas difíciles de optimización combinatoria. Los metaheurísticos proporcionan un marco general para crear nuevos algoritmos híbridos combinando diferentes conceptos derivados de la inteligencia artificial, la evolución biológica y la mecánica estadística. Se trata de algoritmos también heurísticos pero que hacen uso de la mayor capacidad computacional actual para dar en soluciones que se acercan mucho a la optimalidad. Siguen siendo más rápidos que los métodos exactos, pero no tanto como las heurísticas, a la par que mucho más complejos.

En este trabajo final de máster se propone un algoritmo genético (genetic algorithm o GA) para el problema R/$r_j, \bar{d}_j$, $S_{ijk}/\sum_{j=1}^{n}(W_j'E_j + W_jT_j)$. Los algoritmos genéticos son métodos de búsqueda estocástica e iterativa que se

basan en los mecanismos de evolución natural y selección de las especies. Están clasificados como métodos bio-inspirados. Los principios básicos fueron establecidos por Holland [23] y se ha usado en trabajos como Goldberg [20], Reeves [37], Vallada y Ruiz [42], Vallada y Ruiz [43], Şerifoğlu y Ulusoy [41], Chang et al. [10] entre otros. Solo en el campo de la secuenciación en problemas de máquinas se podrían contar por miles los trabajos donde se han empleado algoritmos de tipo genético o variantes relacionadas.

Los GA utilizan operadores de selección, cruce y mutación en la generación y exploración de puntos en el espacio de posibles soluciones creando una nueva población, que estará compuesta de un conjunto de individuos (soluciones). Una solución está compuesta por un cromosoma y este a su vez por genes, donde se codifican los elementos de la solución del problema a resolver. Este conjunto de individuos es llamado población. Cada miembro de esa población es un individuo. Los individuos son evaluados y asignados un valor de la función de aptitud (fitness function). Esta función debe ser diseñada para cada problema. Las mejores soluciones para el problema están correlacionadas con valores altos de la función de aptitud, pero esto depende del problema. Una vez asignados los valores de la función de aptitud se utiliza la selección este debe estar sesgada hacia los mejores individuos de la población, es decir que las mejores soluciones tienen mejores probabilidades de ser elegidas. Estos individuos seleccionados se les llama padres y se reproducen mediante otro operador llamado cruce o combinación. En este cruce se generan nuevas soluciones con el propósito de identificar las mejores características para generar mejores soluciones o hijos, luego se aplica un operador de mutación esto es para lograr la creación de una población más diversificada.

Los algoritmos genéticos están orientados a que los individuos de la población encuentren mejores valores de la función objetivo para las soluciones, de manera que en cada iteración solo se vayan preservando las mejores soluciones. Estos presentan una gran ventaja porque permiten alcanzar un equilibrio entre la eficiencia y eficacia para resolver diferentes y complejos problemas de grandes dimensiones.

A continuación, se presenta el diagrama de flujo del algoritmo genético propuesto:

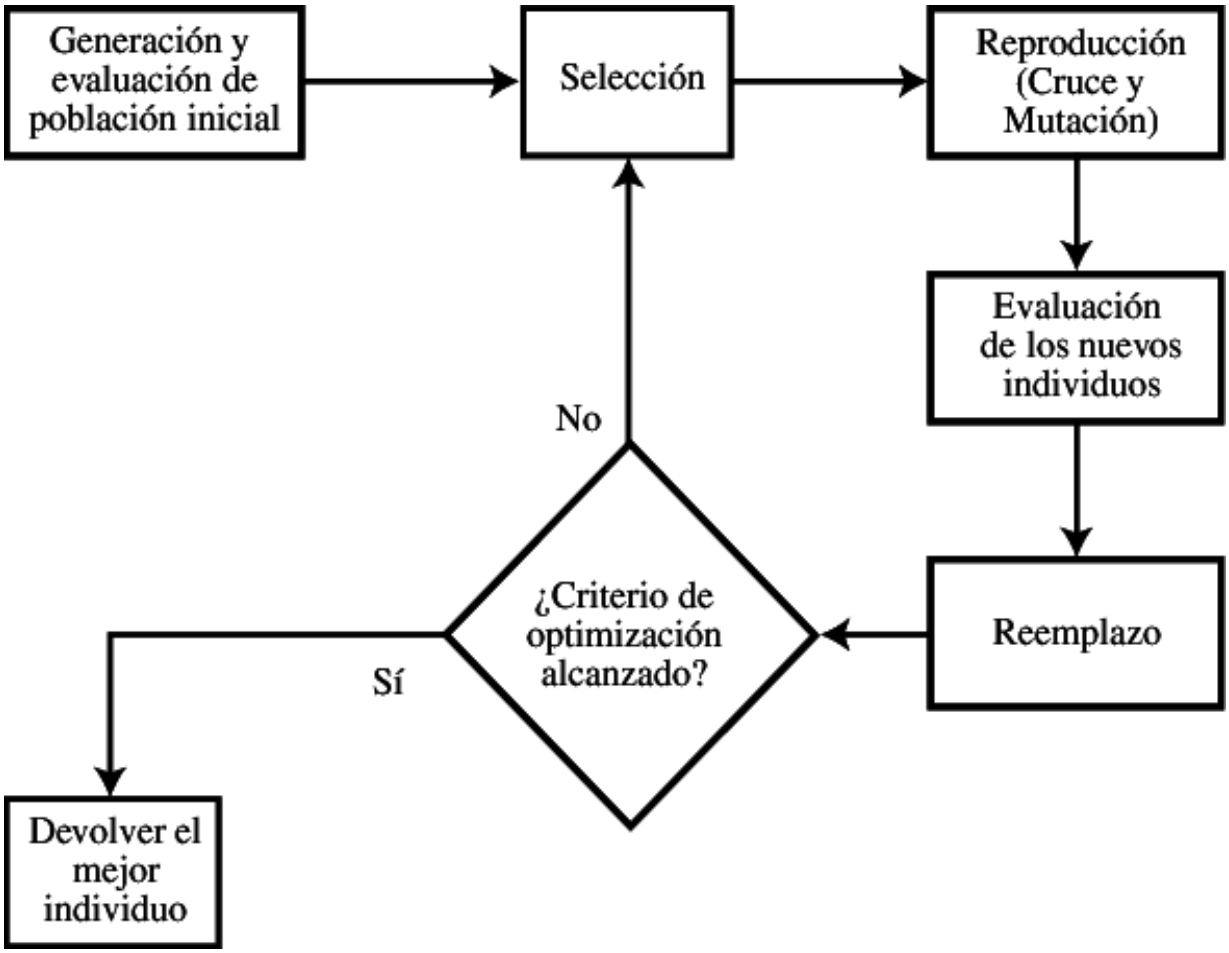

Figura 4. 1: Diagrama de flujo Algoritmo Genético.

4.2.1 Representación de la solución e inicio de la población

La representación de la solución es una de las decisiones más importantes a la hora de implementar el algoritmo genético. Para este tipo de problemas se representan listas de los trabajos para cada máquina representando el orden en que son procesados los trabajos en las máquinas. El algoritmo genético estará conformado por una población de individuos, donde cada individuo tendrá m listas de trabajos.

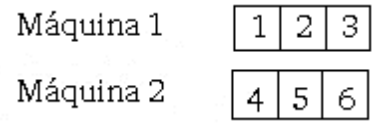

Figura 4. 2 : Representación de las soluciones. Ejemplo: 6 trabajos x 2 máquinas.

Para el inicio de la población se utilizará este procedimiento:

1) Generar individuos de manera aleatoria, pero asignando cada trabajo a la máquina en que menos tiempo de procesamiento tarda (SPT) validando que ningún trabajo exceda su fecha límite.

El tiempo de procesamiento más corto (Shortest Processing Time) es una regla de despacho que determina el orden de prioridad en el que los trabajos serán procesados, de acuerdo a su importancia. Para un tiempo t, el trabajo con el mínimo valor de tiempo de procesamiento (p_j) es procesado primero. Dado que estamos utilizando máquinas paralelas entonces para un tiempo t, el trabajo (j) es realizado en la máquina (i) donde menor valor de tiempo de procesamiento tenga. Para evitar obtener soluciones infactibles se ha establecido un procedimiento que compruebe la secuencia de cada máquina para que los trabajos no excedan su fecha límite, en caso de que excedan la fecha límite se siguen probando otras combinaciones. Se llama validar y en la implementación del algoritmo estará detallado. La programación basada en reglas de despacho es una de las formas más conocidas y clásicas de programación. Las reglas de despacho son útiles a la hora de encontrar una solución razonablemente buena y son utilizadas para generar una secuencia inicial cuando se está aplicando otras heurísticas/metaheurísticas.

4.2.2 Operador de selección

El operador de selección es el mecanismo mediante el cual se eligen los padres que más adelante serán combinados en el operador de cruce para crear nuevos individuos. Se utilizará la selección de los padres mediante torneo binario donde se selecciona una parte de la población al azar y de esta selección se elige el mejor individuo de estos como el primer padre, es decir el que tenga el menor

valor de adelantos y retrasos ponderados. El mismo procedimiento es repetido para obtener el segundo padre, pero sin tomar en cuenta el primer padre seleccionado para evitar repetición. La cantidad de individuos seleccionados en cada torneo serán 2. Existen muchos operadores de selección posibles, pero según Vallada y Ruiz [42] este es el que mejor funciona y por eso se selecciona.

4.2.3 Operador de cruce

El objetivo del operador de cruce es generar hijos o nuevos individuos a partir de dos buenos padres seleccionados con el operador de n-torneos. Este se aplica de acuerdo a una probabilidad (p_c). Si el valor de una variable uniforme entre 0 y 1 es menor que la probabilidad establecida (p_c) se realiza el cruce. El cruce utilizado será el cruce de un punto. Este cruce elige un punto de corte aleatorio en una posición en ambos padres. Los trabajos antes de este punto de corte son heredados del primer padre al primer hijo y del segundo padre al segundo hijo. Los trabajos luego del punto de corte son heredados del primer padre al segundo hijo en el mismo orden en que aparecen en el primer padre. Un proceso similar es llevado a cabo para heredar del segundo padre al primer hijo.

Para aplicar este operador de cruce a máquinas en paralelo es necesario adaptarlo, se selecciona un grupo de trabajos de manera aleatoria en cada máquina del primer padre y son copiados al primer hijo. Una vez se copian los primeros trabajos estos son marcados para evitar trabajos duplicados en los individuos. Los trabajos faltantes son copiados desde el padre 2 en el orden en que aparecen al hijo.

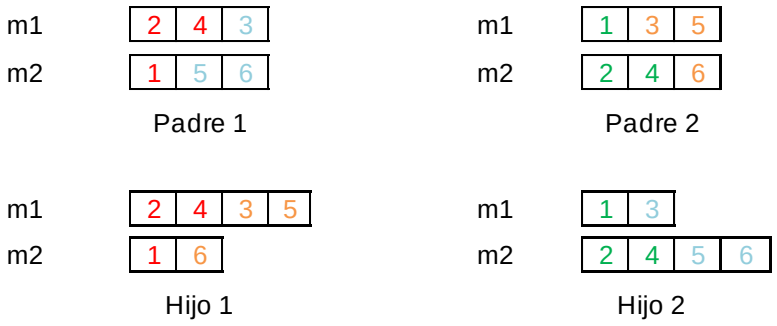

Figura 4. 3 : Ejemplo de operador de cruce para 6 trabajos y 2 máquinas.

4.2.4 Operador de mutación

Una vez se obtienen los dos hijos se procede aplicar el operador de mutación de acuerdo a una probabilidad (p_m). Este operador se utiliza para mantener e introducir diversidad en la población. Esta es la parte del algoritmo genético relacionada con la exploración en el espacio de búsqueda. Se utilizará una mutación de inserción, donde se elige una máquina de manera aleatoria, un trabajo aleatorio dentro de esa máquina y es reinsertado en una posición aleatoria diferente en la misma máquina. Se elige en la misma máquina de acuerdo a Vallada y Ruiz [42] proporciona los mejores resultados.

Cabe destacar que luego de que se realicen tanto el cruce como la mutación antes de agregar los hijos a la nueva población estos son validados mediante la función validar() mencionada anteriormente, evitando obtener secuencias infactibles donde los trabajos violen las fechas límite.

4.2.5 Inserción de tiempos ociosos

Mantener deliberadamente recursos inactivos puede ser deseable en muchas situaciones. Cuando hay trabajos que llegan y el recurso es un procesador a granel (por ejemplo, un horno), o cuando hay una actividad urgente que no puede comenzar inmediatamente pero que se retrasaría adversamente si otros trabajos menos críticos comenzaran. Otra de estas situaciones y que es considerada en este trabajo es cuando se está programando bajo la filosofía justo a tiempo(JIT) ya que se desea que los trabajos sean entregados en su fecha acordada, ni antes ni después.

En la revisión realizada por Kanet y Sridharan [26] se puede constatar esto. En la programación Justo a Tiempo (JIT) donde se busca minimizar los adelantos y retrasos es necesario la inserción de tiempos ociosos en las máquinas, esto significa no empezar a procesar el trabajo inmediatamente esté disponible, ya que si se termina antes existen penalidades por crear inventario de producto terminado.

El objetivo puede ser mejorado cuando se retrasa un trabajo que este adelantado, siempre y cuando el peso de los trabajos adelantados sea menor que el peso de los trabajos retrasados debido a que retrasar el inicio de un trabajo

afecta la secuencia en la máquina donde se encuentre ese trabajo. El procedimiento a utilizar es ir maquina a máquina en las secuencias de trabajos y encontrar el primer trabajo que este adelantado, calcular el sumatorio total de los adelantos y retrasos ponderados, luego si la suma de los adelantos ponderados es mayor que los retrasos ponderados entonces los tiempos ociosos se insertan antes del primer trabajo adelantado logrando que después de esta inserción al menos un trabajo este completado a tiempo. El criterio de parada es hasta que los adelantos ponderados sean menores que los retrasos ponderados. El procedimiento será aplicado luego de que el algoritmo haya terminado al mejor individuo seleccionado.

Procedimiento para insertar tiempo ocioso.
Para cada máquina m:
1.Se busca el primer trabajo que este adelantado en m.
2.Se calcula la sumatoria de los adelantos multiplicados por el peso de los trabajos siguientes. (adelanto ponderado total).
3.Se calcula la sumatoria de los retrasos multiplicados por el peso de los trabajos siguientes. (retraso ponderado total).
4.Se elige el menor adelanto de todos los trabajos adelantados en la máquina (mínimo adelanto).
Si el adelanto ponderado total es mayor que el retraso ponderado total entonces:
4.Inserto el mínimo adelanto antes del primer trabajo adelantado. Es decir, al tiempo de inicio del primer trabajo adelantado se le suma el valor del mínimo adelanto.
5.Calcula los nuevos valores de (adelanto ponderado total) y (retraso ponderado total).
Hasta que los adelantos sean menores que los retrasos en las máquinas.

Figura 4.4: Seudo-Código para la inserción de tiempos ociosos.

Para el ejemplo de 6 trabajos y 2 máquinas en las figuras 4.5 y 4.6 se observa la secuencia de los trabajos con sus fechas de entrega (ddj), fechas de lanzamiento (rj) y tiempos de preparación (Sijk) antes y después de insertar los tiempos ociosos en las máquinas.

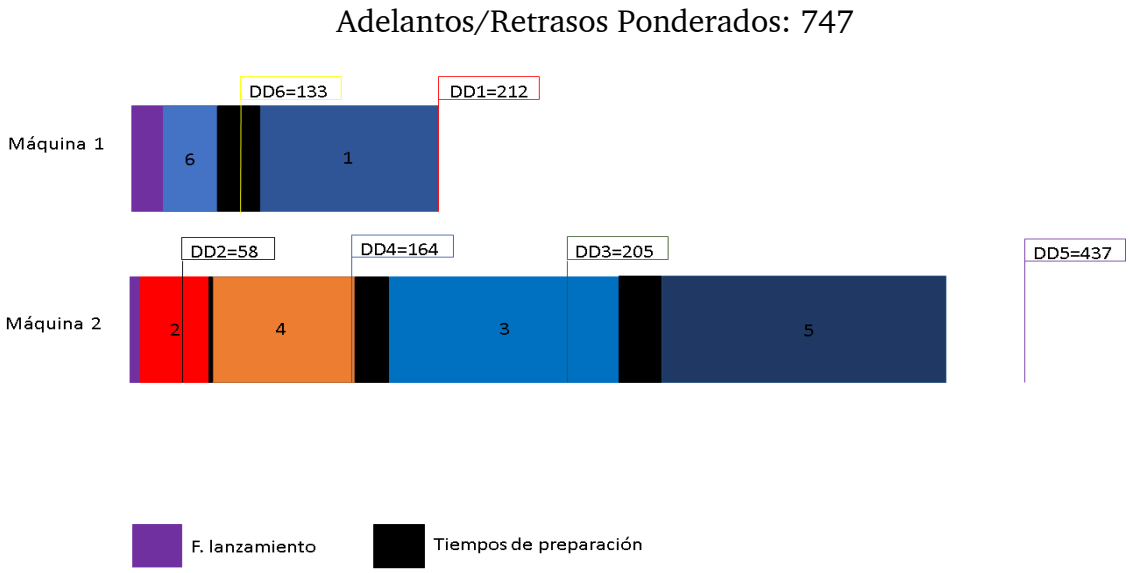

Figura 4. 5 : Ejemplo de 6 trabajos y 2 máquinas sin tiempos ociosos.

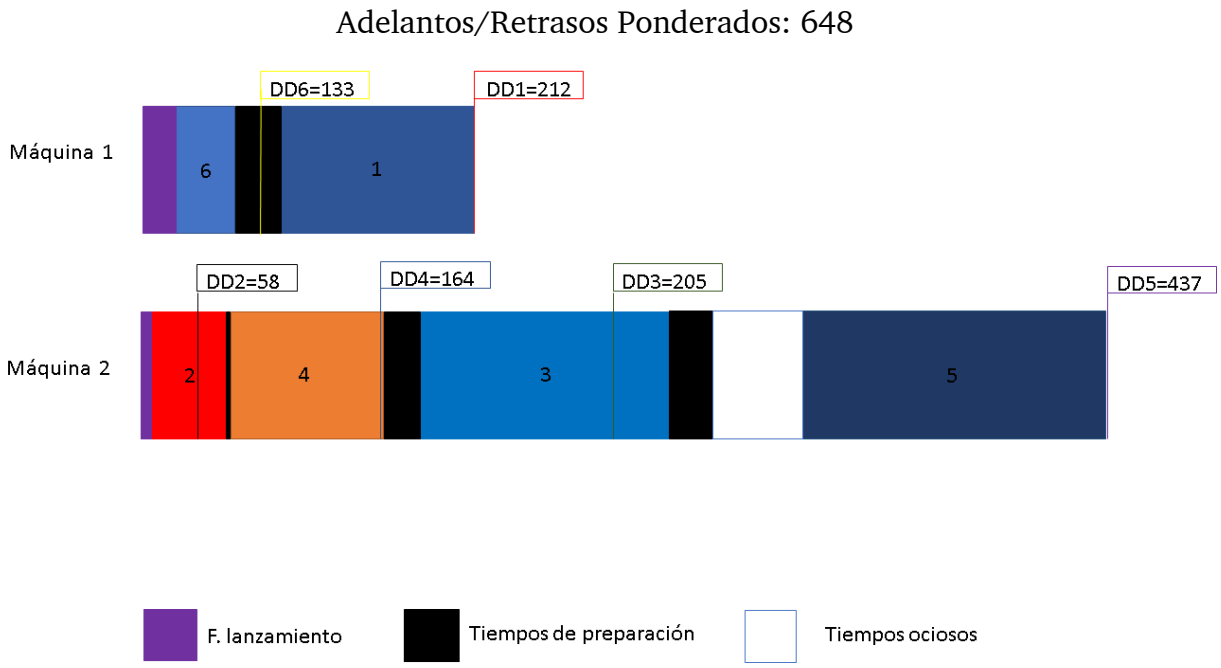

Figura 4. 6: Ejemplo de 6 trabajos y 2 máquinas con tiempos ociosos.

En la tabla 4.1 se puede observar las fechas de lanzamiento r_j , los tiempos de procesamiento p_{ij}, fechas de entrega d_j , fechas límite $\bar{d}_j$, tiempos de finalización de los trabajos antes de los tiempos de ociosos C_j y después de los tiempos ociosos C'_j, los adelantos/retrasos ponderados sin de tiempos ociosos y los adelantos/retrasos ponderados con los tiempos ociosos insertados.

Tabla 4. 1: Ejemplo 6 trabajos y 2 máquinas antes y después de inserción tiempo ocioso.

Trabajos	Máquina	rj	pij	Cj	C'j	Sijk	DDj	$\bar{d}_j$	A/R pond. sin ocio	A/R pond. con ocio
2	2	9	80	89	89	0	58	138	93	93
4	2	9	83	174	174	2	164	244	60	60
3	2	15	58	272	272	40	205	285	402	402
5	2	34	16	338	437	50	437	537	99	**0**
6	1	37	65	102	102	0	133	213	93	93
1	1	14	60	212	212	50	212	292	0	0
								Total A/R	747	648

El trabajo que se ha completado a tiempo está en negrita, para el 5 trabajo su inicio ha sido retrasado hasta el tiempo necesario para que se pueda completar en su fecha de entrega. Se puede observar una reducción en la función objetivo luego de insertar los tiempos ociosos en los trabajos.

4.3 Implementación y código

El algoritmo ha sido escrito en lenguaje C# utilizando el entorno de desarrollo Visual Studio 2017. Se ha utilizado la programación orientada a objetos. Es de destacar que todos estos conocimientos se adquirieron durante la elaboración de este trabajo de Fin de Máster. El código se ha distribuido en 5 clases: Trabajo, Maquina, Individuo, Simulador y Program. Se ira describiendo cada clase y todo el código correspondiente a estas se encuentra en el Apéndice A. En la clase Program es donde se ejecuta el algoritmo indicándole generaciones, población, progenitores, probabilidad de cruce, probabilidad de mutación y elitismo, también indicamos el nombre del fichero .txt donde están los datos de las instancias utilizadas, estarán contenidas en la carpeta inputs.

```
{
    class Program
    {
        static void Main(string[] args)
        {
            Simulador simulacion = new Simulador(60, 60, 3, 50, 40, 2);
            simulacion.cargarDatos("datos_6_2_49.txt");
            simulacion.ejecutar();
        }
    }
}
```

Listado 4.3: Código en C# para la clase Program.cs

La clase simulador contiene el algoritmo genético utilizando el método ejecutar() de acuerdo a los parámetros que se le indiquen en la clase Program se genera la población inicial a través del método generarPoblación() se crea una lista de individuos que son soluciones aleatorias donde se asignan los trabajos a la máquina donde menor tiempo de procesamiento tengan, si esta secuencia es infactible entonces sigue generando secuencias hasta que sean válidas siempre que se genera un individuo es comprobado mediante el método validar (), una vez se tiene la población inicial entonces se empiezan a crear las generaciones a través del bucle utilizando los operadores de selección de torneos, se determina si los padres elegidos se cruzan o no de acuerdo a la probabilidad (p_c) igual para la mutación de inserción de acuerdo a la probabilidad (p_m) antes de insertar los nuevos individuos a la nueva población son evaluados para determinar si son mejores que sus padres y sé comprueba que no excedan las fechas límite de los trabajos mediante el método validar(). Una vez se realizan las generaciones indicadas se elige el mejor individuo, aplicándole la inserción de tiempos ociosos mediante el método reajustar() y se presenta la solución con y sin tiempos ociosos en el fichero .txt generado por el método exportarInforme().

También está contenido el operador de cruce mediante dos métodos que son seleccionarCruce() donde se elige de acuerdo al número de progenitores establecidos cuales se van a cruzar y efectuarCruce() donde se realiza el cruce con los individuos elegidos para crear la descendencia evitando que se repitan los trabajos y operador de selección donde se eligen los progenitores con mejor valor de la función objetivo. Los métodos generarPoblacion(), generar individuo() están definidos en esta clase así como ordenarPoblacion() para ordenar los individuos donde los mejores estarán en las primeras posiciones de la población y getMejorIndividuo().

En la clase Individuo.cs se crean y manejan nuestros individuos que son las listas de máquinas, contiene el método para realizar la mutación de inserción mediante mutar() aquí se elige la máquina de manera aleatoria y un trabajo aleatorio dentro de la máquina intercambia su posición con otro trabajo de la máquina. También aquí se utiliza validar() para la comprobación que las secuencias de trabajos en los individuos sean válidas.

La validación como se ha mencionado anteriormente se realiza mediante el método validar() es planteado en la clase Trabajo.cs, está presente en la clase Maquina.cs para la secuencias de trabajos en las máquinas así como también en la clase Individuo.cs ya que todos los individuos que sean generados deberán ser secuencias factibles donde los Cj de los trabajos deben ser menor o igual que las fechas límite. Se establece mediante una variable bool valido = true cuando Cj es menor o igual que la fecha límite.

```
public bool validar() //Validación de las secuencias de trabajos en las máquinas.
        {
            bool valido = true;

            int trabajo_actual = 0;
            int trabajo_anterior = -1;
            int total = 0;
            if (trabajos.Count > 0)
            {
                total = trabajos[trabajo_actual].getDiasInicio();
                valido = trabajos[trabajo_actual].validar(total);
            }

            while (trabajos.Count > 0 && trabajo_actual < trabajos.Count && valido)
            {
                total += trabajos[trabajo_actual].getTiempoProcesamiento();
                trabajo_anterior = trabajo_actual;
                trabajo_actual++;
                if (trabajo_actual < trabajos.Count)
                {
                    total += getTiempoPreparacion(trabajos[trabajo_anterior],
trabajos[trabajo_actual]);
                }
                if (trabajo_actual < trabajos.Count)
                {
                    valido = trabajos[trabajo_actual].validar(total);
                }
            }
              return valido;
```

Listado 4.4: Método para validar las listas de máquinas en la clase Individuo.cs

```
public bool validar(int diasAcumulados) //valida que los trabajos no exceden la fecha
límite. diasAcumulados representa cuando inician los trabajos. diasAcumulados más
tiempos de procesamiento es igual al Cj. Cj <= deadlines.
        {
            bool valido = true;
            valido = ((diasAcumulados + tiempos_procesamiento[maquina_asociada-1]) <=
diasLimite);
            return valido;
        }
```

Listado 4.5: Método para validar los trabajos en la clase trabajo.cs

```
public bool validar()
        {
            bool valido = true;
            for(int i = 0; valido && i < maquinas.Count; i++)
            {
                if(maquinas[i].getTrabajos().Count > 0)
                {
                    valido = maquinas[i].validar();
                }
                else
                {
                    valido = false;
                }
```

Listado 4.6: Método para validar las secuencias de trabajos en la clase Maquina.cs

La clase Trabajo.cs es donde se definen los métodos para el manejo de los inputs asociados a los trabajos, máquina asociada al trabajo, el método asignación inicial define la asignación de los trabajos de acuerdo al menor tiempo de procesamiento, se calcula el valor de la función objetivo con el método getRatio. La clase Maquina.cs contiene los métodos para el manejo de las listas de trabajos en las máquinas, determinar el tiempo de preparación de acuerdo a los trabajos secuenciados, validar las secuencias de los trabajos, calcular el valor de la función objetivo por máquina, generar el informe final donde se determina cuando empiezan y terminan los trabajos en las máquinas y se define el método reajustar() para insertar los tiempos ociosos.

Capítulo 5. Validación

5.1 Medidas del rendimiento

En este capítulo se describen los procedimientos utilizados para generar las instancias que serán utilizadas para la validación de los métodos, luego con las instancias pequeñas evaluamos el modelo MILP utilizando el Solver Lingo 17.0 y con las instancias grandes realizaremos la evaluación del algoritmo genético propuesto y sus variaciones. Una vez obtenidos los resultados de las dos versiones del algoritmo mediante un análisis de la varianza se determinará si existen diferencias estadísticas.

La medida del rendimiento más utilizada en la literatura para comparar métodos de programación es la desviación porcentual relativa (Relative Percentage Deviation) RPD, dada por la fórmula:

$$RPD = \frac{Solución_{met} - Mejor_{sol}}{Mejor_{sol}} * 100$$

Donde $Solución_{met}$ es la solución obtenida por el método para cada instancia y $Mejor_{sol}$ es la mejor solución obtenida para esa instancia entre las versiones del algoritmo.

Sin embargo, cuando se está minimizando la sumatoria de adelantos/retrasos ponderados, la mejor solución para una instancia en el denominador de la fórmula del RPD puede ser 0, por lo tanto, óptima ya que todos los trabajos terminarían exactamente en su fecha de entrega y no habría penalizaciones. Por lo que una medida del rendimiento más adecuada sería el índice porcentual relativo (Relative Percentage Index) RPI, que se define:

$$RPI = \frac{Solución_{met} - Mejor_{sol}}{Peor_{sol} - Mejor_{sol}} * 100$$

Donde $Peor_{sol}$ es la peor solución obtenida entre todas las variaciones del algoritmo para una instancia determinada. El RPI proporciona un numero entre 0 y 100 donde los valores más cercanos a 0 indican un buen rendimiento de la variante del algoritmo.

5.1.2 Instancias

Para la evaluación de los métodos se generan un conjunto de instancias con tiempos de procesamiento p_{ij} distribuidos de manera uniforme en el rango [1,99] de forma aleatoria, como es usual en los trabajos del área. Se crean 3 conjuntos de instancias. Dos para la evaluación del modelo MILP y del algoritmo genético y un tercer conjunto que será diferente de los dos primeros para la calibración del algoritmo genético. Esto para evitar cualquier sesgo o sobre-ajuste a la hora de realizar la calibración del algoritmo. Las instancias para evaluación de los métodos se dividen en dos. Una instancia pequeña donde se prueban todas las combinaciones de n trabajos = [6,8,10,12] y m máquinas = [2,3,4,5]. Las instancias grandes serán de n trabajos = [50,100,150] y m máquinas = [10,15,20].

Para cada combinación de n x m se consideran fechas de lanzamiento con una distribución uniforme aleatoria en el rango de [1,15], dos distribuciones uniformes aleatorias para los tiempos de preparación. Una distribución aleatoria uniforme en el rango de [1,49] y otra en el rango de [1,124]. Con estas dos distribuciones se puede representar tiempos de preparación pequeños y grandes entre trabajos.

Para generar las fechas de entrega se ha tomado como límite inferior la media de los tiempos de procesamiento de cada trabajo en todas las máquinas, es decir se suman los P_{ij} de cada trabajo y se dividen por el número de máquinas. Para las fechas límites se generan a partir de las fechas de entrega siendo éstas un 50% más que las fechas de entrega. Los pesos para los adelantos y retrasos son generados a partir de una distribución uniforme aleatoria en el rango de [1,10]. Para las combinaciones de n, m y S_{ijk} se obtendrán 4 * 4 * 2 = 32 instancias pequeñas y 3*3*2*5= 90 instancias grandes para evaluación de las variaciones del algoritmo. Y un último conjunto de instancias de calibración para los 8 tratamientos considerados para un total de 512.

La estructura del fichero de datos es la siguiente:

```
total_maquinas
total_trabajos
id_T1  TiempoProcesamiento_M1  TiempoProcesamiento_Mn
PenalizaciónRetraso  PenalizaciónAdelanto  FechaEntrega
FechaLanzamiento  FechaLimite
 id_T2  TiempoProcesamiento_M1  TiempoProcesamiento_Mn
PenalizaciónRetraso  PenalizaciónAdelanto  FechaEntrega
FechaLanzamiento  FechaLimite
id_Tn  TiempoProcesamiento_M1  TiempoProcesamiento_Mn
PenalizaciónRetraso  PenalizaciónAdelanto  FechaEntrega
FechaLanzamiento  FechaLimite
  id_M1
  TPrep_1:1  TPrep_1:2  TPrep_1:n
  TPrep_2:1  TPrep_2:2  TPrep_2:n
   TPrep_n:1  TPrep_n:2  TPrep_n:n
  id_Mn
  TPrep_1:1  TPrep_1:2  TPrep_1:n
  TPrep_2:1  TPrep_2:2  TPrep_2:n
  TPrep_n:1  TPrep_n:2  TPrep_n:n
```

Figura 5. 1: Ejemplo de instancia para 6 trabajos y 2 máquinas.

```
2
6
1       14      1       3       10      269     63      295
2       22      30      3       3       176     14      193
3       61      75      5       9       241     15      265
4       43      19      7       1       192     60      211
5       85      33      4       9       328     55      360
6       91      82      6       3       187     82      205
0       33      48      36      34      27
17      0       34      28      18      16
20      5       0       15      39      27
27      2       34      0       25      16
15      28      11      26      0       45
22      32      14      29      36      0
0       11      25      33      16      30
27      0       22      7       26      39
35      11      0       22      48      37
40      21      1       0       46      27
23      23      27      41      0       5
3       32      9       1       35      0
```

Listado 4.7: Estructura fichero de datos para input del GA.

5.2 Evaluación del modelo matemático

Para evaluar el modelo matemático MILP se utilizan las 32 instancias pequeñas, para cada una de estas instancias se ha generado un archivo Lingo Model, el tamaño más grande de estas instancias es 12 trabajos x 5 máquinas. El software utilizado para la evaluación es Lingo 17.0. Todas las evaluaciones han sido corridas en una computadora con un procesador Intel(r) i5-6198DU a una frecuencia de 2.30Ghz con 12GB de memoria RAM en el sistema operativo Windows 10 Home x64. El procesador cuenta con dos núcleos. Los parámetros utilizados en lingo17.0 para todas las instancias son: el tiempo de parada ha sido establecido a 1800 segundos y un solo subproceso o hilo de ejecución (thread).

Para cada resultado se guarda una variable categórica con dos valores: 0 y 1. El 0 indica que una solución óptima ha sido encontrada en el tiempo establecido con el valor de wET (adelantos y retrasos ponderados) como resultado y el 1 indica que para el tiempo establecido de 1800s, una solución entera factible fue encontrada y reportada, pero no es comprobada que esta solución es óptima.

Se guarda un valor llamado GAP en estos casos. El gap es la diferencia entre la solución factible y la cota inferior(LB). La fórmula con la que se calculará el Gap, que estará expresado en tanto por cien, es:

$$GAP\ (\%) = \frac{wET - LB}{wET} * 100$$

En la tabla 5.1 se presenta la media de los gaps en porcentaje y la media de los tiempos necesitados para resolver las instancias.

Se ha encontrado la solución óptima para las 32 instancias, donde los tiempos y los valores de la función objetivo son mayores en las instancias con menor número de máquinas, la explicación para esto es que se debe asignar y secuenciar en cada máquina muchos trabajos por lo que los tiempos de preparación entre los trabajos juegan un papel importante, ya que dependiendo de estos se requiere mayor tiempo computacional para encontrar el valor óptimo.

En la tabla 5.2 se presentan los valores de gap y segundos para las instancias de acuerdo a la distribución de los tiempos de preparación utilizados [1,49] y [1,124], donde las instancias con setups [1,124] muestran mayor tiempo de cálculo para encontrar la solución. Una vez resueltas las instancias determinadas para probar el modelo se ha intentado resolver instancias de mayor tamaño con el modelo se ha llegado al tope de 14 trabajos con 2 y 3 máquinas. En la tabla 5.3 se muestra que a partir de 13 trabajos ya no se consigue la solución óptima.

Tabla 5. 1 : Resultados del modelo MILP evaluado en Lingo 17.0 para las instancias pequeñas.

Modelo MIP			
Tiempo 1,800s			
1 Thread			
n	m	GAP %	Tiempo(seg)
6	2	0	0.31
6	3	0	0.22
6	4	0	0.19
6	5	0	0.18
8	2	0	30
8	3	0	0.365
8	4	0	0.28
8	5	0	0.595
10	2	0	202.61
10	3	0	2.29
10	4	0	31.50
10	5	0	5.67
12	2	0	688.78
12	3	0	183.89
12	4	0	45.78
12	5	0	10.50

Tabla 5. 2: Resultados del modelo MILP evaluado en Lingo 17.0 de acuerdo a las distribuciones de setup.

Modelo MIP							
Tiempo 1,800				Tiempo 1,800			
1 Thread				1 Thread			
Setups [1,49]				Setups [1,124]			
n	m	GAP %	Tiempo	n	m	GAP %	Tiempo
6	2	0	0.23	6	2	0	0.39
6	3	0	0.19	6	3	0	0.25
6	4	0	0.16	6	4	0	0.22
6	5	0	0.15	6	5	0	0.20
8	2	0	56	8	2	0	62
8	3	0	0.26	8	3	0	0.47
8	4	0	0.22	8	4	0	0.34
8	5	0	0.19	8	5	0	1
10	2	0	162.95	10	2	0	242.26
10	3	0	1.57	10	3	0	3
10	4	0	22.00	10	4	0	57.00
10	5	0	3.80	10	5	0	7.54
12	2	0	1259.00	12	2	0	1363.13
12	3	0	355.87	12	3	0	809.00
12	4	0	14.00	12	4	0	46.00
12	5	0	3.00	12	5	0	6.00

Tabla 5. 3: Resultados del modelo MILP para las instancias de 13 trabajos hasta 5 máquinas.

n	m	GAP %	Tiempo
13	2	55.48	1800
13	3	50.77	1800
13	4	25.64	1800
13	5	6.70	1800

5.3 Evaluación del Algoritmo Genético

5.3.1 Calibración

Ahora procederemos a realizar una pequeña calibración del algoritmo a través de un diseño de experimentos (DOE) de forma exploratoria en el programa Statgraphics Centurion XVII. Para obtener los valores de los parámetros: tamaño de la población(tPob), Probabilidad de cruce (pC) y probabilidad de mutación (pM). Realizaremos pruebas en el algoritmo con las instancias grandes de (50,100,150) trabajos con (10,15,20) máquinas y con las instancias pequeñas de (6,8,10,12) trabajos y (2,3,4,5) máquinas. Los valores utilizados para los parámetros son: tamaño de la población(tPob), Probabilidad de cruce (pC) y probabilidad de mutación (pM). El criterio de parada ha sido establecido en 60 segundos. En la tabla 5.4 se pueden observar los valores de la mejor combinación en negrita.

Tabla 5. 4: Valores probados para los parámetros en el experimento de calibración (mejor combinación en negrita)

Parámetro	Algoritmo Genético
Tamaño de la población (tPob)	**60**;80
Probabilidad de cruce(pC)	0.4;**0.5**
Probabilidad de mutación(pM)	0.4;**0.5**

En la tabla 5.5 se observa el Análisis de la varianza (ANOVA) con interacciones de segundo orden realizado en la calibración todos los factores simples resultan significativos, pero solo las interacciones de AB, BC y BE resultan significativas para un ($\alpha = 0.05$). Si observamos los gráficos de media en las figuras 5.2, 5.3 y 5.4 para los factores Tpob, Pc y Pm con respecto al RPI con los HSD intervalos de Tukey para un ($\alpha = 0.05$) y en las tablas 5.6, 5.7 y 5.8 se observan las pruebas de Múltiples Rangos para RPI para ambos niveles de los tres factores, así como sus intervalos de confianza para un 95%. Los intervalos no se encuentran solapados por lo que se podría concluir que las diferencias son estadísticamente significativas. Existen 3 hipótesis del Anova que deben ser comprobadas: la homocedasticidad, normalidad e independencia de los residuos. En la figura 5.5 se puede observar que existe una desviación de la

normalidad por parte de los residuos, por lo que hemos decidido apoyarnos tambien en otro test no paramétrico para evaluar las diferencias entre ambas los distintos niveles de los factores del GA esta pruebas es el Rank-based Friedman. En este test ocurre una transformacion de la data donde se le asignan rangos a cada valor original y se evalua si existen diferencias entre los distintos niveles de los factores. El test basado en rangos de Friedman sera llevado a cabo en el programa SPSS Statistics 23.

Tabla 5. 5 : Anova para RPI con interacciones de segundo grado.

	Suma de Cuadrados	Gl	Cuadrado Medio	Razón-F	Valor-P (< 0.05)
EFECTOS PRINCIPALES					
A:Tpob	4079.51	1	4079.51	9.73	0.0019
B:Pc	6506.42	1	6506.42	15.51	0.0001
C:Pm	5479.02	1	5479.02	13.06	0.0003
D:n	25407.6	3	8469.2	20.19	0
E:m	14566.6	3	4855.55	11.58	0
INTERACCIONES					
AB	4599.87	1	4599.87	10.97	0.001
AC	105.05	1	105.05	0.25	0.617
AD	450.733	3	150.244	0.36	0.7832
AE	2066.57	3	688.856	1.64	0.1787
BC	2924.14	1	2924.14	6.97	0.0086
BD	642.517	3	214.172	0.51	0.6751
BE	3533.75	3	1177.92	2.81	0.0391
CD	104.374	3	34.7912	0.08	0.9693
CE	427.104	3	142.368	0.34	0.7968
DE	6480.02	9	720.002	1.72	0.0826
RESIDUOS	197950	472	419.386		
TOTAL (CORREGIDO)	275324	511			

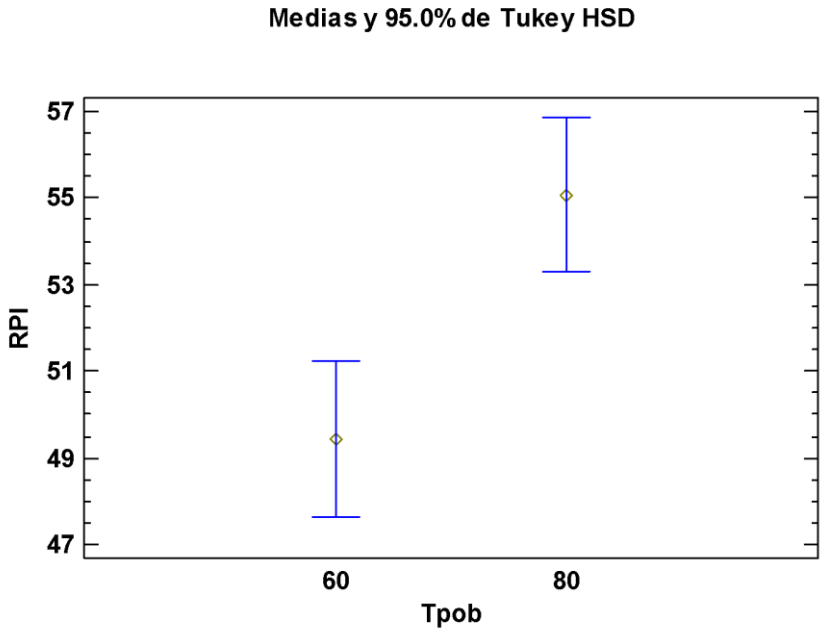

Figura 5.3 : Gráfico de intervalos HSD Tukey para RPI con respecto al factor tamaño de la población(Tpob).

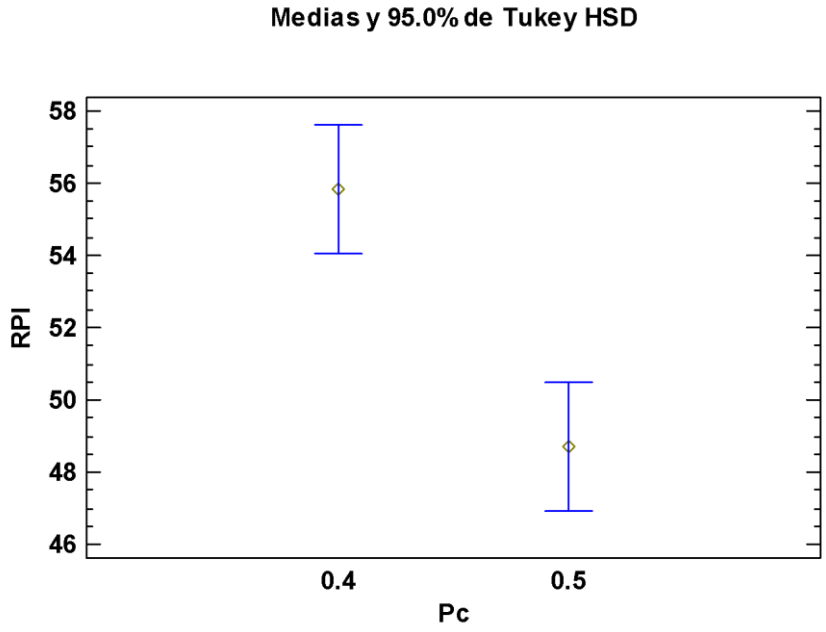

Figura 5.4: Gráfico de intervalos HSD Tukey para RPI con respecto al factor probabilidad de cruce(Pc).

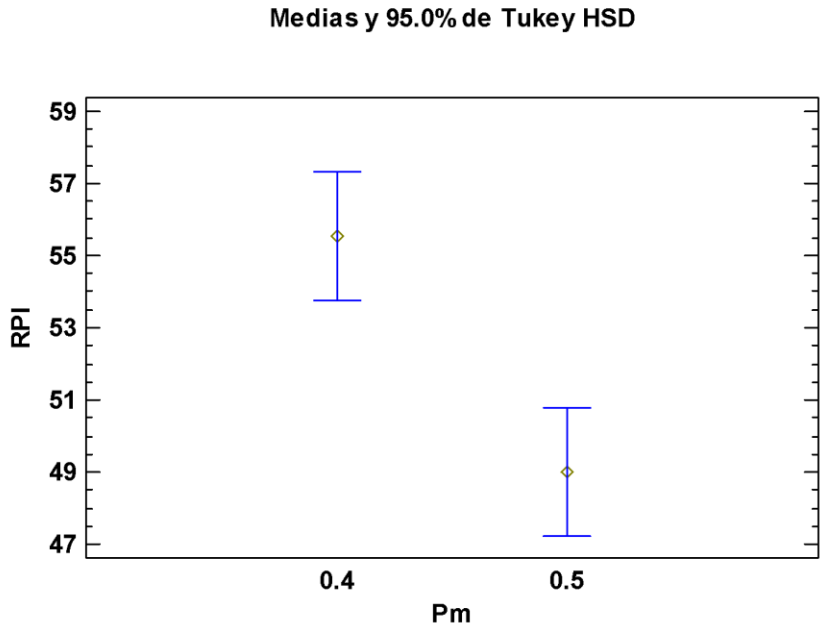

Figura 5.5: Gráfico de intervalos HSD Tukey para RPI con respecto al factor probabilidad de mutación(Pm).

Tabla 5. 6: Pruebas de Múltiple Rangos para RPI por Tpob

Método: 95.0 porcentaje Tukey HSD

Tpob	*Casos*	*Media LS*	*Sigma LS*	*Grupos Homogéneos*
60	256	49.4346	1.27993	X
80	256	55.0801	1.27993	X

Contraste	*Sig.*	*Diferencia*	*+/- Límites*
60 - 80	*	-5.64546	3.55685

* indica una diferencia significativa.

Tabla 5.7: Pruebas de Múltiple Rangos para RPI por Pc

Método: 95.0 porcentaje Tukey HSD

Pc	*Casos*	*Media LS*	*Sigma LS*	*Grupos Homogéneos*
0.5	256	48.6926	1.27993	X
0.4	256	55.8222	1.27993	X

Contraste	*Sig.*	*Diferencia*	*+/- Límites*
0.4 - 0.5	*	7.12962	3.55685

* indica una diferencia significativa.

Tabla 5.8: Pruebas de Múltiple Rangos para RPI por Pm

Método: 95.0 porcentaje Tukey HSD

Pm	*Casos*	*Media LS*	*Sigma LS*	*Grupos Homogéneos*
0.5	256	48.9861	1.27993	X
0.4	256	55.5286	1.27993	X

Contraste	*Sig.*	*Diferencia*	*+/- Límites*
0.4 - 0.5	*	6.54254	3.55685

* indica una diferencia significativa.

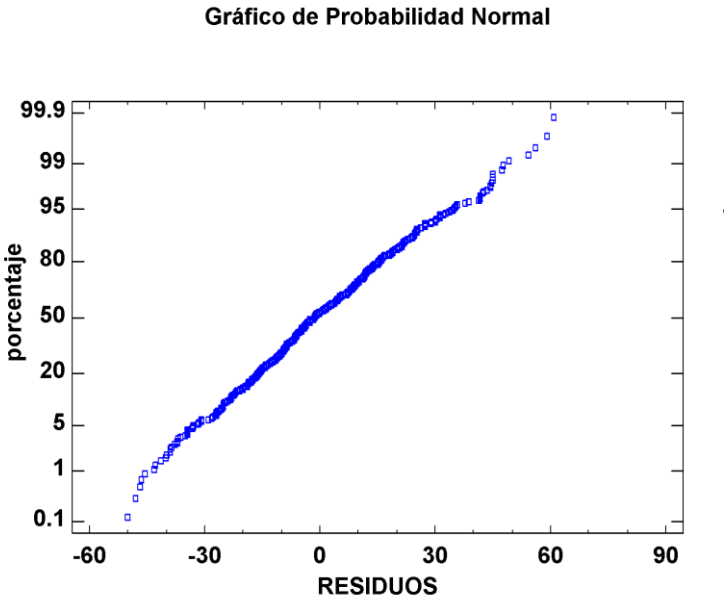

Figura 5. 8 : Gráfico de probabilidad normal para los residuos del RPI en ANOVA de segundo orden.

Se realizará el test basado en rangos de Friedman para los tres factores (Tpob, Pc y Pm) para sus dos niveles con respecto al RPI. En la tabla 5.9 se observan los quartiles para ambos tamaños de población, en 5.10 se puede observar los diferentes rangos de medias de los dos niveles y en 5.8 los resultados indican que si existen diferencias estadísticamente significativas entre los tamaños de la población.

Tabla 5.9: Estadísticos descriptivos del test de Friedman para TPob.

Descriptive Statistics

	N	Percentiles		
		25th	50th (Median)	75th
TPob60	256	32.6735	49.0018	64.9797
TPob80	256	38.8768	56.3889	70.4581

Tabla 5.10: Test de Rangos de Friedman para TPob.

Ranks

	Mean Rank
TPob60	1.42
TPob80	1.58

Tabla 5.11 : Estadístico del Test de Friedman para TPob.

Test Statistics[a]

N	256
Chi-Square	6.644
df	1
Asymp. Sig.	.010

En la tabla 5.12 se observan los quartiles para ambas probabilidades de cruce, en 5.13 se puede observar los diferentes rangos de medias de las prob y en 5.8 los resultados indican que si existen diferencias estadísticamente significativas entre las probabilidades de cruce.

Tabla 5.12: Estadísticos descriptivos del test de Friedman para Pc.

Descriptive Statistics

	N	Percentiles		
		25th	50th (Median)	75th
Pc0.4	256	39.0758	57.7778	71.9250
Pc0.5	256	32.6735	50.1714	63.0806

Tabla 5.13 : Test de Rangos de Friedman para Pc.

Ranks

	Mean Rank
Pc0.4	1.60
Pc0.5	1.40

Tabla 5.14 : Estadístico del Test de Friedman para Pc.

Test Statistics[a]

N	256
Chi-Square	9.490
df	1
Asymp. Sig.	.002

En la tabla 5.15 se observan los quartiles para ambas probabilidades de mutación, en 5.16 se puede observar los diferentes rangos de medias de las prob y en 5.17 los resultados indican que si existen diferencias estadísticamente significativas entre las probabilidades de mutación.

Tabla 5.15 : Estadísticos descriptivos del test de Friedman para Pm.

Descriptive Statistics

	N	Percentiles		
		25th	50th (Median)	75th
Pm0.4	256	39.4226	57.0246	71.6212
Pm0.5	256	30.9041	49.2059	64.2036

Tabla 5.16 : Test de Rangos de Friedman para Pm.

Ranks

	Mean Rank
Pm0.4	1.59
Pm0.5	1.41

Tabla 5.17 : Estadístico del Test de Friedman para Pm.

Test Statistics[a]

N	256
Chi-Square	8.331
df	1
Asymp. Sig.	.004

Dado los resultados obtenidos en las diferentes pruebas podemos apoyar el resultado obtenido en el ANOVA de que, si existen diferencias significativas entre los niveles de los factores Tpob, Pc y Pm. Y que la mejor combinación es:

Tabla 5.18 : Mejor combinación de los factores Tpob, Pc y Pm obtenidos en Statgraphics XVII.

Factor	Establecimiento
Tpob	60
Pc	0.5
Pm	0.5

5.3.2. Evaluación de las instancias pequeñas.

En la tabla 5.19 están contenidos los resultados de la evaluación. La versión del algoritmo con tiempos de ociosos insertados mejora por un 39% los resultados obtenidos en la versión sin tiempos de ocioso, si se observa la versión del algoritmo con tiempos de ocioso insertados y calibrado los resultados son 245% y 148% mejores que la versión GA sin t.ociosos y GA con t.ociosos, respectivamente. Los parámetros utilizados para la versión sin calibrar son Tpob=60, Pc=0.4 y Pm= 0.5. La mejor versión del algoritmo esta 21.6% lejos del 0 dentro del índice porcentual relativo. Cabe recordar que el tiempo tomado por el algoritmo es mucho menor que el tiempo necesitado por el modelo matemático.

Tabla 5.19: Media del índice de porcentual relativo (RPI) para las versiones del algoritmo en instancias pequeñas.

Instancias		GA sin t. ociosos	GA con t. ociosos	GA con t. ociosos Cal.
n	m	RPI	RPI	RPI
6	2	46	30	18
6	3	84	47	40
6	4	89	74	10
6	5	88	50	1.6
8	2	71	32	19
8	3	78	68	23
8	4	76	50	34
8	5	90	57	15
10	2	71	34	27
10	3	62	52	22
10	4	77	50	23
10	5	65	50	25
12	2	53	50	13
12	3	83	82	30
12	4	83	79	28
12	5	79	54	18
Media		74.69	53.69	21.66

Para obtener resultados más concluyentes, un análisis estadístico mediante un análisis de la varianza es llevado a cabo. Podemos ver en la figura 5.6 el grafico de medias para el RPI, con intervalos HSD Tukey para un (α=0.05), estos no se solapan por lo que se puede concluir que si son estadísticamente diferentes. Validando la normalidad de los residuos mediante el grafico de papel probabilístico normal en la figura 5.7 no se observa una fuerte desviación de la normalidad en los residuos y en la prueba de Levene`s dado que el p valor es 0.09 mayor que el (α=0.05) no existe una diferencia estadísticamente significativa entre las varianzas, con un nivel de confianza del 95%. Por lo que también se cumple la hipótesis de homocedasticidad.

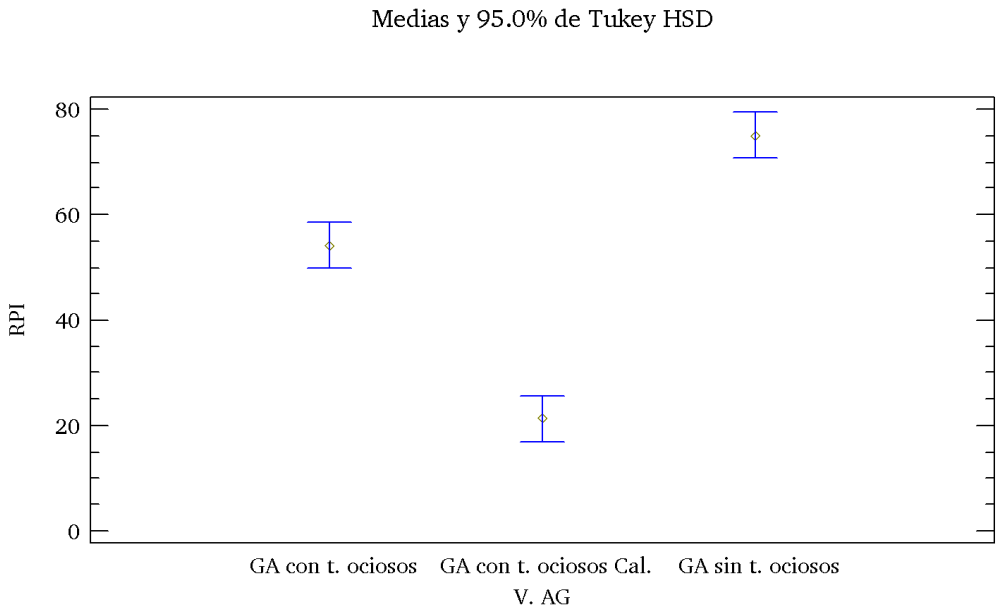

Figura 5. 11: Gráfico de medias e intervalos Tukey HSD para un nivel de confianza del 95% para las versiones del algoritmo genético propuesto (instancias pequeñas).

Tabla 5. 20 : Prueba de Levene´s para verificación de varianza

	Prueba	*Valor-P*
Levene's	2.38651	0.0947

Comparación	*Sigma1*	*Sigma2*	*F-Ratio*	*P-Valor*
GA con t. ociosos / GA con t. ociosos Cal.	24.1028	18.4134	1.71342	0.0643
GA con t. ociosos / GA sin t. ociosos	24.1028	20.0061	1.45147	0.1420
GA con t. ociosos Cal. / GA sin t. ociosos	18.4134	20.0061	0.84712	0.5122

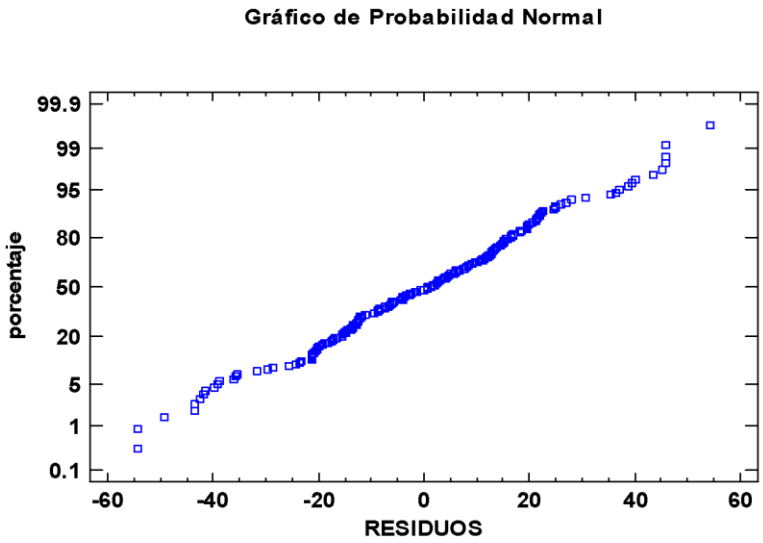

Figura 5. 12 : Gráfico de Papel probabilístico normal para los residuos del RPI en el ANOVA para las versiones del GA en instancias pequeñas.

5.3.3 Evaluación de las instancias grandes

En la tabla 5.21 podemos observar los resultados obtenidos para las instancias grandes. Donde podemos resaltar el buen desempeño de la versión del algoritmo con tiempos ociosos calibrado dada que su media en RPI es de 6.24, recordando que mientras más cerca el valor a 0 es mejor.

La mejora del RPI del GA. con t. ociosos calibrado son de 1378% y 41% con respecto la versión de AG sin t. ociosos y AG con t. ocioso. La versión de AG con t. ociosos mejora en 945% la versión sin t. ociosos. Aquí se ve reflejado lo antes expuesto de la importancia de considerar la inserción de tiempos ociosos cuando se está trabajando con adelantos/retrasos como objetivo.

Tabla 5. 21: Medías del índice porcentual relativo(RPI) para el algoritmo propuesto en las instancias grandes.

Instancias		AG sin t. ociosos	AG con t. ociosos	AG con t. ociosos Cal.
n	m	RPI	RPI	RPI
50	10	71.68	13.32	7.49
50	15	91.89	7.55	7.55
50	20	95.62	10.09	3.74
100	10	94.47	8.46	7.37
100	15	95.63	8.86	4.98
100	20	96.93	4.14	3.54
150	10	95.05	10.71	10.64
150	15	93.79	10.33	7.03
150	20	95.56	6.05	3.87
Media		92.29	8.83	6.24

Una vez terminadas las pruebas se realizó un análisis de la varianza (ANOVA) para determinar si las diferencias observadas son estadísticamente significativas, este análisis será llevado a cabo en el programa Statgraphics Centurion XVII.

En la figura 5.8 se puede observar el grafico de medias para las versiones del algoritmo con respecto al RPI con los HSD intervalos de Tukey para un ($\alpha = 0.05$). En este caso los intervalos no se encuentran solapados por lo que se podría concluir que las diferencias son estadísticamente significativas para la versión calibrada con respecto a la versión sin tiempos ociosos. Por lo que el algoritmo con tiempos de ociosos insertados y calibrado tiene un mejor desempeño que la versión sin tiempos ociosos.

Es necesario comprobar las 3 hipótesis del Anova que son la homocedasticidad, normalidad e independencia de los residuos. En la figura 5.9 podemos observar que existe una desviación de la normalidad por parte de los residuos, por lo que hemos decidido apoyarnos tambien en otros test no paramétricos para evaluar las diferencias entre ambas versiones del algoritmo estas pruebas son : Rank-based Friedman y la prueba de kruskal – wallis. En estos test ocurre una transformacion de la data donde se le asignan rangos a cada valor original y se evalua si existen diferencias entre las dos versiones del algoritmo.

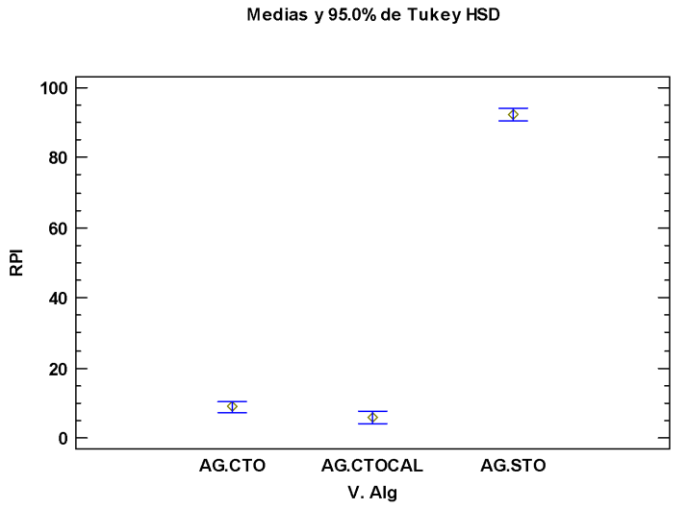

Figura 5. 15: Gráfico de medias e intervalos Tukey HSD para un nivel de confianza del 95% para las versiones del algoritmo genético propuesto (instancias pequeñas).

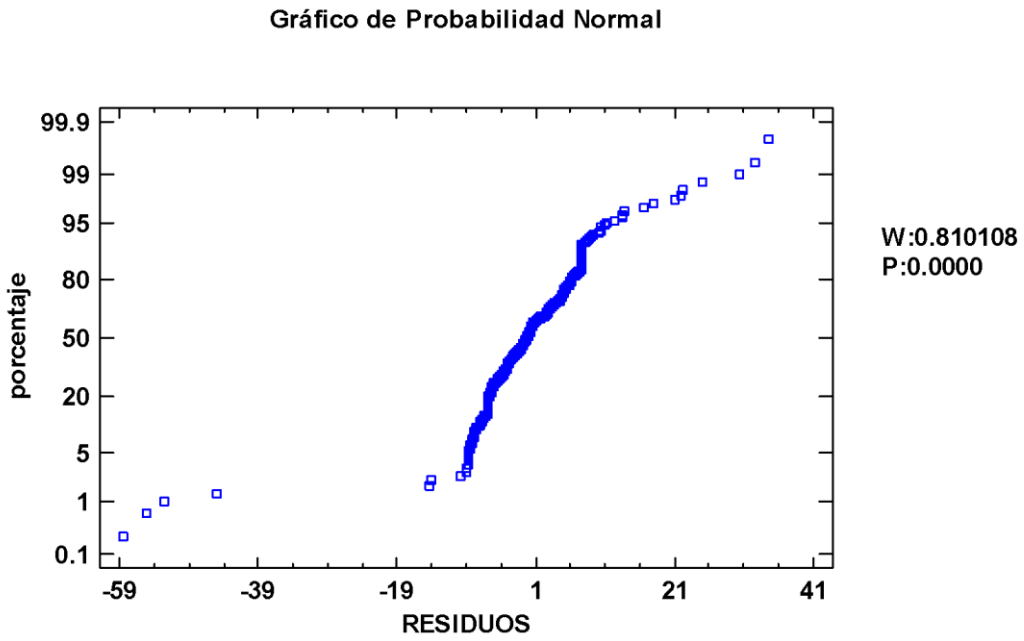

Figura 5. 9: Gráfico de Papel probabilístico normal para los residuos del RPI en el ANOVA para las versiones del GA en instancias grandes.

Los resultados de la prueba Kruskal – Wallis se observan en la tabla 5.22 donde se evalúa la hipótesis de que las medianas de RPI dentro de cada uno de los 3 niveles de V. Alg son iguales. Primero se combinan los datos de todos los niveles y se ordenan de menor a mayor. Luego se calcula el rango (rank) promedio para los datos de cada nivel. Puesto que el valor-P es menor que 0.05, existe una diferencia estadísticamente significativa entre las medianas con un nivel del 95.0% de confianza.

En el segundo contraste muestra comparaciones por pares entre los rangos promedio de los 3 grupos. Usando el procedimiento de Bonferroni la comparación es estadísticamente significativa para las versiones AG.sin t.ociosos - AG.con t.ociosos y AG.sin t.ociosos - AG.con t.ociosos Cal. para un intervalo de confianza del 95%. Para la última comparación AG.con t.ociosos - AG.con t.ociosos Cal. refleja que no existen diferencias estadísticas entre la versión con tiempos ociosos y la versión con tiempos ociosos calibrada.

Tabla 5. 22 : Prueba de Kruskal-Wallis para RPI por V. Alg

V. Alg	Tamaño Muestra	Rango Promedio
AG.sin t.ociosos	90	225.411
AG.con t.ociosos	90	98.6
AG.con t.ociosos Cal.	90	82.4889

Estadístico = 181.004 Valor-P = 0

intervalos de confianza del 95.0%

Contraste	Sig.	Diferencia	+/- Límites
AG.sin t.ociosos - AG.con t.ociosos	*	126.811	27.867
AG.sin t.ociosos - AG.con t.ociosos Cal.	*	142.922	27.867
AG.con t.ociosos - AG.con t.ociosos Cal.		16.1111	27.867

* indica una diferencia significativa.

Para llevar a cabo el test basado en rangos de Friedman utilizaremos el software IBM SPSS Statistics 23. En la tabla 5.23 se observan los quartiles para las dos versiones, en 5.24 se puede observar los diferentes rangos de medias de las versiones del algoritmo y en 5.25 los resultados indican que si existen diferencias estadísticamente significativas entre las versiones del algoritmo.

Tabla 5. 23 : Estadísticos descriptivos del test de Friedman para las versiones algoritmo genético.

Descriptive Statistics

	N	Percentiles		
		25th	50th (Median)	75th
GASTO	90	91.4987	95.4422	98.6854
GACTO	90	2.2057	5.8335	12.4847
GACTOCAL	90	1.1030	4.7759	8.5924

Tabla 5. 24 : Test de Rangos de Friedman para las versiones del algoritmo genético.

Ranks

	Mean Rank
GASTO	3.00
GACTO	1.64
GACTOCAL	1.36

Tabla 5. 25 : Estadístico del Test de Friedman para las versiones del algoritmo genético.

Test Statistics[a]

N	90
Chi-Square	138.756
df	2
Asymp. Sig.	.000

a. Friedman Test

Dado los resultados obtenidos en las diferentes pruebas podemos apoyar el resultado obtenido en el ANOVA de que si existen diferencias significativas para un intervalo de confianza de 95% entre las versiones del algoritmo y que el algoritmo con tiempos de ocioso insertado y calibrado tiene un mejor desempeño que la versión sin tiempos de ocioso insertados.

Capítulo 6. Conclusiones y trabajo futuro

6.1 Conclusiones

En este trabajo final de máster se ha tratado con el problema de máquinas paralelas no relacionadas tomando en cuenta fechas de lanzamiento (release dates), tiempos de preparación (setups times) y fechas límite (deadlines) con el objetivo de minimizar el total de adelantos/retrasos ponderados, primero se ha realizado una revisión bibliográfica de trabajos con atributos similares al nuestro para conocer el estado del arte y los diferentes métodos propuestos, constatando que la literatura es escasa cuando se combinan estas tres restricciones pero aun así existen una gran diversidad de métodos propuestos para la resolución del problema de programación en máquinas paralelas que proporcionan una buena referencia.

Se realizo una descripción del problema, se ha propuesto y resuelto mediante Lingo 17.0 un modelo matemático MILP para el problema, la adaptación del modelo en Lingo ha sido fácil debido a los conocimientos previos adquiridos en el transcurso de este Máster en las asignaturas de Programación Multicriterio y en Planificación y Programación de la Producción.

A pesar de encontrar la solución óptima para las instancias pequeñas propuestas se decidió resolver más instancias utilizando 13 trabajos asignados entre 2 y 5 máquinas, en estas instancias no se encuentra la solución óptima generando gaps alcanzando el tope en 14 trabajos y 2 máquinas, ya no fue capaz de resolver en el tiempo computacional evaluado. De acuerdo a los resultados obtenidos el tiempo requerido para encontrar las soluciones con tiempos de preparación [1;49] son ligeramente mejores que los tiempos obtenidos con la distribución [1;124].

Debido a esto, se propone un algoritmo genético adaptado al problema de programación en máquinas en paralelo no relacionadas, con una población inicial generada de soluciones aleatorias y una asignación inicial de los trabajos de acuerdo a su tiempo de procesamiento más corto (SPT) siempre verificando que no violen las restricciones impuestas.

Capítulo 6. Conclusiones y Trabajo futuro

La realización del Máster fue la principal motivación para adquirir los conocimientos adicionales de programación. Para la implementación de este método se adquirieron los conocimientos en programación en base a objetos en el lenguaje C#, lo que supuso todo un reto debido a la complejidad del desarrollo y la puesta en ejecución del mismo.

El algoritmo genético permite encontrar buenas soluciones al problema, aunque no óptimas, pero en tiempos computacionales razonables, presentando la ventaja de no estar limitado a tamaño de las instancias. Se presentan tres versiones del algoritmo una con la inserción de tiempos ociosos, sin tiempos ociosos y se realizó una pequeña calibración de este con fines exploratorios con lo aprendido en la asignatura Diseño de Experimentos obteniendo una última versión con tiempos ociosos insertados y calibrado. De acuerdo a los resultados obtenidos en las pruebas y análisis estadísticos para comprobar si existen diferencias entre versiones del algoritmo evaluando las instancias pequeñas y grandes se obtuvieron las siguientes conclusiones:

- En instancias pequeñas las tres versiones muestran diferentes desempeños, resultando mejor la versión con tiempos ociosos insertados y calibrada, se comprobó que existen diferencias estadísticamente significativas entre ellas para un nivel de confianza del 95%.

- En instancias grandes las tres versiones también muestran diferentes resultados, obteniendo el mejor desempeño con las versiones: con tiempos ociosos insertados y con tiempos ociosos insertados calibrada. Entre estas dos no se encuentran diferencias estadísticamente significativas, pero si con respecto a la versión sin tiempos ociosos insertados para un nivel de confianza del 95%.

Estos resultados ayudan a resaltar y validar lo antes explicado en el capítulo 4 que cuando se está trabajando con adelantos/retrasos lo beneficioso que es retrasar los trabajos lo suficientes para disminuir los adelantos logrando una reducción considerable en el valor de la función objetivo.

Con todo lo comentado anteriormente, a lo largo de este trabajo final de máster, se estudió un problema de máquinas en paralelo no relacionadas, aunque teórico, pero con diversas aplicaciones prácticas. Debido a las restricciones y función objetivo consideradas este se vuelve más complejo a la hora de formularlo y resolverlo. Lo que nos permite constatar la importancia de la programación de la producción para la toma de decisiones.

6.2 Trabajo futuro

El problema tratado en este trabajo final de máster puede ampliarse de distintas maneras aumentando su grado de complejidad y tiempos para resolución:

- Considerando otras restricciones a las máquinas como las paradas (brkdwns) y a los trabajos establecer relaciones de precedencias adaptando todavía más el problema a un entorno más realista.

- La combinación de otras metaheurísticas como GRASP para mejorar el algoritmo genético.

- Explorar otras maneras de inserción de tiempos ociosos.

Bibliografía

1. Allahverdi, A., Ng, C. T., Cheng, T. E., & Kovalyov, M. Y. (2008). A survey of scheduling problems with setup times or costs. European journal of operational research, 187(3), 985-1032.

2. Allahverdi, A. (2015). The third comprehensive survey on scheduling problems with setup times/costs. European Journal of Operational Research, 246(2), 345-378.

3. Allahverdi, A., Gupta, J. N., & Aldowaisan, T. (1999). A review of scheduling research involving setup considerations. Omega, 27(2), 219-239.

4. Allahverdi, A., & Soroush, H. M. (2008). The significance of reducing setup times/setup costs. European Journal of Operational Research, *187*(3), 978-984.

5. Avalos-Rosales, O., Angel-Bello, F., & Alvarez, A. (2015). Efficient metaheuristic algorithm and re-formulations for the unrelated parallel machine scheduling problem with sequence and machine-dependent setup times. The International Journal of Advanced Manufacturing Technology, 76(9-12), 1705-1718.

6. Avdeenko, T. V., & Mesentsev, Y. A. (2016). Efficient approaches to scheduling for unrelated parallel machines with release dates. IFAC (International Federation of Automatic Control), 49(12), 1743-1748.

7. Bajestani, M. A., & Tavakkoli-Moghaddam, R. (2009). A New Branch-and-Bound Algorithm for the Unrelated Parallel Machine Scheduling Problem with Sequence-Dependent Setup Times. IFAC (International Federation of Automatic Control) Proceedings Volumes, 42(4), 792-797.

8. Baker, K. R., & Scudder, G. D. (1990). Sequencing with earliness and tardiness penalties: a review. Operations research, *38*(1), 22-36.

9. Balakrishnan, N., Kanet, J. J., & Sridharan, V. (1999). Early/tardy scheduling with sequence dependent setups on uniform parallel machines. Computers & Operations Research, 26(2), 127-141.

10. Bank, J., & Werner, F. (2001). Heuristic algorithms for unrelated parallel machine scheduling with a common due date, release dates, and linear earliness and tardiness penalties. Mathematical and computer modelling, 33(4), 363-383.

11. Chang, P. C., Chen, S. H., & Lin, K. L. (2005). Two-phase sub population genetic algorithm for parallel machine-scheduling problem. Expert Systems with Applications, 29(3), 705-712.

12. Cheng, C. Y., & Huang, L. W. (2017). Minimizing total earliness and tardiness through unrelated parallel machine scheduling using distributed release time control. Journal of Manufacturing Systems, 42, 1-10.

13. Cheng, T. C. E., & Gupta, M. C. (1989). Survey of scheduling research involving due date determination decisions. European Journal of Operational Research, 38(2), 156-166.

14. Cheng, T. C. E., & Sin, C. C. S. (1990). A state-of-the-art review of parallel-machine scheduling research. European Journal of Operational Research, 47(3), 271-292.

15. de CM Nogueira, J., Arroyo, J., Villadiego, H., & Gonçalves, L. B. (2014). Hybrid GRASP heuristics to solve an unrelated parallel machine scheduling problem with earliness and tardiness penalties. Electronic Notes in Theoretical Computer Science, 302, 53-72.

16. Diana, R. O. M., de França Filho, M. F., de Souza, S. R., & de Almeida Vitor, J. F. (2015). An immune-inspired algorithm for an unrelated parallel machines' scheduling problem with sequence and machine dependent setup-times for makespan minimisation. Neurocomputing, 163, 94-105.

17. Fanjul-Peyro, L., & Ruiz, R. (2011). Size-reduction heuristics for the unrelated parallel machines scheduling problem. Computers & Operations Research, 38(1), 301-309.

18. Fanjul-Peyro, L., & Ruiz, R. (2012). Scheduling unrelated parallel machines with optional machines and jobs selection. Computers & operations research, 39(7), 1745-1753.

19. Fanjul-Peyro, L., Perea, F., & Ruiz, R. (2017). Models and matheuristics for the unrelated parallel machine scheduling problem with additional resources. European Journal of Operational Research, 260(2), 482-493.

20. Goldberg, D. (1989). Genetic algorithms in optimization, search and machine learning. Reading: Wesley-Addison.

21. Graham, R. L., Lawler, E. L., Lenstra, J. K., & Kan, A. R. (1979). Optimization and approximation in deterministic sequencing and scheduling: a survey. Annals of discrete mathematics, *5*, 287-326.

22. Heady, R. B., & Zhu, Z. (1998). Minimizing the sum of job earliness and tardiness in a multimachine system. International Journal of Production Research, 36(6), 1619-1632.

23. Holland, J. H. (1975). Adaptation in natural and artificial systems. An introductory analysis with application to biology, control, and artificial intelligence. MI: University of Michigan Press, Ann Arbor.

24. Joo, C. M., & Kim, B. S. (2015). Hybrid genetic algorithms with dispatching rules for unrelated parallel machine scheduling with setup time and production availability. Computers & Industrial Engineering, 85(9), 102-109.

25. Kanet, J. J. (1981). Minimizing the average deviation of job completion times about a common due date. Naval Research Logistics (NRL), 28(4), 643-651.

26. Kanet, J. J., & Sridharan, V. (2000). Scheduling with inserted idle time: problem taxonomy and literature review. Operations Research, 48(1), 99-110.

27. Kayvanfar, V., Komaki, G. M., Aalaei, A., & Zandieh, M. (2014). Minimizing total tardiness and earliness on unrelated parallel machines with controllable processing times. Computers & Operations Research, 41(4), 31-43.

28. Kim, D. W., Kim, K. H., Jang, W., & Chen, F. F. (2002). Unrelated parallel machine scheduling with setup times using simulated annealing. Robotics and Computer-Integrated Manufacturing, 18(3), 223-231.

29. Kim, D. W., Na, D. G., & Chen, F. F. (2003). Unrelated parallel machine scheduling with setup times and a total weighted tardiness objective. Robotics and Computer-Integrated Manufacturing, 19(1), 173-181.

30. Liaw, C. F., Lin, Y. K., Cheng, C. Y., & Chen, M. (2003). Scheduling unrelated parallel machines to minimize total weighted tardiness. Computers & Operations Research, 30(12), 1777-1789.

31. Lin, Y. K., & Lin, H. C. (2015). Bicriteria scheduling problem for unrelated parallel machines with release dates. Computers & Operations Research, 64(3), 28-39.

32. Logendran, R., McDonell, B., & Smucker, B. (2007). Scheduling unrelated parallel machines with sequence-dependent setups. Computers & Operations Research, 34(11), 3420-3438.

33. McNaughton, R. (1959). Scheduling with deadlines and loss functions. Management Science, 6(1), 1-12.

34. Mokotoff, E. (2001). Parallel machine scheduling problems: A survey. Asia-Pacific Journal of Operational Research, 18(2), 193.

35. Mokotoff, E., & Jimeno, J. L. (2002). Heuristics based on partial enumeration for the unrelated parallel processor scheduling problem. Annals of Operations Research, 117(1), 133-150.

36. Omar, M. K., & Teo, S. C. (2006). Minimizing the sum of earliness/tardiness in identical parallel machines schedule with incompatible job families: An improved MIP approach. Applied Mathematics and Computation, 181(2), 1008-1017.

37. Reeves, C. R. (1995). A genetic algorithm for flowshop sequencing. Computers & operations research, 22(1), 5-13.

38. Rocha, P. L., Ravetti, M. G., Mateus, G. R., & Pardalos, P. M. (2008). Exact algorithms for a scheduling problem with unrelated parallel machines and sequence and machine-dependent setup times. Computers & Operations Research, 35(4), 1250-1264.

39. Rodriguez, F. J., Lozano, M., Blum, C., & García-Martínez, C. (2013). An iterated greedy algorithm for the large-scale unrelated parallel machines scheduling problem. Computers & Operations Research, 40(7), 1829-1841.

40. Sels, V., Coelho, J., Dias, A. M., & Vanhoucke, M. (2015). Hybrid tabu search and a truncated branch-and-bound for the unrelated parallel machine scheduling problem. Computers & Operations Research, 53, 107-117.

41. Sivrikaya-Şerifoğlu, F., & Ulusoy, G. (1999). Parallel machine scheduling with earliness and tardiness penalties. Computers & Operations Research, 26(8), 773-787.

42. Vallada, E., & Ruiz, R. (2011). A genetic algorithm for the unrelated parallel machine scheduling problem with sequence dependent setup times. European Journal of Operational Research, 211(3), 612-622.

43. Vallada, E., & Ruiz, R. (2012). Scheduling unrelated parallel machines with sequence dependent setup times and weighted earliness–tardiness minimization. In Just-in-Time Systems (pp. 67-90). Springer New York.

44. Weng, M. X., Lu, J., & Ren, H. (2001). Unrelated parallel machine scheduling with setup consideration and a total weighted completion time objective. International journal of production economics, 70(3), 215-226.

45. Zheng, X. L., & Wang, L. (2016). A two-stage adaptive fruit fly optimization algorithm for unrelated parallel machine scheduling problem with additional resource constraints. Expert Systems with Applications, 65(3, 28-39.

46. Zhu, Z., & Heady, R. B. (2000). Minimizing the sum of earliness/tardiness in multi-machine scheduling: a mixed integer programming approach. Computers & Industrial Engineering, 38(2), 297-305.

47. Zhu, Z., Meredith, P. H., & Makboonprasith, S. (1994). Defining critical elements in JIT implementation: a survey. Industrial Management & Data Systems, 94(5), 3-10.

Apéndice A

```
class Program
    {
        static void Main(string[] args)
        {
            Simulador simulacion = new Simulador(60, 60, 3, 50, 50, 2);
            simulacion.cargarDatos("datos_6_5_49.txt");
            simulacion.ejecutar();
        }
    }
}
```

Listado 4.7: Código en C# para la clase Program.cs

```
class Individuo
    {
        private List<Maquina> maquinas;

        public Individuo(List<Maquina> maquinas_nuevas)
        {
            maquinas = new List<Maquina>();
            foreach (Maquina m in maquinas_nuevas)
            {
                maquinas.Add(new Maquina(m));
            }
        }
        public Individuo(Individuo i)
        {
            foreach(Maquina m in i.getMaquinas())
            {
                maquinas.Add(new Maquina(m));
            }
        }

        public List<Maquina> getMaquinas()
        {
            return maquinas;
        }
        public void setMaquinas(List<Maquina> v) {
            maquinas = v;
        }
        public Maquina getMaquina(int id)
        {
            Maquina m = null;
            for(int i = 0; m == null && i < maquinas.Count; i++)
            {
                if(maquinas[i].getID() == id)
                {
                    m = maquinas[i];
                }
            }
            return m;
```

```
public bool addMaquina(Maquina nueva)
        {
            bool exist = false;
            for(int i = 0; !exist && i < maquinas.Count; i++)
            {
                if(maquinas[i].getID() == nueva.getID())
                {
                    exist = true;
                }
            }
            if (!exist)
            {
                maquinas.Add(nueva);
            }
            return !exist;
        }

        public bool validar()
        {
            bool valido = true;
            for(int i = 0; valido && i < maquinas.Count; i++)
            {
                if(maquinas[i].getTrabajos().Count > 0)
                {
                    valido = maquinas[i].validar();
                }
                else
                {
                    valido = false;
                }
            }
            return valido;
        }
        public int getRatio()
        {
            int ratio = 0;
            foreach(Maquina m in maquinas)
            {
                ratio += m.getRatio();
            }
            return ratio;
        }
        public void mutar()
        {
            Random rnd = new Random();
            int maquina_elegida = rnd.Next(0, maquinas.Count);
            int trabajo_1_elegido = rnd.Next(0,
maquinas[maquina_elegida].getTrabajos().Count);
            int trabajo_2_elegido;
            do
            {
                trabajo_2_elegido = rnd.Next(0,
maquinas[maquina_elegida].getTrabajos().Count);
            } while (maquinas[maquina_elegida].getTrabajos().Count > 1 &&
trabajo_2_elegido == trabajo_1_elegido);
            maquinas[maquina_elegida].intercambiarTrabajos(trabajo_1_elegido,
trabajo_2_elegido);
        }
```

```
public List<Trabajo> getTrabajos()
        {
            List<Trabajo> trabajos = new List<Trabajo>();
            foreach(Maquina m in maquinas)
            {
                foreach(Trabajo t in m.getTrabajos())
                {
                    trabajos.Add(t);
                }
            }
            return trabajos;
        }

        public string ToString()
        {
            string datos = "Ratio: "+getRatio()+"\n";
            foreach(Maquina m in maquinas)
            {
                datos += m.ToString();
            }
            datos += validar() ? "APTO" : "NO APTO";
            return datos;
        }
        public string getInforme(bool reducido)
        {
            string informe = "";
            int total_desfase = 0;
            foreach (Maquina m in maquinas)
            {
                string[] informe_maquina = m.getInforme(reducido);
                informe += informe_maquina[0];
                total_desfase += int.Parse(informe_maquina[1]);
            }
            informe += "\nTotal Adelanto/Retraso: " + Math.Abs(total_desfase) +
"\n";
            return informe;
        }
        public void reajustar()
        {
            foreach(Maquina m in maquinas)
            {
                m.reajustar();
            }
        }
    }
}
```

Listado 4.7: Código en C# para la clase Individuo.cs

```
class Trabajo
    {
        private int ID;
        private int diasEntrega;
        private int diasLimite;
        private int diasInicio;
        private int pesoRetraso;
        private int reajuste;
        private int pesoAdelanto;
        private int[] tiempos_procesamiento;
        private int maquina_asociada;

        public int getID()
        {
            return ID;
        }
        public void setID(int v)
        {
            ID = v;
        }
        public int getDiasEntrega()
        {
            return diasEntrega;
        }
        public void setDiasEntrega(int v)
        {
            diasEntrega = v;
        }
        public int getDiasInicio()
        {
            return diasInicio;
        }
        public void setDiasInicio(int v)
        {
            diasInicio = v;
        }
        public int getDiasLimite()
        {
            return diasLimite;
        }
        public void setDiasLimite(int v)
        {
            diasLimite = v;
        }
        public int getPesoRetraso()
        {
            return pesoRetraso;
        }
        public void setPesoRetraso(int v)
        {
            pesoRetraso = v;
        }
        public int getPesoAdelanto()
        {
            return pesoAdelanto;
        }
        public void setPesoAdelanto(int v)
        {
            pesoAdelanto = v;
        }
```

```
public int[] getTiemposProcesamiento()
        {
            return tiempos_procesamiento;
        }
        public void setTiemposProcesamiento(int[] v)
        {
            tiempos_procesamiento = v;
        }
        public int getTiempoProcesamiento(int maquina)
        {
            int v = -1;
            maquina--;
            if (maquina >= 0 && maquina < tiempos_procesamiento.Length)
            {
                v = tiempos_procesamiento[maquina];
            }
            return v;
        }
        public int getTiempoProcesamiento()
        {
            int v = -1;
            if (maquina_asociada-1 >= 0 && maquina_asociada-1 <
tiempos_procesamiento.Length)
            {
                v = tiempos_procesamiento[maquina_asociada-1];
            }
            return v;
        }
        public void setTiempoProcesamiento(int maquina, int v)
        {
            maquina--;
            if (maquina >= 0 && maquina < tiempos_procesamiento.Length)
            {
                tiempos_procesamiento[maquina] = v;
            }
        }
        public int getMaquinaAsociada()
        {
            return maquina_asociada;
        }
        public void setMaquinaAsociada(int v)
        {
            maquina_asociada = v;
        }

        public Trabajo(int id, int dE, int dL, int dI, int pR, int pA, int []tP)
        {
            ID = id;
            diasEntrega = dE;
            diasLimite = dL;
            diasInicio = dI;
            pesoRetraso = pR;
            pesoAdelanto = pA;
            tiempos_procesamiento = tP;
            asignacionInicial();
            reajuste = 0;
        }
```

```
public Trabajo(Trabajo t) // Constructor
        {
            ID = t.getID();
            diasEntrega = t.getDiasEntrega();
            diasLimite = t.getDiasLimite();
            diasInicio = t.getDiasInicio();
            pesoRetraso = t.getPesoRetraso();
            pesoAdelanto = t.getPesoAdelanto();
            tiempos_procesamiento = t.getTiemposProcesamiento();
            maquina_asociada = t.getMaquinaAsociada();
            reajuste = t.getReajuste();
        }
        public void asignacionInicial() // Asignacion inicial de los trabajos de
acuerdo a SPT.
        {
            int mejor_maquina = 0;
            for(int i = 0; i < tiempos_procesamiento.Length; i++)
            {
                if(tiempos_procesamiento[i] <
tiempos_procesamiento[mejor_maquina])
                {
                    mejor_maquina = i;
                }
            }
            maquina_asociada = mejor_maquina + 1;
        }

        public int getReajuste()
        {
            return reajuste;
        }
        public void incrementarReajuste(int v)
        {
            reajuste += v;
        }
        public void decrementarReajuste(int v)
        {
            reajuste -= v;
        }
        public void setReajuste(int v)
        {
            reajuste = v;
        }

public bool validar(int diasAcumulados) //validación que los trabajos no exceden
la fecha limite. diasAcumulados representa cuando inician los trabajos.
        {
            bool valido = true;
            valido = ((diasAcumulados + tiempos_procesamiento[maquina_asociada-1])
<= diasLimite);
            return valido;
        }
```

```
public int getRatio(int maquina, int diasAcumulados) //Funcion Objetivo
        {
            int ratio = 0;
            maquina--;

            int diferencia = (diasAcumulados + tiempos_procesamiento[maquina]) -
diasEntrega;

            if (diferencia < 0)
            {
                ratio += (Math.Abs(diferencia) * pesoAdelanto);
            }
            else if(diferencia > 0)
            {
                ratio += (Math.Abs(diferencia) * pesoRetraso);
            }

            return ratio;
        }

        public bool mismoTrabajo(Trabajo t)
        {
            return t.getID() == ID && t.getMaquinaAsociada() == maquina_asociada;
        }

        public string ToString()
        {
            string datos = "ID: " + ID;
            foreach(int tp in tiempos_procesamiento)
            {
                if(datos != "")
                {
                    datos += "\t";
                }
                datos += tp;
            }
            datos += "\t" + pesoRetraso + "\t" + pesoAdelanto + "\t" + diasEntrega
+ "\t" + diasInicio + "\t" + diasLimite + "\n";
            return datos;
        }
    }
```

Listado 4.8: Código en C# para la clase Trabajo.cs

```
class Maquina
    {
        private int ID;
        private int[,] tiempos_preparacion;
        private List<Trabajo> trabajos;

        public Maquina(int id, int [,] tp, List<Trabajo> t)
        {
            ID = id;
            tiempos_preparacion = tp;
            trabajos = t;
        }
        public Maquina(Maquina m)
        {
            ID = m.getID();
            tiempos_preparacion = m.getTiemposPreparacion();
            trabajos = new List<Trabajo>();
            foreach(Trabajo t in m.getTrabajos())
            {
                trabajos.Add(new Trabajo(t));
            }
        }

        public int getID()
        {
            return ID;
        }
        public void setID(int v)
        {
            ID = v;
        }
        public int[,] getTiemposPreparacion()
        {
            return tiempos_preparacion;
        }
        public void setTiemposPreparacion(int[,] v) {
            tiempos_preparacion = v;
        }

public int getTiempoPreparacion(Trabajo actual, Trabajo nuevo)
        {
            int v = 0;
            int idNuevo = nuevo.getID() - 1;
            int idActual = actual.getID() - 1;
            if(idActual >= 0 && idActual < tiempos_preparacion.GetLength(0) &&
idNuevo >= 0 && idNuevo < tiempos_preparacion.GetLength(1))
            {
                v = tiempos_preparacion[idActual, idNuevo];
            }
            return v;

public void setTiempoPreparacion(Trabajo actual, Trabajo nuevo, int v)
        {
            int idNuevo = nuevo.getID() - 1;
            int idActual = actual.getID() - 1;
            if (idActual >= 0 && idActual < tiempos_preparacion.GetLength(0) &&
idNuevo >= 0 && idNuevo < tiempos_preparacion.GetLength(1))
            {
                tiempos_preparacion[idActual, idNuevo] = v;
            }
        }
```

```
public List<Trabajo> getTrabajos()
        {
            return trabajos;
        }
        public void setTrabajo(List<Trabajo> v)
        {
            trabajos = v;
        }
        public Trabajo getTrabajo(int i)
        {
            Trabajo t = null;
            if(i >= 0 && i < trabajos.Count)
            {
                t = trabajos[i];
            }
            return t;
        }
        public void addTrabajo(Trabajo t)
        {
            t.setMaquinaAsociada(ID);
            trabajos.Add(new Trabajo(t));
        }

public bool validar() //Validación de las secuencias de trabajos en las máquinas.
        {
            bool valido = true;

            int trabajo_actual = 0;
            int trabajo_anterior = -1;
            int total = 0;
            if (trabajos.Count > 0)
            {
                total = trabajos[trabajo_actual].getDiasInicio();
                valido = trabajos[trabajo_actual].validar(total);
            }

            while (trabajos.Count > 0 && trabajo_actual < trabajos.Count &&
valido)
            {
                total += trabajos[trabajo_actual].getTiempoProcesamiento();
                trabajo_anterior = trabajo_actual;
                trabajo_actual++;
                if (trabajo_actual < trabajos.Count)
                {
                    total += getTiempoPreparacion(trabajos[trabajo_anterior],
trabajos[trabajo_actual]);
                }
                if (trabajo_actual < trabajos.Count)
                {
                    valido = trabajos[trabajo_actual].validar(total);
                }
            }

            return valido;
```

```
public int getRatio()
        {
            int ratio = 0;

            int trabajo_actual = 0;
            int trabajo_anterior = -1;
            if(trabajos.Count > 0)
            {
                int total = trabajos[trabajo_actual].getDiasInicio();

                while (trabajo_actual < trabajos.Count)
                {
                    ratio += trabajos[trabajo_actual].getRatio(ID, total);
                    total += trabajos[trabajo_actual].getTiempoProcesamiento(ID);
                    trabajo_anterior = trabajo_actual;
                    trabajo_actual++;
                    if (trabajo_actual < trabajos.Count)
                    {
                        total += getTiempoPreparacion(trabajos[trabajo_anterior],
trabajos[trabajo_actual]);
                    }
                }
            }else
            {
                Console.WriteLine("Máquina " + ID + " vacía");
            }


            return ratio;
        }

public void intercambiarTrabajos(int t1, int t2)
        {
            if(t1 >= 0 && t1 < trabajos.Count && t2 >= 0 && t2 < trabajos.Count)
            {
                Trabajo aux = trabajos[t1];
                trabajos[t1] = trabajos[t2];
                trabajos[t2] = aux;
            }
        }
        public void limpiarCola()
        {
            trabajos.Clear();
        }
        public string ToString()
        {
            string datos = "ID: "+ID+" ["+getRatio()+"]\n";
            for(int i = 0; i < tiempos_preparacion.GetLength(0); i++)
            {
                for(int j = 0; j < tiempos_preparacion.GetLength(1); j++)
                {
                    datos += "\t" + tiempos_preparacion[i, j];
                }
                datos += "\n";
            }
            datos += "\nTrabajos M"+ID+":\n";
            foreach (Trabajo t in trabajos)
```

```
{
                datos += t.ToString();
            }
            datos += "\n\n";
            return datos;
        }
        public string[] getInforme(bool reducido)
        {
            string informe = "";
            int tAnterior = -1;
            int tActual = 0;

            int tiempo_acumulado = 0;
            tiempo_acumulado += trabajos[tActual].getDiasInicio();
            informe += "\tMáquina " + ID + " : " + ((tiempo_acumulado -
trabajos[tActual].getDiasLimite()) > 0 ? "FUERA DE LÍMITES DE ENTREGA" : "A TIEMPO
PARA ENTREGAR") + "\n";
            int desfase_total = 0;
            while (tActual < trabajos.Count)
            {
                if (tAnterior != -1)
                {
                    tiempo_acumulado += getTiempoPreparacion(trabajos[tAnterior],
trabajos[tActual]);
                }
                tiempo_acumulado += trabajos[tActual].getReajuste();
                informe += "\t\tTrabajo " + trabajos[tActual].getID() + "\n";
                informe += "\t\t\tEmpieza   : " + tiempo_acumulado +
(tiempo_acumulado != 1 ? " días" : " día") + "\n";
                tiempo_acumulado += trabajos[tActual].getTiempoProcesamiento();
                informe += "\t\t\tTermina   : " + tiempo_acumulado +
(tiempo_acumulado != 1 ? " días" : " día") + "\n";
                informe += "\t\t\tEntrega   : " +
trabajos[tActual].getDiasEntrega() + (trabajos[tActual].getDiasEntrega() != 1 ? "
días" : " día") + "\n";
                if (!reducido) { informe += "\t\t\tLímite    : " +
trabajos[tActual].getDiasLimite() + (trabajos[tActual].getDiasLimite() != 1 ? "
días" : " día") + "\n"; }
                int desfase = tiempo_acumulado -
trabajos[tActual].getDiasEntrega();
                informe += "\t\t\tDesfase   : " + (desfase > 0 ? desfase +
(desfase != 1 ? " DIAS" : " DIA") + " DE RETRASO\n" : (desfase == 0) ? "0 días
FECHA EXACTA\n" : Math.Abs(desfase) + (desfase != 1 ? " DIAS" : " DIA") + " DE
ADELANTO\n");
                desfase_total += Math.Abs(desfase);
                tAnterior = tActual;
                tActual++;
            }
            informe += "\t\t\tTotal      : " + (desfase_total > 0 ? desfase_total
+ (desfase_total != 1 ? " DIAS" : " DIA") + " DE RETRASO\n" : (desfase_total == 0)
? "0 días FECHA EXACTA\n" : Math.Abs(desfase_total) + (desfase_total != 1 ? "
DIAS" : " DIA") + " DE ADELANTO\n");

            return new string[2] { informe, ""+desfase_total};
        }
```

```
public void reajustar() //Insercion tiempos ociosos
        {   int tAnterior;
            int tActual;
            int primer_adelanto;
            List<int> retrasos;
            List<int> adelantos;
            int total_adelantos;
            int total_retrasos;

            do
            {   int minimo_adelanto = int.MaxValue;
                retrasos = new List<int>();
                adelantos = new List<int>();
                primer_adelanto = -1;
                tAnterior = -1;
                tActual = 0;
                int tiempo_acumulado = 0;
                tiempo_acumulado += trabajos[tActual].getDiasInicio();
                while (tActual < trabajos.Count)
                {
                    if (tAnterior != -1)
                    {
                        tiempo_acumulado +=
getTiempoPreparacion(trabajos[tAnterior], trabajos[tActual]);
                    }
                    tiempo_acumulado += trabajos[tActual].getReajuste();
                    tiempo_acumulado +=
trabajos[tActual].getTiempoProcesamiento();
                     int desfase = tiempo_acumulado -
trabajos[tActual].getDiasEntrega();
                    if(desfase > 0)
                    {retrasos.Add(desfase * trabajos[tActual].getPesoRetraso());
                    }
                    else if(desfase < 0)
                    { adelantos.Add(Math.Abs(desfase *
trabajos[tActual].getPesoAdelanto()));
                        if(minimo_adelanto > Math.Abs(desfase))
                        {
                            minimo_adelanto = Math.Abs(desfase);
                        }
                        if(primer_adelanto == -1)
                        {
                            primer_adelanto = tActual;
                        }
                    }
                    tAnterior = tActual;
                    tActual++;
                }
                total_adelantos = 0;
                total_retrasos = 0;
                foreach(int adelanto in adelantos) { total_adelantos +=
Math.Abs(adelanto); }
                  foreach(int retraso in retrasos) { total_retrasos += retraso;  }
                if(total_adelantos > total_retrasos)
    {  trabajos[primer_adelanto].incrementarReajuste(minimo_adelanto);
                }
            } while (total_adelantos > total_retrasos);
 }
```

Listado 4.9: Código en C# para la clase Maquina.cs

```
class Simulador
    {
        private const Int32 BufferSize = 128;
        private int generaciones;
        private int tam_poblacion;
        private int elite;
        private int prob_mutacion;
        private int prob_cruce;
        private int total_maquinas;
        private int total_trabajos;
        private int total_progenitores;
        private List<Trabajo> trabajos;
        private List<Maquina> maquinas;

        public Simulador(int gen, int tPob, int e, int pC, int pM, int prg)
        {
            generaciones = gen;
            tam_poblacion = tPob;
            elite = e;
            total_progenitores = prg;
            prob_mutacion = pM;
            prob_cruce = pC;
            trabajos = new List<Trabajo>();
            maquinas = new List<Maquina>();
        }

        public int getGeneraciones()
        {
            return generaciones;
        }
        public void setGeneraciones(int v)
        {
            generaciones = v;
        }
        public int getTamPoblacion()
        {
            return tam_poblacion;
        }
        public void setTamPoblacion(int v)
        {
            tam_poblacion = v;
        }
        public int getElite()
        {
            return elite;
        }
        public void setElite(int v)
        {
            elite = v;
        }
        public int getProbMutacion()
        {
            return prob_mutacion;
        }
        public void setProbMutacion(int v)
        {
            prob_mutacion = v;
        }
        public int getProbCruce()
        {            return prob_cruce; }
```

```
public int getTotalProgenitores()
        {
            return total_progenitores;
        }
        public void setTotalProgenitores(int v)
        {
            total_progenitores = v;
        }

        public void cargarDatos(string input_file) {
            using (var fileStream = File.OpenRead("./inputs/"+input_file))
            using (var streamReader = new StreamReader(fileStream,
Encoding.UTF8, true, BufferSize))
            {
                String line;
                total_maquinas = 0;
                total_trabajos = 0;

                //Se hace una primera lectura para obtener el total de máquinas
                line = streamReader.ReadLine();
                total_maquinas = int.Parse(line);
                //Se hace una segunda lectura para obtener el total de trabajos
                line = streamReader.ReadLine();
                total_trabajos = int.Parse(line);

                string[] delimitador = { "\t" };
                Console.WriteLine("Importando trabajos...");
                for (int trabajo = 0; trabajo < total_trabajos; trabajo++)
                {
                    line = streamReader.ReadLine();
                    String[] campos = line.Split(delimitador,
StringSplitOptions.None);
                    int id = int.Parse(campos[0]);
                    int[] tiempos_procesamiento = new int[total_maquinas];
                    for (int maquina = 0; maquina < total_maquinas; maquina++)
                    {
                        tiempos_procesamiento[maquina] =
int.Parse(campos[maquina + 1]);
                    }
                    int pRetraso = int.Parse(campos[total_maquinas + 1]);
                    int pAdelanto = int.Parse(campos[total_maquinas + 2]);
                    int fEntrega = int.Parse(campos[total_maquinas + 3]);
                    int fInicio = int.Parse(campos[total_maquinas + 4]);
                    int fLimite = int.Parse(campos[total_maquinas + 5]);

Trabajo nuevo = new Trabajo(id, fEntrega, fLimite, fInicio, pRetraso,
pAdelanto, tiempos_procesamiento);
                    trabajos.Add(nuevo);
                    Console.WriteLine((trabajo + 1) + "/" + total_trabajos);

}
                Console.WriteLine("Trabajos importados.");
                Console.WriteLine("\nImportando máquinas...");
                for (int maquina = 0; maquina < total_maquinas; maquina++)
                {
                    int[,] tiempos_preparacion = new int[total_trabajos,
total_trabajos];
                    for (int trabajo_a = 0; trabajo_a < total_trabajos;
trabajo_a++)
                    {
```

```
line = streamReader.ReadLine();
                        if (line != null)
                        {
                            String[] fila = line.Split(delimitador,
StringSplitOptions.None);
                            for (int trabajo_b = 0; trabajo_b < total_trabajos;
trabajo_b++)
                            {
                                tiempos_preparacion[trabajo_a, trabajo_b] =
int.Parse(fila[trabajo_b]);
                            }
                        }
                    }
                    Maquina nueva = new Maquina(maquina + 1,
tiempos_preparacion, new List<Trabajo>());
                    maquinas.Add(nueva);
                    Console.WriteLine((maquina + 1) + "/" + total_maquinas);
                }
                Console.WriteLine("Máquinas importadas.");
                Console.WriteLine("\nEnlazando trabajos y máquinas...");
                foreach (Trabajo t in trabajos)
                {
                    maquinas[t.getMaquinaAsociada() - 1].addTrabajo(t);
                }
                Console.WriteLine("Máquinas y trabajos enlazados.\n");

}
        }
        public void ejecutar()
        {
            //Generar la población inicial de individuos de manera aleatoria
            Console.WriteLine("> Generando población inicial...");
            List<Individuo> poblacion = generarPoblacion();

            Console.WriteLine("> Población inicial generada.\n");
            Console.WriteLine("> Ordenando población inicial...");
            ordenarPoblacion(poblacion);
            Console.WriteLine("> Población inicial ordenada.\n");
            Console.WriteLine("> Calculando generaciones...\n");
            for (int generacion = 0; generacion < generaciones; generacion++)
            {Console.WriteLine("\t> Desarrollando Generación " + (generacion +
1) + "...");

                List<Individuo> poblacion_nueva = new List<Individuo>();
                for (int i = 0; i < elite; i++) {
poblacion_nueva.Add(poblacion[i]); }
                while (poblacion_nueva.Count < tam_poblacion)
                {
                    List<Individuo> elegidos = seleccionarCruce(poblacion);
                    Individuo descendiente;
                    if (tirada(prob_cruce, 100))
                    {
                        descendiente = efectuarCruce(elegidos);
                        if (tirada(prob_mutacion, 100)) { descendiente.mutar();
}
                        elegidos.Add(descendiente);
                        Individuo elegido = seleccionTorneo(elegidos);
                        if (elegido.validar())
```

```
{
                            poblacion_nueva.Add(elegido);
                        }
                    }
                }
                ordenarPoblacion(poblacion_nueva);
                poblacion = poblacion_nueva;
                Console.WriteLine("\t> Generación " + (generacion + 1) + "
desarrollada.\n");
            }
            Console.WriteLine("> Cálculo finalizado.\n");
            Console.WriteLine("> Exportando datos a fichero...\n");
            Individuo mejor = getMejorIndividuo(poblacion);
            exportarInforme(mejor);
            Console.WriteLine("> Datos exportados a fichero.\n");
            Console.ReadKey();
        }

public void exportarInforme(Individuo mejor)
        {
            string datos = "Simulación "+ DateTime.Now.ToString("dd/MM/yyyy
hh:mm:ss") +"\n";
            datos += "Máquinas                    : " + maquinas.Count +
(maquinas.Count != 1 ? " máquinas" : " máquina")+"\n";
            datos += "Trabajos                    : " + trabajos.Count +
(trabajos.Count != 1 ? " trabajos" : " trabajo") + "\n";
            datos += "Tamaño de élite             : " + elite + (elite != 1 ? "
individuos" : " individuo") + "\n";
            datos += "Generaciones                : " + generaciones +
(generaciones != 1 ? " ciclos" : " ciclo") + "\n";
            datos += "Tamaño de población         : " + tam_poblacion +
(tam_poblacion != 1 ? " individuos" : " individuo") + "\n";
            datos += "Probabilidad de mutación    : " + prob_mutacion + "%" +
"\n";
            datos += "Probabilidad de cruce       : " + prob_cruce +
"%\n\nResultado de la secuencia :\n";
            if(mejor != null)
            {
                datos += "\nMEJOR SECUENCIA:\n";
                datos += mejor.getInforme(false);
                mejor.reajustar();
                datos += "\nSECUENCIA TRAS TIEMPOS OCIOSOS: \n";
                datos += mejor.getInforme(true);
            }
            else
            {
                datos += "NO CONCLUYENTE";
            }
            System.IO.StreamWriter file = new
System.IO.StreamWriter("./outputs/"+DateTime.Now.ToString("dd_MM_yyyy_hh_mm_ss"
) +".txt");
            file.WriteLine(datos);
            file.Close();
```

```
public List<Individuo> seleccionarCruce(List<Individuo> poblacion)//Se
seleccionan de los progenitores indicados los que se van a crucar
        {
            Random rnd = new Random();
            List<int> selecciones = new List<int>();
            List<Individuo> elegidos = new List<Individuo>();

            while (elegidos.Count < total_progenitores)
            {
                int seleccion;
                do
                {
                    seleccion = rnd.Next(0, poblacion.Count);
                } while (selecciones.Contains(seleccion));
                selecciones.Add(seleccion);
                elegidos.Add(poblacion[seleccion]);
            }

            return elegidos;
        }
        public Individuo efectuarCruce(List<Individuo> elegidos) //Realiza el
cruce con los elegidos evitando duplicar los trabajos en las máquinas.
        {
            Individuo hijo = null;
            Random rnd = new Random();
            List<Trabajo> seleccion = new List<Trabajo>();
            List<Maquina> maquinas_elegidas = new List<Maquina>();
            foreach(Maquina m in maquinas){ maquinas_elegidas.Add(new
Maquina(m)); }
            while (seleccion.Count < total_trabajos)
            {
                int ie = rnd.Next(0, elegidos.Count);
                int me = rnd.Next(0, elegidos[ie].getMaquinas().Count);
                int te = rnd.Next(0,
elegidos[ie].getMaquinas()[me].getTrabajos().Count);
                Trabajo t = new
Trabajo(elegidos[ie].getMaquinas()[me].getTrabajos()[te]);
                if (!existeTrabajo(seleccion, t))
                {
                    seleccion.Add(t);
                }
            }
            foreach(Trabajo t in seleccion)
            {
                if(t.getMaquinaAsociada() - 1 >= 0)
                {
                    maquinas_elegidas[t.getMaquinaAsociada() -
1].addTrabajo(t);
                }
            }

hijo = new Individuo(maquinas_elegidas);
            return hijo;
        }
```

```
public bool existeTrabajo(List<Trabajo> seleccion, Trabajo t)
        {
            bool existe = false;
            for(int i = 0; !existe && i < seleccion.Count; i++)
            {
                existe = seleccion[i].getID() == t.getID();
            }
            return existe;
        }

public Individuo seleccionTorneo(List<Individuo> contendientes)// Elige el
mejor individuo para ir a la nueva poblacion.
        {
            int elegido = 0;

            for(int i = 0; i < contendientes.Count; i++)
            {
                if(contendientes[i].getRatio() <
contendientes[elegido].getRatio())
                {
                    elegido = i;
                }
            }

            return contendientes[elegido];
        }
        public List<Individuo> generarPoblacion()
        {
            List<Individuo> individuos = new List<Individuo>();

            Console.WriteLine("\nGenerando individuos...");
            while (individuos.Count < tam_poblacion)
            {
                Individuo nuevo = null;
                do
                {
                    nuevo = generarIndividuo();
                } while (!nuevo.validar());
                Console.WriteLine(individuos.Count);
                individuos.Add(nuevo);
            }

Console.WriteLine("Individuos generados.");
            return individuos;

public Individuo generarIndividuo()
        {
            Individuo nuevo = null;
            Random rnd = new Random();
            List<Maquina> alelos = new List<Maquina>();
            List<Trabajo> trabajos_restantes = new List<Trabajo>();
            foreach(Trabajo t in trabajos) { trabajos_restantes.Add(new
Trabajo(t)); }
            foreach (Maquina m in maquinas) { Maquina nueva = new Maquina(m);
nueva.limpiarCola(); alelos.Add(nueva); }
            while (trabajos_restantes.Count > 0)
            {int trabajo_seleccionado;

                trabajo_seleccionado = rnd.Next(0, trabajos_restantes.Count);

                int maquina_elegida = rnd.Next(0, alelos.Count);
```

```
alelos[maquina_elegida].addTrabajo(new
Trabajo(trabajos_restantes[trabajo_seleccionado]));
                trabajos_restantes.RemoveAt(trabajo_seleccionado);

}
            nuevo = new Individuo(alelos);
            return nuevo;
        }
        public void ordenarPoblacion(List<Individuo> poblacion)
        {

            for (int i = 0; i < poblacion.Count; i++)
            {
                for (int j = 0; j < poblacion.Count; j++)
                {
                    if (poblacion[i].getRatio() < poblacion[j].getRatio())
                    {
                        Individuo aux = poblacion[i];
                        poblacion[i] = poblacion[j];
                        poblacion[j] = aux;
                    }
                }
            }
        }
        public Individuo getMejorIndividuo(List<Individuo> poblacion)
        {
            Individuo mejor = null;
            foreach(Individuo ejemplar in poblacion)
            {
                if (ejemplar.validar())
                {
                    mejor = ejemplar;
                    break;
                }
            }
            return mejor;
        }
        public bool tirada(int probabilidad, int limite)
        {
            Random rnd = new Random();

            return rnd.Next(0, limite + 1) <= probabilidad;
        }
    }
}
```

Listado 4.10: Código en C# para la clase Simulador.cs

Ejemplo del archivo de la salida con la mejor solución encontrada con los valores de la función objetivo antes y después de insertar los tiempos de inactividad.

```
Simulación 18/09/2017 10:56:35
Máquinas                    : 5 máquinas
Trabajos                    : 6 trabajos
Tamaño de élite             : 3 individuos
Generaciones                : 60 ciclos
Tamaño de población         : 60 individuos
Probabilidad de mutación    : 50%
Probabilidad de cruce       : 50%

Resultado de la secuencia :

MEJOR SECUENCIA:
    Máquina 1 : A TIEMPO PARA ENTREGAR
        Trabajo 3
            Empieza    : 12 días
            Termina    : 97 días
            Entrega    : 417 días
            Límite     : 625 días
            Desfase    : 320 DIAS DE ADELANTO
            Total       : 320 DIAS DE ADELANTO
    Máquina 2 : A TIEMPO PARA ENTREGAR
        Trabajo 6
            Empieza    : 11 días
            Termina    : 105 días
            Entrega    : 455 días
            Límite     : 682 días
            Desfase    : 350 DIAS DE ADELANTO
            Total       : 350 DIAS DE ADELANTO
    Máquina 3 : A TIEMPO PARA ENTREGAR
        Trabajo 1
            Empieza    : 13 días
            Termina    : 64 días
            Entrega    : 304 días
            Límite     : 456 días
            Desfase    : 240 DIAS DE ADELANTO
            Total       : 240 DIAS DE ADELANTO
    Máquina 4 : A TIEMPO PARA ENTREGAR
        Trabajo 5
            Empieza    : 6 días
            Termina    : 81 días
            Entrega    : 418 días
            Límite     : 627 días
            Desfase    : 337 DIAS DE ADELANTO
            Total       : 337 DIAS DE ADELANTO
    Máquina 5 : A TIEMPO PARA ENTREGAR
        Trabajo 2
            Empieza    : 3 días
            Termina    : 64 días
            Entrega    : 308 días
            Límite     : 462 días
            Desfase    : 244 DIAS DE ADELANTO
        Trabajo 4
            Empieza    : 75 días
            Termina    : 152 días
            Entrega    : 499 días
            Límite     : 748 días
            Desfase    : 347 DIAS DE ADELANTO
            Total       : 591 DIAS DE ADELANTO

    Total Adelanto/Retraso: 1838
```

```
SECUENCIA TRAS TIEMPOS OCIOSOS:
    Máquina 1 : A TIEMPO PARA ENTREGAR
        Trabajo 3
            Empieza   : 332 días
            Termina   : 417 días
            Entrega   : 417 días
            Desfase   : 0 días FECHA EXACTA
            Total      : 0 días FECHA EXACTA
    Máquina 2 : A TIEMPO PARA ENTREGAR
        Trabajo 6
            Empieza   : 361 días
            Termina   : 455 días
            Entrega   : 455 días
            Desfase   : 0 días FECHA EXACTA
            Total      : 0 días FECHA EXACTA
    Máquina 3 : A TIEMPO PARA ENTREGAR
        Trabajo 1
            Empieza   : 253 días
            Termina   : 304 días
            Entrega   : 304 días
            Desfase   : 0 días FECHA EXACTA
            Total      : 0 días FECHA EXACTA
    Máquina 4 : A TIEMPO PARA ENTREGAR
        Trabajo 5
            Empieza   : 343 días
            Termina   : 418 días
            Entrega   : 418 días
            Desfase   : 0 días FECHA EXACTA
            Total      : 0 días FECHA EXACTA
    Máquina 5 : A TIEMPO PARA ENTREGAR
        Trabajo 2
            Empieza   : 247 días
            Termina   : 308 días
            Entrega   : 308 días
            Desfase   : 0 días FECHA EXACTA
        Trabajo 4
            Empieza   : 422 días
            Termina   : 499 días
            Entrega   : 499 días
            Desfase   : 0 días FECHA EXACTA
            Total      : 0 días FECHA EXACTA

Total Adelanto/Retraso: 0
```