Trabajo Fin de Máster

Máster en Ingeniería Industrial

Programación de la producción de un Job Shop con Lot Streaming. Resolución metaheurística.

Autor: Francisco Manuel Vallejo Gómez de Travecedo

Tutor: Pedro Luis González Rodríguez

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

Sevilla, 2024

Trabajo Fin de Máster
Organización Industrial y Gestión de Empresas

Programación de la producción de un Job Shop con Lot Streaming. Resolución metaheurística.

Autor:

Francisco Manuel Vallejo Gómez de Travecedo

Tutor:

Pedro Luis González Rodríguez

Catedrático de Universidad

Dpto. Organización Industrial y Gestión de Empresas I

Escuela Técnica Superior de Ingeniería

Universidad de Sevilla

Sevilla, 2024

Trabajo Fin de Máster: Programación de la producción de un Job Shop con Lot Streaming. Resolución metaheurística.

Autor: Francisco Manuel Vallejo Gómez de Travecedo

Tutor: Pedro Luis González Rodríguez

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

Presidente:

Vocales:

Secretario:

Acuerdan otorgarle la calificación de:

Sevilla, 2024

El Secretario del Tribunal

Agradecimientos

Quisiera expresar mi más sincero agradecimiento a todas las personas que me han acompañado a lo largo de este extenso recorrido, sin las cuales no estaría hoy escribiendo estas líneas. A mis amigos de Sevilla y de distintos rincones del mundo.

A mi familia, por su apoyo incondicional a lo largo de todo este proceso, y por depositar siempre en mí su plena confianza, brindándome la seguridad necesaria para continuar.

Quisiera expresar un agradecimiento especial a Pedro por su amabilidad, cercanía y constante predisposición. Su apoyo y motivación han sido esenciales para la realización de este Trabajo Fin de Máster. Su compromiso y generosidad han convertido este proceso en una experiencia mucho más gratificante y amena.

Francisco Manuel Vallejo Gómez de Travecedo

Sevilla, 2024

Resumen

Este Trabajo Fin de Máster se centra en el diseño e implementación de un algoritmo genético para la lotificación y programación conjunta de operaciones en un entorno de tipo taller.

El proyecto abarca todas las fases del estudio, desde la revisión bibliográfica hasta la prueba y validación del algoritmo, incluyendo el modelado matemático del problema. En particular, se aborda la resolución del problema genérico y de tres variantes adicionales, derivadas de la inclusión de restricciones temporales (turnos) y de tiempos de setup dependientes de la secuencia.

Todos los modelos, heurísticas, algoritmos y demás funcionalidades han sido implementados en Python, utilizando diversas librerías especializadas. Se proporciona la arquitectura del programa empleada, junto con una descripción detallada de las principales funcionalidades.

La validación del algoritmo genético se lleva a cabo mediante el cálculo del gap de optimalidad, comparando las soluciones obtenidas con el algoritmo genético con las soluciones óptimas obtenidas a través de modelos de programación lineal entera mixta (MILP) para instancias de tamaño pequeño. Asimismo, se evalúa su rendimiento al considerar las variantes más complejas y los parámetros de una instancia de gran tamaño, que no puede ser resuelta mediante métodos exactos debido al elevado coste computacional que conlleva.

Abstract

This Master's Thesis focuses on the design and implementation of a genetic algorithm for job lot-streaming and scheduling in a job-shop environment.

The project encompasses all phases of the study, from the literature review to the testing and validation of the genetic algorithm (GA), including the mathematical modeling of the problem. Specifically, it addresses the resolution of the generic problem and three additional variants, derived from the inclusion of time constraints (shifts) and the sequence-dependent nature of setup times.

All mathematical models, heuristics, algorithms, and other functionalities have been implemented in Python, using various specialized libraries. The program architecture employed is provided, along with a detailed description of the main functionalities.

The genetic algorithm is validated by calculating the optimality gap, thanks to the optimal solutions obtained through mixed-integer linear programming (MILP) models. Furthermore, its performance is tested and validated by considering more complex variants and the parameters of a large-scale instance, which cannot be solved by exact methods due to the high computational cost involved.

ÍNDICE

Agradecimientos ..... **ix**

Resumen ..... **xi**

Abstract ..... **xiii**

Índice ..... **xv**

Índice de Tablas ..... **xvii**

Índice de Figuras ..... **xviii**

1 Introducción **21**

1.1 Contexto y justificación *21*

1.2 Objetivo general y objetivos específicos *22*

2 Estado del arte **24**

2.1 Lot Streaming Job Shop Scheduling Problem (LSJSP) *24*

2.1.1 Descripción del problema 24

2.1.2 Caracterización del problema según el tamaño de los sublotes. 27

2.1.3 Ventajas, desventajas y contrapartidas. 28

2.1.4 Técnicas de optimización 30

2.2 Algoritmo genético (GA) *33*

2.2.1 Introducción 33

2.2.2 Algoritmos genéticos en problemas LSJSP 35

2.3 Optimización Multiobjetivo en Job Shop Scheduling *37*

2.3.1 Introducción 37

2.3.2 Evaluación, algoritmos y toma de decisiones. 38

2.4 Conclusiones de la revisión de la literatura *41*

3 Modelado del problema **43**

3.1 Hipótesis y función objetivo *43*

3.2 Modelado MILP *44*

3.2.1 Notación 44

3.2.2 LSJSP 45

3.2.3 LSJSP con turnos 48

3.2.4 LSJSP con tiempos de setup dependientes de la secuencia 50

3.2.5 LSJSP con tiempos de setup dependientes de la secuencia y turnos 54

4 Diseño del algoritmo genético 57

4.1 Flujo de diseño 57

4.2 Notación 60

4.3 Representación del cromosoma 61

4.4 Decodificación y evaluación del cromosoma 63

4.4.1 Distribución de la demanda (LHS) 64

4.4.2 Programa semiactivo (RHS) 64

4.4.3 Función de aptitud 67

4.5 Inicialización de la población 68

4.6 Operadores genéticos 69

4.6.1 Segmento LHS 69

4.6.2 Segmento RHS 70

4.7 Hiperparámetros 73

4.8 Criterio de parada 75

5 Implementación en Python 77

5.1 Librerías 77

5.2 Módulos 78

5.3 Bloque principal 80

6 Experimentación numérica y validación 83

6.1 Generación de instancias (datos del problema) 84

6.2 Validación y Gap de optimalidad 87

6.2.1 LSJSP 88

6.2.2 LSJSP con turnos 89

6.2.3 LSJSP con tiempos de setup dependientes de la secuencia 91

6.2.4 LSJSP con tiempos de setup dependientes de la secuencia y turnos 92

6.3 Evolución en instancias grandes 94

6.4 Conclusiones de la experimentación 100

7 Conclusiones 102

Referencias 104

ÍNDICE DE TABLAS

Tabla 1. Modelos matemáticos desarrollados para el LSJSP. 43

Tabla 2. Notación para el algoritmo genético. 60

Tabla 3. Hiperparámetros del algoritmo genético. Efecto y valores de referencia. 74

Tabla 4. Parámetros de la instancia pequeña. 85

Tabla 5. Parámetros de la instancia grande. 87

Tabla 6. Resultados del algoritmo genético para el caso genérico con datos de una instancia pequeña. 89

Tabla 7. Resultados del algoritmo genético para el caso con turnos y tiempos de setup independientes de la secuencia con datos de una instancia pequeña. 90

Tabla 8. Resultados del algoritmo genético para el caso sin turnos y tiempos de setup dependientes de la secuencia con datos de una instancia pequeña. 91

Tabla 9. Resultados del algoritmo genético para el caso con turnos y tiempos de setup dependientes de la secuencia con datos de una instancia pequeña. 92

Tabla 10. Resultados del algoritmo genético para el caso con turnos y tiempos de setup dependientes de la secuencia con datos de una instancia pequeña. Aumento del número de lotes máximos a 3 unidades. 93

Tabla 11. Resumen de los resultados obtenidos en la experimentación. 100

ÍNDICE DE FIGURAS

Figura 1. Problema JSP: (a) sin lotificación de trabajos; y (b) con lotificación de trabajos. [Fuente: [3]]. 25

Figura 2. Comparación de sublotes con holguras y sin holguras y tamaños consistentes y variables en un *job shop* de 3 máquinas y 2 trabajos. [Fuente: [7]]. 28

Figura 3. Comparativa entre métodos integrados y métodos de dos fases para resolver el LSJSP. [Fuente: elaboración propia]. 31

Figura 4. Comparativa entre los métodos exactos y los métodos metaheurísticos. [Fuente: elaboración propia]. 32

Figura 5. Operación de recombinación entre dos individuos para intercambio de material genético. [Fuente: [15]]. 34

Figura 6. Operación de mutación. [Fuente: [15]]. 34

Figura 7. Ciclo del algoritmo genético. Inicialización (i) → Evaluación (f(x)) → Selección (Se) → Cruce (Cr) → Mutación (Mu) → Evaluación (f(x)) → Reemplazo (Re). El símbolo ? es la condición de parada. X* es la mejor solución. [Fuente: [15]]. 35

Figura 8. Representación de la solución. El cromosoma está formado por dos segmentos: LHS (Left Hand Side) y RHS (Right Hand Side). [Fuente: [17]]. 36

Figura 9. Frente de Pareto para los objetivos makespan y consumo de energía en un job shop. [Fuente: [19]] 38

Figura 10. Clasificación de soluciones según dominancia. [Fuente: [21]] 40

Figura 11. Precedencias en el modelo LSJSP. Se conoce que P1-L0 precede a P0-L1 pero no si lal precedencia es inmediata. En este caso, otros lotes se procesan entre estos 2. [Fuente: elaboración propia]. 50

Figura 12. Representación de los dummy job en la secuencia de producción de un job shop con 3 máquinas y 2 trabajos, cada uno dividido en 2 lotes. [Fuente: elaboración propia]. 51

Figura 13. Flujo de diseño del algoritmo genético. [Fuente: elaboración propia] 59

Figura 14. Representación del segmento izquierdo del cromosoma (LHS) para una instancia con número máximo de lotes $U = 3$. $\alpha ju \in [0,1]$. [Fuente: elaboración propia] 62

Figura 15. Representación del segmento derecho del cromosoma (RHS) para una instancia de 3 máquinas, 3 trabajos y 3 lotes. Todos los lotes pasan por todas las máquinas. [Fuente: elaboración propia] 63

Figura 16. Representación de la solución (cromosoma) usada en el algoritmo genético. [Fuente: elaboración propia] 63

Figura 17. Operador genético SPC-1. [Fuente: elaboración propia]. 69

Figura 18. Operador genético SPC-2. [Fuente: elaboración propia]. 70

Figura 19. Operador genético PMX. [Fuente: [25]]. 71

Figura 20. Operador genético OX. [Fuente: [25]]. 72

Figura 21. Operador genético JL. [Fuente: elaboración propia]. 73

Figura 22. Diagrama de flujo del algoritmo genético. [Fuente: elaboración propia]. 76

Figura 23. Módulos implementados en Python. funciones principales, librerías usadas y flujo. [Fuente: elaboración propia]. 80

Figura 24. Evolución del mejor fitness en una instancia grande considerando restricción de turnos y tiempos de setup independientes de la secuencia. [Fuente: elaboración propia]. 96

Figura 25. Diagrama de Gantt obtenido para la instancia grande tras 1000 generaciones considerando restricciones de turnos y tiempos de setup independientes de la secuencia. [Fuente: elaboración propia]. 97

Figura 26. Evolución del mejor fitness en una instancia grande considerando restricción de turnos y tiempos de setup dependientes de la secuencia. [Fuente: elaboración propia]. 98

Figura 27. Diagrama de Gantt obtenido para la instancia grande tras 1000 generaciones considerando restricciones de turnos y tiempos de setup dependientes de la secuencia. [Fuente: elaboración propia]. 99

Figura 28. Extracto temporal de un diagrama de Gantt generado con *plotly.express.timeline*. [Fuente: elaboración propia]. 99

1 INTRODUCCIÓN

En este capítulo se expone el objetivo principal del proyecto, la forma en la que se aborda y las razones que justifican su elección. Complementariamente, se ofrecen los objetivos específicos que ayudan a desgranar el proyecto y que son necesarios para cumplir el objetivo general.

1.1 Contexto y justificación

En los últimos años, la gestión de la cadena de suministro ha experimentado un cambio de paradigma, pivotando hacia la logística como factor clave de competitividad Además existe una tendencia creciente hacia la personalización de los productos. Surge la necesidad de replantear los sistemas productivos para reducir tiempos de producción, mejorar la capacidad de respuesta a la demanda, aumentar la flexibilidad en la fabricación de productos diversos y optimizar la gestión logística integrada.

La programación de la producción es una actividad crítica en la gestión de la cadena de suministro. Por lo tanto, una mejora en esta actividad puede tener importantes repercusiones en el sistema productivo. Una forma de obtener mejores resultados es implementar técnicas de lotificación, que permiten reducir el trabajo en proceso (WIP), mejorar la utilización de recursos y aumentar la flexibilidad del sistema.

El problema Lot Streaming Job Shop Scheduling (en español se podría denominar "lotificación y secuenciación de operaciones conjunta en entornos tipo taller") aborda de forma íntegra ambos retos, la lotificación y la programación de operaciones. La importancia económica de esta mejora es sustancial. La optimización de la programación de la producción mediante la lotificación puede resultar en reducciones significativas en los costes operativos, mejoras en el cumplimiento de plazos de entrega y una utilización más eficiente de los recursos disponibles. Esto se traduce en una ventaja competitiva tangible para las empresas que implementen estos sistemas. El impacto social de estas mejoras también es relevante, ya que contribuyen a una mejor planificación laboral y optimización de turnos de trabajo, aspectos que afectan directamente al bienestar de los trabajadores.

Además, la mejora en la eficiencia de recursos tiene un impacto positivo en el medio ambiente.

En el Trabajo Fin de Máster [1] realizado previo a este se estudió la resolución del problema genérico mediante métodos exactos. Se encontró una limitación importante: la incapacidad para obtener soluciones factibles para instancias de gran tamaño. Por ello, surge la necesidad de desarrollar un método aproximado que, aun renunciando al criterio de optimalidad pueda aportar soluciones de calidad. Dentro de las alternativas parece adecuado el uso de una metaheurística con la que superar esta limitación. Además, se busca aportar una base sólida sobre la que desarrollar soluciones que puedan ser implementadas en la industria. Por ello se decide ampliar la mira y abordar el problema considerando restricciones temporales de turnos y tiempos de setup dependientes de la secuencia. Ambas características son comunes en la industria y muy importantes a la hora de programar la producción.

En conclusión, la lotificación y secuenciación conjunta de operaciones es un problema de combinatoria complejo para el cual es necesario implementar una metaheurística para solucionar problemas en escenarios industriales reales. Una vez implementado e integrado en el proceso productivo, puede suponer una ventaja competitiva.

1.2 Objetivo general y objetivos específicos

El objetivo principal es el desarrollo de una metaheurística capaz de resolver instancias grandes del problema LSJSP considerando restricciones de turnos y tiempos de setup dependientes de la secuencia. De esta forma, se busca aportar una base sólida sobre la que trabajar este problema y desarrollar técnicas y herramientas avanzadas y escalables que puedan ser utilizadas en la industria.

Para alcanzar el objetivo general planteado, se divide el problema en los siguientes objetivos específicos:

- **OE 1. Revisar la literatura científica y explicar el problema.** Antes de diseñar la metaheurística, es preciso describir el problema y conocer como se ha tratado en la comunidad científica. Será fundamental saber cómo caracterizarlo, conocer a fondo que ventajas y desventajas aporta este enfoque y cuáles son las técnicas empleadas en su resolución.

- **OE 2. Modelar el problema genérico y las variantes a resolver.** Para poder evaluar posteriormente la metaheurística desarrollada, es conveniente modelar el problema matemáticamente con el fin de obtener la solución óptima de una determinada instancia. Con la solución óptima, es inmediato obtener el gap de optimalidad que presentan las soluciones obtenidas con la metaheurística.

- **OE 3. Diseñar e implementar la metaheurística.** Diseñar, explicar e implementar los mecanismos de diversificación e intensificación, así como definir los hiperparámetros. En esto se incluye el diseño de la codificación, decodificación y evaluación de las soluciones.

- **OE 4. Implementar en Python la metaheurística.** Implementar todas las funcionalidades necesarias para ejecutar la metaheurística diseñada y sus correspondientes mecanismos. Explicar cómo se estructura el código, que librerías se usan y cuál es el bloque principal.

- **OE 5. Probar y validar la metaheurística.** Se busca confirmar que la metaheurística desarrollada tiene la habilidad de solucionar instancias pequeñas y hallar soluciones de alta calidad cercanas al óptimo. Para instancias grandes, comprobar que la evolución en la exploración del espacio de soluciones es satisfactoria y que se tiene la capacidad de encontrar soluciones factibles y de buena calidad.

2 ESTADO DEL ARTE

En este capítulo se hace una breve revisión de la literatura disponible relevante para el presente trabajo. En el apartado 2.1 se ofrece una introducción concisa al problema Lot Streaming Job Shop Scheduling (LSJSP). En el apartado 2.1.1 se describe el problema general, dejando claros cuales son los subproblemas que resolver. Se da una visión global de la elaboración de artículos científicos que tratan este problema. En el apartado 2.1.2 se detalla la clasificación de problemas según el tamaño de los sublotes y las holguras que se permiten entre estos. Adicionalmente, se tratan las ventajas, desventajas y contrapartidas que implica la división de trabajos en lotes más pequeños (2.1.3). Por último, se hace un repaso general de las técnicas de optimización empleadas en la resolución del problema (2.1.4).

En el apartado 2.2 se hace una breve introducción a la metaheurística más común en la resolución de este tipo de problemas, el algoritmo genético. Se explica muy brevemente el mecanismo de esta metaheurística para encontrar soluciones buenas y mejorarlas (2.2.1). Se revisan, de entre los pocos artículos que hay, aquellos que estudian la resolución del problema con un algoritmo genético. Sin duda, Defersha [2] es el autor que más ha profundizado en este campo. (2.2.2).

En el apartado 2.3 se hace una breve introducción a la optimización multiobjetivo en entornos tipo Job Shop. Aunque en este trabajo no se trata un problema multiobjetivo como tal, existe una restricción (ventanas temporales o turnos) que se modela como un objetivo. Además, es interesante tener una visión global para futuros trabajos.

Por último, en el apartado 2.4, se ofrecen las conclusiones extraídas de la literatura científica revisada.

2.1 Lot Streaming Job Shop Scheduling Problem (LSJSP)

2.1.1 Descripción del problema

El problema, denominado en inglés Lot Streaming Job Shop Scheduling (LSJSP), es un problema complejo de

optimización en el campo de la programación de la producción. Combina dos conceptos claves: la lotificación (Lot Streaming*)* y la programación de operaciones (Scheduling*)* en un entorno tipo taller (Job Shop*)*.

El problema de programación de operaciones en un entorno tipo taller puede ser descrito de la siguiente manera: sea un entorno productivo donde N trabajos necesitan ser procesados en M máquinas. Cada trabajo debe seguir una ruta preestablecida a través de las máquinas y cada máquina puede procesar un solo trabajo en el mismo tiempo. Por lo tanto, el problema consiste en encontrar un programa factible que maximice o minimice un objetivo, por ejemplo, el tiempo total de procesado (makespan).

Por otro lado, además, hay que considerar el **problema de lotificación**. Tradicionalmente, en un job shop, un trabajo debe ser procesado en su totalidad en una máquina antes de entrar en la siguiente máquina. La lotificación (lot streaming) introduce el concepto de dividir un trabajo en trabajos más pequeños, o lo que es análogo, dividir un lote en lotes más pequeños.

En la Figura 1 se ofrece una comparativa de una programación de un job shop con y sin lotificación.

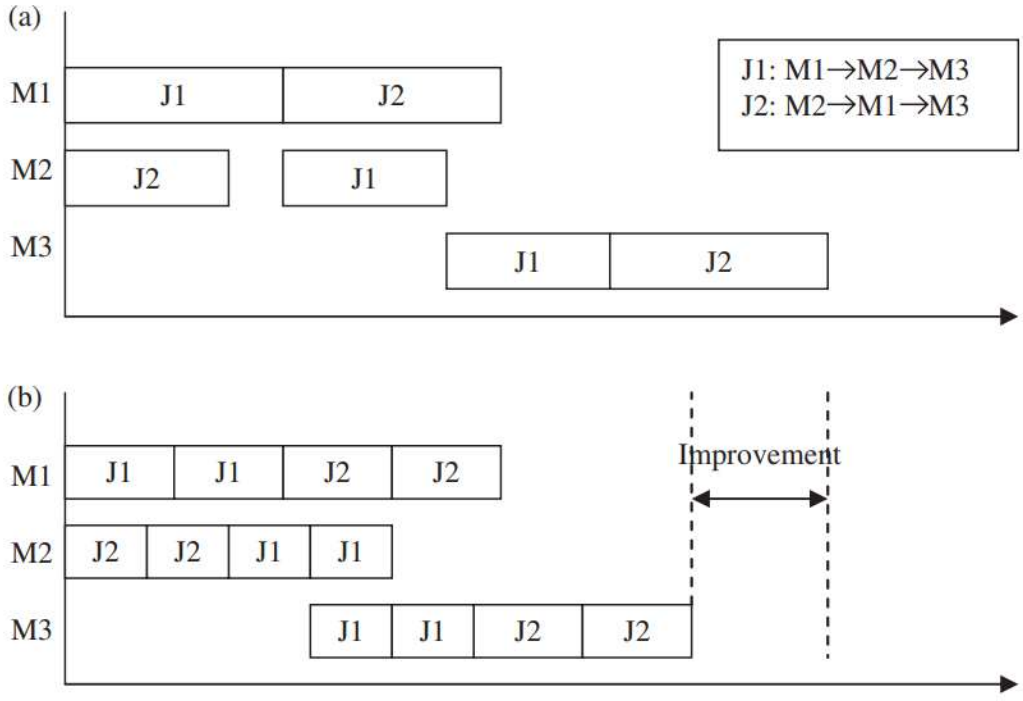

Figura 1. Problema JSP: (a) sin lotificación de trabajos; y (b) con lotificación de trabajos. [Fuente: [3]].

El objetivo principal de este enfoque es mejorar la eficiencia productiva mediante la posibilidad de procesar en

paralelo diferentes sublotes del mismo trabajo. Después, el objetivo específico puede variar. Los más comunes, en este contexto, son la minimización del makespan, el trabajo en proceso, el tiempo en proceso, etc. Sin embargo, existen múltiples escenarios reales posibles. El objetivo dependerá de las metas específicas de la empresa, las características del sistema de producción y las restricciones existentes [4].

La solución del problema será un programa de producción. Este programa aborda 3 decisiones clave:

1. Determinar el número óptimo de sublotes para cada trabajo.
2. Determinar el tamaño de cada sublote.
3. Programar las operaciones de todos los sublotes en las máquinas correspondientes.

El problema es mucho más complejo que la programación de un job shop tradicional, debido a la dimensión adicional que nace de la subdivisión de los trabajos. Además, no solo requiere hacer la programación, ya que de manera simultánea, se debe optimizar la lotificación.

A pesar de su complejidad, el LSJSP tiene un gran nicho de mercado en la industria, en diferentes sectores. Por ejemplo, en el sector aeronáutico, multitud de empresas son subcontratas de grandes empresas como Airbus o Boeing. El entorno tipo taller es común entre ellas. Como ya se ha dicho, este enfoque a la hora de programar la producción puede llevar a mejoras sustanciales en la eficiencia del sistema productivo y en la capacidad de respuesta a la demanda de los clientes.

Mientras que el problema lot streaming ha sido ampliamente estudiado para entornos flow shop, hay una carencia importante en el número de artículos que lo tratan para un entorno job shop. Representan un 10 % de la literatura que trata lot streaming [5]. Esta laguna presenta multitud de oportunidades para futuras investigaciones en las que se aborde la resolución de diversos escenarios con diversas metodologías.

La revisión llevada a cabo por Salazar-Moya y García [5] arroja los siguientes datos sobre la distribución geográfica de los artículos publicados:

- El 70 % de las publicaciones proviene de Asia (la contribución de China es del 43%)
- El 14 % proviene de Norte America.
- El 10% proviene de Europa.

- El 6 % proviene de Sudamérica.

Esta distribución sugiere que el continente asiático, y más en concreto, China, es la región en la que más foco se pone en la optimización de la producción.

Históricamente la investigación centrada en este tema ha sido limitada y escasa pero, sin embargo, en los últimos años, parece haber más interés. Probablemente esto está relacionado con la complejidad del problema y los avances en computación. Ambos, algoritmos y solvers para modelos matemáticos de programación lineal requieren una gran capacidad de procesamiento.

2.1.2 Caracterización del problema según el tamaño de los sublotes.

En la literatura, cada artículo aborda características específicas de diferentes entornos de producción y diferentes objetivos, únicos o múltiples. Más allá de esta caracterización, para problemas de lot streaming Lei et al proponen una clasificación según la coherencia del tamaño de los sublotes en las distintas máquinas [6]. Se tiene:

a) **Tamaños iguales (Equal sizes).** Todos los sublotes de un trabajo tienen el mismo tamaño.

b) **Tamaños consistentes (Consistent sizes).** Los sublotes de un mismo trabajo pueden tener tamaño diferente. El tamaño de cada sublote permanece constante a medida que pasa por las diferentes máquinas.

c) **Tamaños variables (Variable sizes).** No se impone ninguna restricción en el tamaño de los sublotes. Cada sublote de un mismo trabajo puede tener un tamaño diferente y puede cambiar su tamaño a medida que pasa por las diferentes máquinas.

Por otro lado, Chan et al. proponen una clasificación basada en la continuidad del procesado de los sublotes. Para los 3 casos vistos, se añade una característica más. Permitir o no permitir holguras o tiempos muertos (intermingling o idling) entre sublotes del mismo trabajo [3]. En la Figura 2 se muestra el efecto de esta característica.

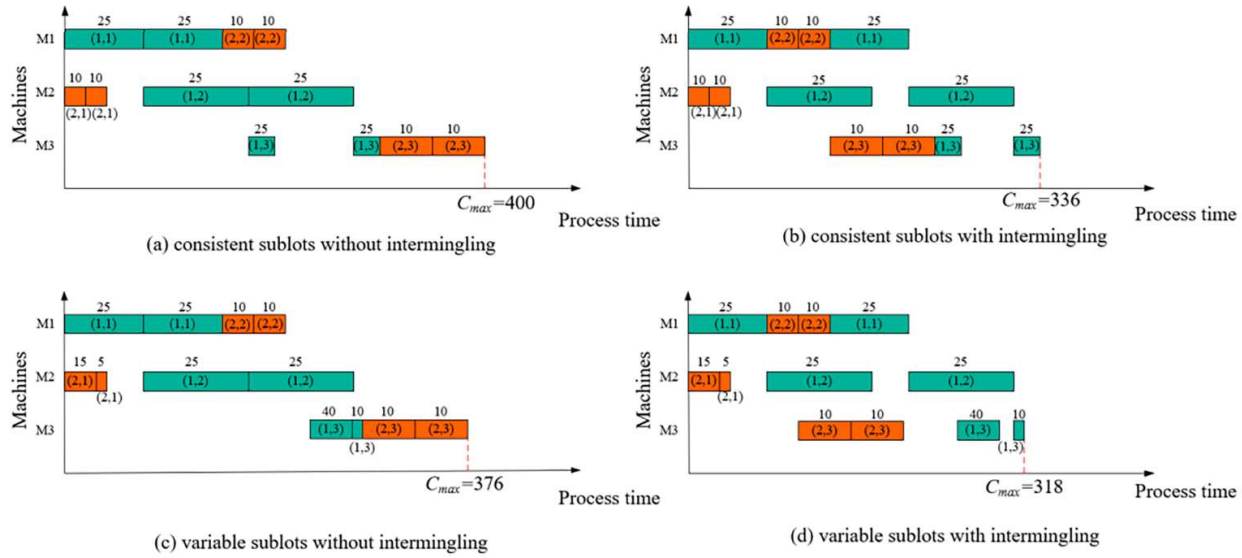

Figura 2. Comparación de sublotes con holguras y sin holguras y tamaños consistentes y variables en un *job shop* de 3 máquinas y 2 trabajos. [Fuente: [7]].

Por lo general, los problemas en los que se consideran tamaños variables son los más complejos a la hora de modelarlos y resolverlos.

Respecto a las holguras, permitirlas (intercalando sublotes de diferentes trabajos o tiempos muertos entre ellos) aumenta el espacio de soluciones posibles, y por lo tanto la complejidad del problema.

La elección del tipo de problema dependerá normalmente del entorno y las restricciones específicas de producción. Algunos procesos pueden requerir tamaños consistentes debido a limitaciones en las máquinas o en la transferencia de lotes entre máquinas. No obstante, es claro que, la consideración de lotes de tamaños variables y tiempos muertos hace que aumente la flexibilidad del problema y por lo tanto puede llevar a mejores soluciones. Sin embargo, requieren técnicas de optimización más complejas.

2.1.3 Ventajas, desventajas y contrapartidas.

Low et al. ofrecen un estudio detallado de los beneficios de la lotificación o lot streaming en entornos tipo taller [8]. En líneas generales, las ventajas y desventajas son:

A. **Ventajas.**

1. **Reducción del makespan.** La lotificación permite el procesamiento en paralelo de sublotes de un mismo trabajo en las diferentes máquinas. Esto puede reducir considerablemente el tiempo máximo de terminación de todos los trabajos (makespan).

2. **Menor inventario de trabajo en proceso (WIP).** Procesar lotes más pequeños permite tener un menor tamaño de inventario esperando entre diferentes máquinas. Esto implica la reducción del capital inmovilizado en productos parcialmente terminados.
3. **Reducción del tiempo de flujo (Flow Time).** El tiempo desde que empieza el procesado de un trabajo hasta que se completa puede ser reducido. Esto mejora la capacidad de respuesta a nuevas órdenes de clientes y puede contribuir a reducir los tiempos de entrega.
4. **Mejor utilización de máquinas.** Los tiempos muertos u ociosos pueden ser reducidos cuando se permite intercalar sublotes de diferentes trabajos. El efecto inmediato es un uso más eficiente de los recursos de producción.
5. **Aumento de la flexibilidad.** Procesar lotes más pequeños permite respuestas más ágiles a cambios en la demanda o en las prioridades.

2. **Desventajas.**

1. **Aumento de la frecuencia de los setup.** Independientemente de si los tiempos de setup son dependientes o no de la secuencia, en general, más lotes significa más setups, lo que puede aumentar el tiempo y coste total del proceso. Este efecto será especialmente importante en industrias donde los tiempos de setup son considerables.
2. **Mayores costes de transporte.** Mover más lotes significa un aumento del transporte interno y de la complejidad del sistema.
3. **Mayor complejidad en el control de la producción.** La gestión y el seguimiento de múltiples sublotes de trabajos diferentes aumenta la complejidad de los sistemas de control de la producción.
4. **Problemas potenciales en la calidad.** En algunos casos la lotificación puede llevar a una mayor variabilidad en la calidad del producto.
5. **Mayor coste computacional.** Los problemas de optimización son más complejos cuando se aplica el lot streaming. Consecuentemente, el coste computacional aumenta y se requieren técnicas de optimización más complejas y refinadas y adaptadas a sistemas de producción específicos.

Dauzère-Pérès y Lasserre presentaron algunos de los primeros artículos que tratan este problema. En uno de ellos, también ponen encima de la mesa las contrapartidas del lot streaming en job shops [9]. Constatan de manera detallada las ventajas y desventajas ya comentadas.

3. **Contrapartidas.**

1. **Número de sublotes VS Eficiencia.** Aumentar el número de sublotes generalmente mejora el makespan y el tiempo de flujo, sin embargo, como ya hemos comentado, aumenta los tiempos

y costes de setup y transporte entre estaciones. Se debe encontrar un equilibrio óptimo para cada entorno de producción específico. Tener en cuenta la ratio tiempos de setup / tiempos de procesamiento es fundamental.

2. **Consistencia de los sublotes VS Flexibilidad.** Mientras que los sublotes de tamaño consistente son más fáciles de manejar y programar, tamaños variables pueden ofrecer una mejor programación y mayor flexibilidad a cambio de un aumento en la complejidad del problema.

3. **Eficiencia a corto plazo VS Flexibilidad a largo plazo.** La optimización del programa de producción probablemente implique sublotes de un tamaño muy específico, no estándar. Sin embargo, la estandarización de los tamaños de los sublotes ofrece más flexibilidad en el largo plazo para incluir trabajos aún desconocidos. Aunque la programación de la producción en principio debería plantearse desde el punto de vista operacional y no estratégico, será la empresa la que finalmente decida, de acuerdo con sus objetivos estratégicos, cuál es el horizonte temporal adecuado.

2.1.4 Técnicas de optimización

El problema LSJSP hereda la propiedad **NP-hard** del problema clásico de programación de la producción en job shops. Encontrar soluciones óptimas comienza a ser intratable desde el punto de vista del coste computacional a medida que aumenta el tamaño del problema. Además, la introducción de sublotes aumenta considerablemente el número de programas factibles. Al fin y al cabo, cada sublote es tratado como un trabajo individual.

Además, la optimización simultánea del número y tamaño de sublotes y de la programación aumenta de manera importante la complejidad del problema. No es solo que se tenga un espacio de soluciones muy grande, también es que cambios en una variable, por ejemplo, el tamaño de un sublote puede afectar drásticamente a las otras variables. Esto complica aún más la exploración del espacio de soluciones.

En los artículos revisados, en general, el problema se aborda de 2 maneras diferentes:

1. **Métodos de 1 fase integrada.** Es el caso de aquellos métodos en los que se resuelve de forma simultánea la lotificación y la programación de la producción. Potencialmente conduce a mejores soluciones, pero aumenta la complejidad del problema. Por ejemplo, Novas ofrece un modelo de programación con restricciones para solucionar conjuntamente ambos aspectos [10]. Xie et al. ofrecen una metaheurística más compleja, que además aborda varios objetivos [7].

2. **Métodos de 2 fases.** En este caso, se resuelve primero la lotificación y luego la programación o viceversa.

   2.1. **Lotificación primero, programación después.** Este método simplifica en gran medida la programación, sin embargo, no garantiza alcanzar el óptimo global del problema. Las soluciones pueden tener una calidad pobre.

   En algunos casos se opta por añadir una tercera fase iterativa para mejorar la lotificación.

   2.2. **Programación primero, lotificación después.** Aquí se puede destacar el enfoque innovador ofrecido por Bożek y Werner. En la primera fase se crea una programación con sublotes del menor tamaño posible. En la segunda fase se optimiza el número y tamaño de los sublotes. Aunque el enfoque es innovador e ingenioso, los costes computacionales resultan ser extremadamente altos [11].

En la Figura 3 se ofrece una comparativa resumida entre ambos métodos.

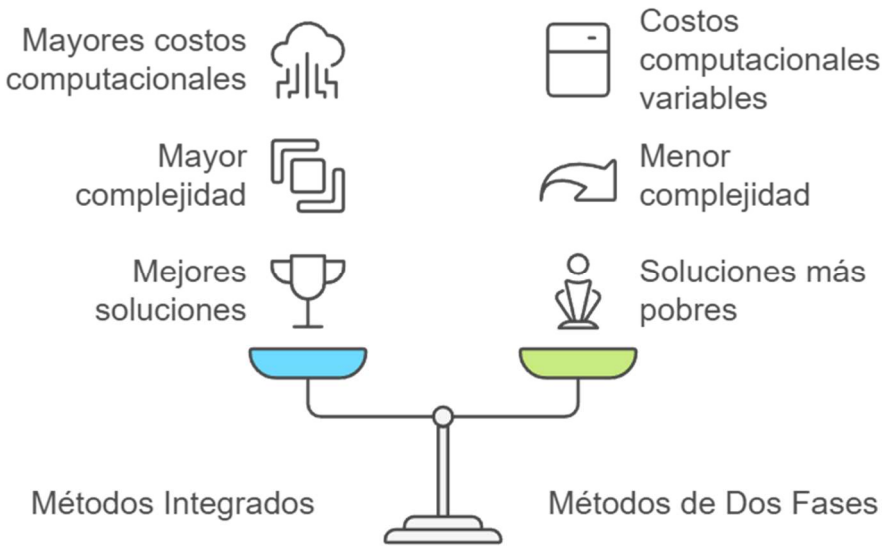

Figura 3. Comparativa entre métodos integrados y métodos de dos fases para resolver el LSJSP. [Fuente: elaboración propia].

Aunque es reducido el número de artículos que tratan este problema, el rango de técnicas utilizadas para obtener soluciones es amplio. Se pueden agrupar en 4 tipos:

1. **Métodos exactos.** Son aquellos que garantizan la optimalidad, pero son costosos desde el punto de vista computacional para instancias de gran tamaño. Se ofrecen modelos de Programación Matemática (MILP, MIP) y modelos de Programación con Restricciones (CP). Estos solo resultan apropiados para instancias de tamaño medio y pequeño.

   Por otro lado, casi todos los artículos que ofrecen métodos metaheurísticos ofrecen también un modelo matemático del problema.

2. **Heurísticas.** Se pueden encontrar en los artículos más antiguos de la década de los 90, ofrecidos por Dauzère-Pérès y Lasserre (ver por ejemplo [9], [12], [13]). Son específicos y con técnicas que ya han sido

superadas, por lo que no son de gran interés.

3. **Metaheurísticas.** Son los métodos más presentes en los artículos de los últimos años y sin duda los más eficientes en términos de coste computacional. El más común es el Algoritmo Genético (GA) y sus derivados, pero también se ofrecen otros como la Búsqueda Tabú (TS), Algoritmo de Colonia de Hormigas (PSO) y Algoritmo de Colonia de Abejas Artificiales (ABC). En particular y más recientemente, Xie et al. ofrecieron un nuevo tipo de metaheurística llamada Jaya (JA) modificada [7].
4. **Metaheurísticas híbridas o mateheurísticas.** Son métodos que combinan múltiples técnicas, obteniendo las bondades de cada una. Por ejemplo, Defersha y Bayat Movahed ofrecen un Algoritmo Genético asistido con Programación Lineal [4].

La tendencia actual es el uso de metaheurísticas avanzadas y técnicas híbridas, cada vez más complejas y adaptadas a problemas con restricciones y objetivos específicos.

Es claro que los métodos exactos están limitados a instancias pequeñas y que para resolver problemas de casos reales en la industria es fundamental desarrollar métodos aproximados que ofrezcan una buena solución con un coste computacional aceptable. Sin embargo, para modelar, desarrollar y validar estos métodos es conveniente contar con métodos exactos.

En la Figura 4 se ofrece una comparativa destacando las características más importantes de los métodos exactos y los métodos metaheurísticos.

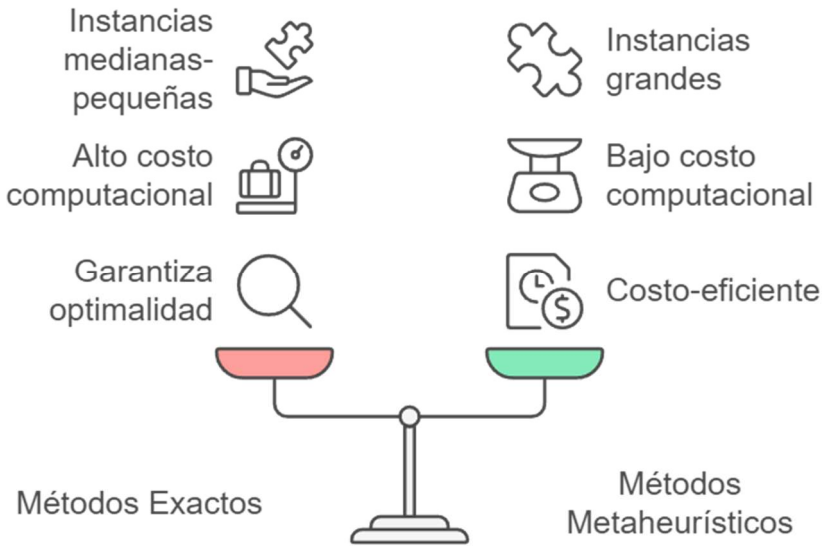

Figura 4. Comparativa entre los métodos exactos y los métodos metaheurísticos. [Fuente: elaboración propia].

2.2 Algoritmo genético (GA)

2.2.1 Introducción

El problema LSJSP discutido en el presente proyecto está clasificado como NP-hard y por lo tanto existe la necesidad de aplicar técnicas de optimización potentes para encontrar buenos resultados en instancias de tamaños medianos-grandes.

Se ha optado por trabajar con el algoritmo genético, por ser una metaheurística flexible y escalable, y que por lo tanto sirve de base para estudiar otros problemas con objetivos diferentes o multiobjetivo (mediante NSGA-II o similar). Además, puede ser fácilmente combinada con otras técnicas de optimización.

El algoritmo genético es una metaheurística basada en mecanismos genéticos y de selección natural. Pertenece al grupo de los algoritmos evolutivos (EA). El concepto de algoritmo genético fue introducido por Holland en 1962 [14]. Desde entonces, el concepto se ha desarrollado y el algoritmo ha sido usado en múltiples campos del conocimiento, entre ellos la optimización. En inglés se denota como Genetic Algorithm (GA).

El algoritmo se basa en la idea de que un grupo de soluciones, llamado población, pueden combinarse entre ellas a través de mecanismos de reproducción para generar soluciones aún mejores. En líneas generales, las fases del algoritmo son:

1. **Inicialización de la población y selección de padres**.
    1.1. Se **genera una población**, un número determinado de soluciones (a las que se le llaman individuos) para formar parte de la población inicial. El método más común y sencillo es la generación aleatoria. Garantiza la diversidad genética inicial, importante para cubrir varias partes del espacio de soluciones. Sin embargo, estas soluciones pueden ser de poca calidad.
    1.2. Para mejorar la calidad de la población inicial, se **seleccionan los padres** (ciertos individuos de la población inicial) para generar descendencia. Para esta selección existen varios métodos, los más comunes son la **selección por torneo y la selección por ruleta**. El proceso de selección se repite hasta tener una población inicial del tamaño deseado.
2. **Evolución de la población**. Se aplican mecanismos evolucionarios para generar soluciones nuevas a partir de los individuos de la población inicial. Se lleva a cabo:
    2.1. La **recombinación o cruce (crossover)**, operación en la que se combinan dos individuos (padres) para crear nuevos (hijos). Se intercambia información de los padres.

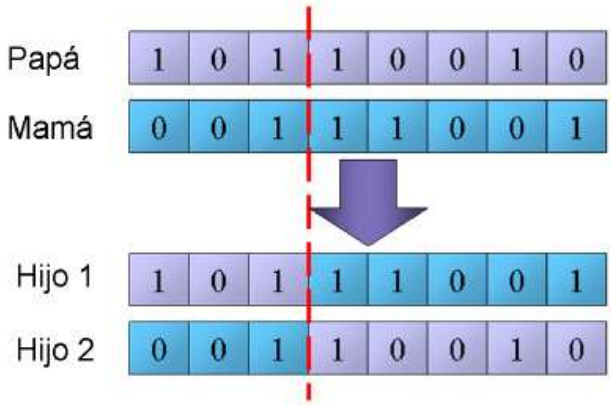

Figura 5. Operación de recombinación entre dos individuos para intercambio de material genético. [Fuente: [15]].

2.2. La **mutación**, operación en la que a ciertos individuos se le alteran uno o más genes (los elementos de información que componen una solución). Con esto se introduce variabilidad y se evita que el algoritmo se estanque en óptimos locales.

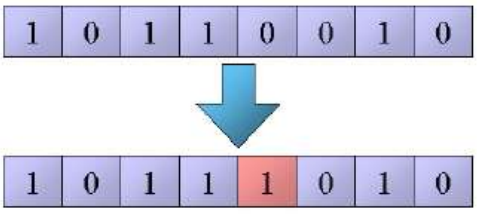

Figura 6. Operación de mutación. [Fuente: [15]].

3. **Reemplazo de la población.** Se decide qué individuos formarán parte de la nueva generación. Se puede optar por reemplazar la población completa o por aplicar elitismo: conservar los mejores individuos de la generación anterior junto con los mejores de la nueva.

4. **Condición de parada.** Los pasos 2 y 3 se repiten hasta que se cumple alguna condición de terminación. Esta puede ser un número máximo de generaciones, convergencia o un criterio de objetivo alcanzado.

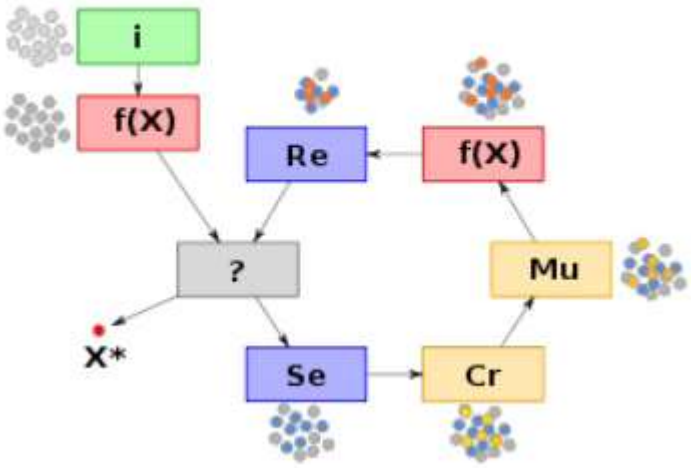

Figura 7. Ciclo del algoritmo genético. Inicialización (i) → Evaluación (f(x)) → Selección (Se) → Cruce (Cr) → Mutación (Mu) → Evaluación (f(x)) → Reemplazo (Re). El símbolo ? es la condición de parada. X* es la mejor solución. [Fuente: [15]].

Para diseñar el algoritmo es fundamental decidir como representar la solución, a lo que se le llama **representación del cromosoma**. El cromosoma suele ser una cadena de valores (binarios, reales, enteros, etc.). Cada valor representa un gen, una pieza de información de la solución. La cadena de valores (el cromosoma) representa una solución factible.

Otro aspecto a tener en cuenta es el **diseño de la función fitness.** Se trata de implementar una función para evaluar la calidad de la solución. Evidentemente, la función fitness debe reflejar el objetivo del problema.

2.2.2 Algoritmos genéticos en problemas LSJSP

Inicialmente, el desarrollo de problemas de lot streaming se centró en los entornos Flow Shop. En la literatura el número de artículos de este tipo de entorno es significativamente superior a aquellos que tratan con entornos Job Shop [5]. Sin embargo, debido al auge de la producción en lotes pequeños, los esfuerzos en el campo de la investigación han pivotado gradualmente hacia los entornos Flexible Job Shop (FJS). Por lo tanto, el número de artículos de este entorno también es superior al número de artículos que tratan el entorno Job Shop clásico.

El método más común en la resolución de problemas de lot streaming en entornos job shop es el algoritmo genético y sus variantes.

Chan et al. abordan el problema dividiéndolo en 2 fases: por un lado, la determinación de las condiciones de la lotificación y por otro la programación de los lotes. Aunque hacen uso del algoritmo genético para resolver

ambas fases, no abordan el problema de forma conjunta. Consideran lotes de tamaño consistente [3].

Liu et al. propusieron un enfoque similar. En este caso, la programación de los trabajos se hace mediante reglas de prioridad y la lotificación con un algoritmo genético. En cada cromosoma se incluye información sobre el número de lotes y la política de lotificación. Con la política, se calcula el tamaño de los lotes. Este enfoque no aborda el problema conjuntamente y la optimización del tamaño de los lotes es limitada [16].

Defersha y Chen. abordan ambos problemas, lotificación y programación de tareas en un entorno FJS. Presentan un modelo que incluye tiempos de setup dependientes de la secuencia y disponibilidad de las máquinas. Para solucionar el modelo hacen uso de un algoritmo genético paralelo en modelo de islas. Esta es una variante del algoritmo genético que divide la población en varias subpoblaciones, llamadas islas, y ejecuta un algoritmo genético en cada una de ellas de manera independiente. De vez en cuando, las islas intercambian individuos entre sí (proceso conocido como migración). La ventaja de esta variante es por un lado la mejora de la diversidad genética y por otro lado y fundamental, la ejecución paralela en múltiples procesadores o máquinas. En este enfoque el problema se aborda en una sola fase. La codificación de la solución se muestra en la Figura 8. El segmento izquierdo contiene valores que representan el tamaño de cada lote de cada trabajo (α toma valores entre 0 y 1). El segmento izquierdo representa la secuencia de operaciones (j = trabajo, s = lote, o = operación, m = máquina asignada) [17].

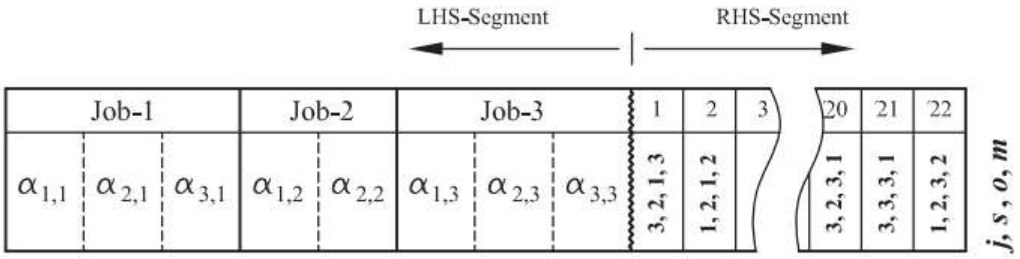

Figura 8. Representación de la solución. El cromosoma está formado por dos segmentos: LHS (Left Hand Side) y RHS (Right Hand Side). [Fuente: [17]].

Con la misma estrategia de codificación de la solución, Defersha y Bayat Movahed desarrollan un algoritmo genético asistido con programación lineal para solucionar el mismo problema con un solo recurso computacional, es decir, ya no hacen uso del modelo de islas. Las variables enteras de determinan mediante el algoritmo genético mientras que las variables continuas (tamaño de los lotes) se optimizan con un modelo de programación lineal (LP) extraído de un MILP en el que se omiten las restricciones con variables enteras [4].

Defersha y Rooyani desarrollan un algoritmo genético de dos etapas. La representación de las soluciones cambia ligeramente, en concreto el segmento RHS es diferente en cada etapa. Este planteamiento permite al algoritmo encontrar soluciones muy buenas desde el principio y converger rápidamente a regiones prometedoras del espacio de búsqueda. La segunda etapa elimina la naturaleza codiciosa de la primera siguiendo el enfoque habitual de un algoritmo genético para FJSP e intenta mejorar las soluciones encontradas en la primera etapa [2].

Fan et al. indican que la programación matemática y los métodos metaheurísticos se vuelven menos eficaces cuando un problema de programación contiene atributos de optimización tanto discretos como continuos. Por ello, una mateheurística que combina las ventajas de ambas metodologías, se considera una solución prometedora. Este trabajo investiga un problema de lot streaming en entorno FJS y reconfiguración de máquinas (FJSP-LSMR) para la minimización del retraso total ponderado. En primer lugar, se establece un modelo de programación lineal entera mixta (MILP) para el FJSP-LSMR. A continuación, se aplica un método mateheurístico con un componente de búsqueda de vecindad variable (MH-VNS) para abordar el problema. El MH-VNS adopta el algoritmo genético clásico (AG) como marco de trabajo, e introduce dos estrategias de optimización del flujo de lotes basadas en MILP, LSO1 y LSO2, para mejorar los planes de dimensionamiento de lotes con diversos grados [18].

2.3 Optimización Multiobjetivo en Job Shop Scheduling

2.3.1 Introducción

La optimización multiobjetivo consiste en resolver problemas que tienen dos o más funciones objetivo que deben optimizarse simultáneamente. En los problemas JSP, estos objetivos pueden entrar en conflicto al ser minimizados. Por ejemplo, minimizar el makespan y al mismo tiempo la tardanza, la transferencia de lotes (en el caso de considerar lot streaming) y el consumo de energía. Por lo tanto, el JSP puede formularse como un problema de optimización multiobjetivo, en el que el óptimo no es una solución única, sino un conjunto de soluciones de compromiso que presentan diferentes equilibrios entre los objetivos establecidos (conocido como frente de Pareto). En la Figura 9 se muestra un frente de Pareto para la optimización multiobjetivo del makespan y el consumo de energía en un job shop.

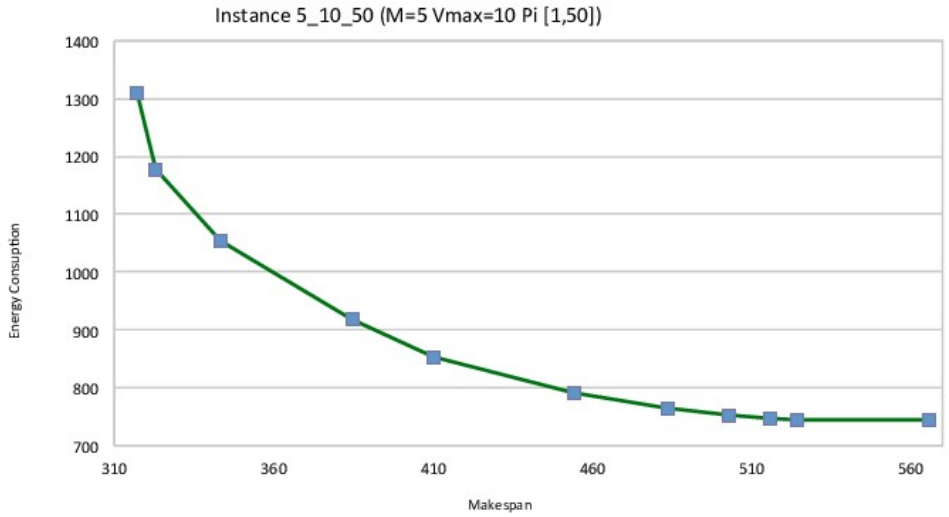

Figura 9. Frente de Pareto para los objetivos makespan y consumo de energía en un job shop. [Fuente: [19]]

La creciente importancia de la eficiencia energética en la industria ha resultado en la inclusión de objetivos relacionados con la energía en los problemas JSP. A raíz de querer minimizar el consumo de los costes energéticos y encontrar un balance entre el consumo energético y otros indicadores de rendimiento, se han estudiado y desarrollado múltiples problemas JSP (y derivados del JSP) que tienen en cuenta el rendimiento energético [20].

La inclusión de algunas restricciones para adaptar mecanismos heurísticos a diferentes escenarios o que simplemente son difíciles de modelar, puede ser también objeto de la optimización multiobjetivo. Por ejemplo, la inclusión de restricciones temporales y/o en el tamaño de lotes de producción. Con este enfoque, se pueden considerar esas restricciones y definir, de alguna manera, el grado de cumplimiento deseado de las mismas.

2.3.2 Evaluación, algoritmos y toma de decisiones.

Existen varios métodos para evaluar y tomar decisiones en problemas que involucran múltiples objetivos conflictivos. A continuación, se resumen los enfoques principales **asumiendo que se quieren minimizar todos los objetivos**:

1. **Curva o Frente de Pareto.** Consiste en construir una curva que representa el conjunto de soluciones que son Pareto óptimas, es decir, que no pueden ser mejoradas en un objetivo sin que al menos otro objetivo se vea afectado negativamente. Es útil para visualizar el equilibrio entre los objetivos e identificar el mejor trade-off posible. Este es el enfoque en el que se basan la mayoría de los métodos metaheurísticos [20].
2. **Suma ponderada.** Consiste en convertir el problema multiobjetivo en uno de un solo objetivo

combinando los diferentes objetivos en una función lineal ponderada. A cada objetivo se le asigna un peso según su importancia relativa y se suman para formar una única función objetivo.

En escenarios reales de programación de la producción, los decisores seguramente tengan un conjunto de pesos ya definidos, basándose en la experiencia [2].

3. **Suma escalada.** Implica normalizar los objetivos para que estén en una escala comparable o similar antes de combinarlos en una suma ponderada. Elimina la barrera que imponen las unidades de las métricas de los objetivos seleccionados.
4. **Programación por metas.** Se establecen objetivos específicos para cada criterio y se buscan soluciones que minimicen las desviaciones de estas metas. Se penalizan las desviaciones de las metas deseadas.
5. **Programación por compromisos.** Minimiza una función de distancia entre la solución actual y una solución ideal. La solución ideal presenta buenos compromisos o equilibrios entre los diferentes objetivos.

En la resolución de problemas multiobjetivo con métodos exactos, generar un conjunto de soluciones óptimas pertenecientes al frente de Pareto es muy costoso desde el punto de vista computacional. Este coste crece según crece el tamaño de la instancia. Es por ello por lo que resulta prohibitivo utilizar un método basado en la eficiencia de Pareto.

Por otro lado, en la resolución de problemas multiobjetivo con métodos heurísticos, se usan principalmente métodos basados en Pareto. Los algoritmos más populares son:

- **Non Dominated Sorting Genetic Algorithm II (NSGA-II).** Es un Algoritmo Genético que clasifica a la población en diferentes curvas basadas en la dominancia de Pareto, tal y como se ilustra en la Figura 10. Se dice que una solución domina a otra si es mejor en al menos un objetivo y no es peor en ningún otro objetivo.

  Para asegurar que la población es rica en material genético (hay cromosomas diferentes), se usa una métrica que mide como de cerca están las soluciones en el espacio, con el objetivo de fomentar soluciones que están bien distribuidas a lo largo de la curva de Pareto. La selección de cromosomas para el posterior cruce y/o mutación se lleva a cabo teniendo en cuenta el ranking y la distancia calculada anteriormente.

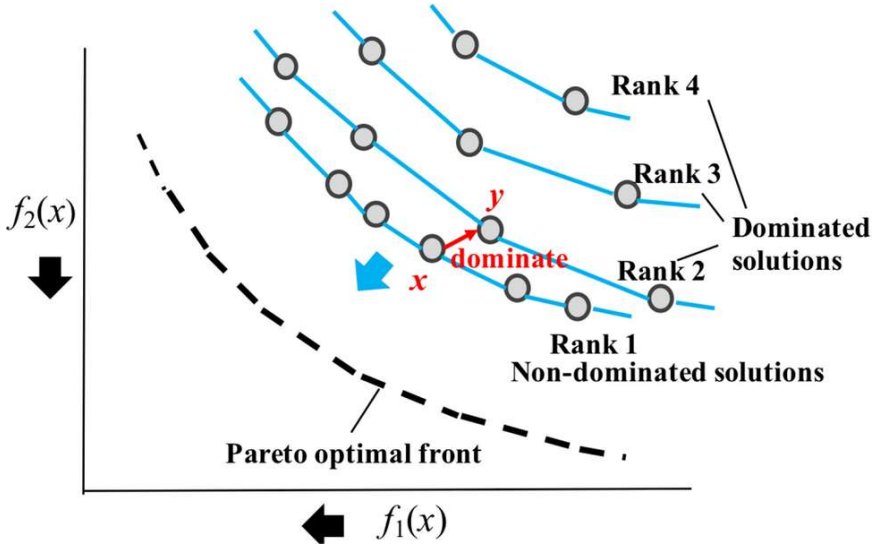

Figura 10. Clasificación de soluciones según dominancia. [Fuente: [21]]

- **Strength Pareto Evolutionary Algorithm 2 (SPEA-2).** Este algoritmo usa el concepto de la "fuerza" para evaluar las soluciones. Se calculan cuantas soluciones son dominadas por una solución particular. A cada solución se le asigna un fitness, un valor que tienen en cuenta no solo como de buenas son, sino como de bien distribuidas en el espacio están.

  Usa el mismo mecanismo evolutivo que el Algoritmo Genético para generar soluciones nuevas y moverse por el espacio de soluciones. Sin embargo, se diferencia en el cálculo del fitness y en el uso de un "archivo" externo con soluciones no dominadas y vecinos de estas. Este archivo sirve para guardar las mejores soluciones (no dominadas) y guiar al algoritmo en la búsqueda de nuevas soluciones. Es una forma de implementar elitismo.

- **Multi-Objective Evolutionary Algorithm based on Decomposition (MOEA/D).** A diferencia del NSGA-II y el SPEA-2, esta metaheurística presenta un enfoque totalmente distinto al descomponer el problema multiobjetivo en múltiples problemas de optimización unidimensionales, utilizando vectores con pesos asociados a los diferentes objetivos para representar diferentes trade-offs entre los objetivos, y optimizando estos problemas simultáneamente mediante un método basado en la vecindad, que ayuda a explorar el conjunto de soluciones que son parte del frente de Pareto.

Cabría esperar encontrar un gran número de artículos que resuelven problemas JSP con estos algoritmos evolutivos, sin embargo, la mayoría, ofrecen nuevas técnicas de optimización que intentan mejorar el rendimiento de los presentados aquí. Para et al. indican que el 88.5 % de los artículos revisados desarrollan nuevos algoritmos o mejoras de los ya existentes, mientras que solo el 9.8 % usan algoritmos ya existentes. No obstante, estos algoritmos si son los más usados para comparar resultados [20].

En la práctica, en la mayoría de los casos reales, la decisión final la toma el responsable de producción basándose en la experiencia y en los objetivos estratégicos de la organización. Mediante métodos heurísticos, se buscan conjuntos relativamente pequeños de soluciones que pertenecen al frente de Pareto, para no complicar la toma de decisiones.

En general, en la programación de job shops se suelen considerar pocos objetivos. Así lo reflejan los artículos científicos. Según Para et al. el 75.44 % de los artículos revisados consideran solo 2 objetivos.

2.4 Conclusiones de la revisión de la literatura

El Lot Streaming Job Shop Scheduling Problem (LSJSP) es una variante compleja del problema clásico de programación de la producción en entornos tipo taller. Esta variante introduce el concepto de dividir los trabajos en lotes más pequeños, con el objetivo de mejorar la eficiencia productiva. El LSJSP aborda tres decisiones clave: determinar el número óptimo de sublotes para cada trabajo, establecer el tamaño de cada sublote, y programar las operaciones de todos los sublotes en las máquinas correspondientes.

La lotificación ofrece varias ventajas significativas, como la reducción del makespan, la disminución del inventario de trabajo en proceso (WIP), la reducción del tiempo de flujo, una mejor utilización de las máquinas y un aumento de la flexibilidad productiva. Sin embargo, también presenta desventajas que deben ser consideradas, como el aumento en la frecuencia de los setups, mayores costes de transporte, una mayor complejidad en el control de la producción, potenciales problemas de calidad y un incremento en el coste computacional de la optimización.

El LSJSP está clasificado como un problema NP-hard, lo que implica que los métodos exactos de resolución son computacionalmente costosos para instancias de tamaño medio o grande. Por esta razón, se utilizan principalmente técnicas metaheurísticas para su resolución, siendo el Algoritmo Genético (GA) una de las más populares. Los enfoques de resolución se dividen principalmente en métodos de una fase integrada, que resuelven simultáneamente la lotificación y la programación, y métodos de dos fases, que abordan estos aspectos de forma secuencial.

Las tendencias actuales en la investigación del LSJSP apuntan hacia el uso de metaheurísticas avanzadas y técnicas híbridas (mateheurísticas), la consideración de múltiples objetivos incluyendo la eficiencia energética, y el desarrollo de algoritmos adaptados a problemas con restricciones y objetivos específicos. En el campo de la

optimización multiobjetivo, los algoritmos más populares son NSGA-II, SPEA-2 y MOEA/D, aunque la mayoría de los artículos científicos proponen nuevas técnicas o mejoras de las existentes.

En la práctica industrial, la aplicación de estas técnicas de optimización avanzadas aún presenta desafíos. La toma de decisiones final suele recaer en el responsable de producción, quien se basa en su experiencia y en los objetivos estratégicos de la organización. Esto subraya la importancia de desarrollar métodos que no solo sean teóricamente sólidos, sino también prácticos y comprensibles para los profesionales de la industria.

En conclusión, el LSJSP representa un área de investigación compleja y en crecimiento, con un gran potencial de aplicación en la industria moderna. La tendencia hacia la producción en lotes pequeños y la necesidad de equilibrar múltiples objetivos, incluyendo la eficiencia energética, están impulsando el desarrollo de nuevos métodos de optimización. Sin embargo, todavía existe un desafío importante en la traducción de estos avances teóricos a aplicaciones prácticas en entornos industriales reales. El futuro de la investigación en este campo probablemente se centrará en cerrar esta brecha entre la teoría y la práctica, desarrollando algoritmos que sean tanto potentes como accesibles para su implementación en escenarios de producción reales.

3 MODELADO DEL PROBLEMA

En el presente capítulo se ofrecen modelos MILP de los 4 escenarios que posteriormente se resuelven con el algoritmo genético. En el apartado 3.1 se detallan las hipótesis que se han hecho. Con estas consideraciones se facilita información acerca del entorno de producción y se desgranan las limitaciones con las que se modela el problema.

Por último, en el apartado 3.2 se ofrece el modelado de cada uno de los escenarios. Para ello, se proporciona una notación común a todos los modelos y posteriormente se presentan y se explican las restricciones de cada modelo.

3.1 Hipótesis y función objetivo

En el presente trabajo se trata el problema con lotes de tamaño consistente y permitiendo holguras. Es decir, mismo tamaño de sublotes para un mismo trabajo y se pueden procesar sublotes de un trabajo distinto entre 2 sublotes del mismo trabajo. Las rutas de los trabajos son parciales. Esto significa que un trabajo puede tener una ruta que no contiene todas las máquinas.

A continuación se ofrecen 4 modelos con características diferentes en cuanto a tiempos de setup y restricciones temporales (ventanas temporales o turnos). En la Tabla 1 se resumen las características.

Modelo	**Tiempos de Setup**	**Restricción de turnos**
LSJSP	Independientes de la secuencia	No
LSJSP con turnos	Independientes de la secuencia	Si
LSJSP Setup dependiente	Dependientes de la secuencia	No
LSJSP Setup dependiente con turnos	Dependientes de la secuencia	Si

Tabla 1. Modelos matemáticos desarrollados para el LSJSP.

Se consideran las siguientes hipótesis en el modelado del problema:

- Cada trabajo *j* con una ruta parcial asignada R_j se considera un lote.
- Todos los lotes están disponibles en el instante inicial.
- Todas las máquinas están disponibles en el instante inicial.
- Se permiten holguras entre sublotes de diferentes trabajos (intermingling*)*.
- No se permiten interrupciones (preemption*)* en el procesado de sublotes. Es decir, una vez que un lote ha empezado a procesarse debe terminarse sin interrupciones.
- Los tiempos de transferencia de los sublotes entre máquinas se consideran despreciables e iguales a 0.
- Las máquinas no pueden procesar más de un sublote a la vez.
- El *buffer* para el trabajo en proceso es ilimitado. Es decir, se considera que el sistema de producción tiene capacidad infinita.
- No se consideran inventarios iniciales, finales o entre estaciones.
- No se consideran fallos en las máquinas ni paradas por mantenimiento.
- Los tiempos de setup no son anticipatorios. Esto significa que no se puede realizar el setup antes de que se libere el sublote correspondiente.

Respecto a la métrica de rendimiento. La función objetivo considerada es minimizar el makespan, es decir, el tiempo comprendido entre el inicio de la primera operación y la finalización de la última operación.

3.2 Modelado MILP

El modelado del problema se basa en el modelo propuesto por Manne [22]. Es conocido como modelo disyuntivo, ya que se basa en el grafo disyuntivo. En el Trabajo Fin de Máster previamente realizado [1] se explica en detalle este enfoque. Se ha probado que es el modelo más eficiente (mejora más rápido) para instancias pequeñas en entornos tipo Job Shop [23].

3.2.1 Notación

Naturaleza	**Notación**	**Definición**
Constantes y Parámetros	J	Número de trabajos o productos.
	M	Número de máquinas
	U	Número de sublotes máximo.
	O	Número de turnos máximo

	d_j	Demanda de cada trabajo j
	p_{mj}	Tiempo de procesado de cada unidad del trabajo j en la máquina m
	R_j	Ruta del trabajo j
	SHT	Duración de un turno
	s_{mjk}	Setup en la máquina m si un lote del trabajo j precede directamente a un lote del trabajo k

Variables continuas	C	Makespan
	x_{mju}	Instante en el que el lote u del trabajo j empieza a procesarse en la máquina m
	y_{mju}	Instante en el que el setup asociado al lote u del trabajo j empieza a procesarse en la máquina m
	p_{mju}	Tiempo de procesado del lote u del trabajo j empieza a procesarse en la máquina m
	c_{mju}	Makespan del lote u del trabajo j en la máquina m

Variables enteras	q_{ju}	Tamaño del sublote u del trabajo j
binaria	$z_{mkl ju}$	Variable binaria que vale 1 si el lote l del trabajo k precede al lote u del trabajo j en la máquina m
	$z_{mkl ju}$	Variable binaria que vale 1 si el lote l del trabajo k precede INMEDIATAMENTE al lote u del trabajo j en la máquina m
binaria	Q_{ju}	Variable binaria que vale 1 si el tamaño del lote q_{ju} es mayor que 0
binaria	$W_{mo ju}$	Variable binaria que vale 1 si el lote ju para por la máquina m en el turno o

Usadas para la modelización de turnos

Usadas para la inclusión de tiempos de setup dependientes de la secuencia

3.2.2 LSJSP

A continuación, se presenta el modelo de programación lineal entera mixta para el caso más sencillo. Consta de 14 expresiones: 1 función objetivo y 13 restricciones, incluyendo las restricciones que determinan la naturaleza de las variables.

Objetivo

$$\min C \tag{1}$$

Restricciones

$$y_{R^h ju} \geq x_{R^{h-1} j,j,u} + p_{R^{h-1} j,j,u} \qquad \forall\, h \in (2,\ldots,M)\ in\ R_j;\ \forall\, j \in J;\ \forall\, u \in U \tag{2}$$

$$y_{mju} \geq x_{mj,u-1} + p_{mj,u-1} \qquad \forall\, m \in (M \wedge R_j);\ \forall\, j \in J;\ \forall\, u \in U\backslash\{0\} \tag{3}$$

$$z_{mklju} + z_{mjukl} = 1 \qquad \forall\, m \in \left(M \wedge R_j \wedge R_k\right); \forall j,k \in J; \forall u,l \in U \quad j \neq k \vee u \neq l \tag{4}$$

$$y_{mju} + V \cdot \left(1 - z_{mklju}\right) \geq (x_{mkl} + p_{mkl}) \qquad \forall\, m \in \left(M \wedge R_j \wedge R_k\right); \forall j,k \in J; \forall u,l \in U \quad j \neq k \vee u \neq l \tag{5}$$

$$x_{mju} \geq y_{mju} + s_{mj} \cdot Q_{ju} \qquad \forall\, m \in (M \wedge R_j);\ \forall\, j \in J;\ \forall\, u \in U \tag{6}$$

$$C \geq x_{mju} + p_{mju} \qquad \forall\, m \in M;\ \forall\, j \in J;\ \forall\, u \in U \tag{7}$$

$$\sum_{u}^{U} q_{ju} = d_j \qquad \forall\, j \in J \tag{8}$$

$$q_{ju} \leq V \cdot Q_{ju} \qquad \forall\, j \in J;\ \forall\, u \in U \tag{9}$$

$$Q_{ju} \leq q_{ju} \qquad \forall\, j \in J;\ \forall\, u \in U \tag{10}$$

$$p_{mju} = p_{mj} \cdot q_{ju} \qquad \forall\, m \in M;\ \forall\, j \in J;\ \forall\, u \in U \tag{11}$$

$$x_{mju} \geq 0 \qquad \forall\, m \in M;\ \forall\, j \in J;\ \forall\, u \in U \tag{12}$$

$$y_{mju} \geq 0 \qquad \forall\, m \in M;\ \forall\, j \in J;\ \forall\, u \in U \tag{13}$$

$$z_{mklju} \in \{0,1\} \qquad \forall\, m \in M;\ \forall\, k,j \in J, k \neq j \tag{14}$$

$$Q_{ju} \in \{0,1\} \qquad \forall\, j \in J;\ \forall\, u \in U \tag{15}$$

La restricción (2) describe la ruta de trabajo de cada producto j. Se impone que el *setup* asociado a un sublote no puede comenzar en la siguiente máquina hasta que no haya terminado de procesarse en la máquina anterior.

La restricción (3) indica que en cada máquina, los sublotes de cada producto j deben procesarse en orden. El sublote número $x - 1$ debe preceder al lote número x.

Las restricciones (4) y (5) describen la precedencia entre sublotes en cada máquina. En (4) se indica que un

sublote siempre precede a otro sublote en la misma máquina, siempre que sean sublotes con igual número pero asociados a distintos trabajos o sublotes con distinto número pero asociados al mismo trabajo. En ningún caso ambos sublotes pueden estar asociados al mismo trabajo y tener el mismo índice, pues se estaría imponiendo que el propio sublote se precede a sí mismo, lo cual lógicamente no es posible. En (5) se especifica que, en cada máquina, el *setup* del sublote sucesor tiene que empezar después de que acabe de procesarse el sublote predecesor.

La restricción (6) describe la ejecución del *setup* para el procesado de cada sublote. Si el tamaño del sublote no es 0, entonces se ejecuta el *setup* y el procesado comenzará cuando este termine. En otras palabras, todos los sublotes tienen que empezar a procesarse cuando haya acabado su *setup* siempre y cuando el tamaño del lote no sea 0.

La restricción (7) describe el *makespan.* Simplemente sirve para calcular el valor de la función objetivo y por ende establecer la dirección de mejora.

La restricción (8) indica que todas las unidades de un lote o producto han de ser repartidas en sublotes. Por lo tanto, la suma del tamaño de los sublotes ha de ser el número total de unidades que componen la orden de trabajo del producto.

Las restricciones (9) y (10) son fundamentales para la ejecución de los *setup.* Son complementarias a la restricción (6). Mediante la variable aleatoria Q_{ju} se indica si el sublote u asociado al producto j va a contener alguna unidad.

Al modelo se le indica el número máximo de sublotes que se quieren considerar, sin embargo puede que lo óptimo no sea considerar ese número de sublotes sino un número menor. Por ejemplo, se podría considerar una división máxima en 10 sublotes pero que para un producto lo óptimo fuese dividir el lote en 3 sublotes y para otro producto dividirlo en 5 sublotes.

La restricción (11) describe el tiempo de proceso de cada sublote en cada máquina, teniendo en cuenta el tamaño de este y el tiempo de proceso unitario del producto en la correspondiente máquina. En el momento en el que el tamaño de los sublotes es consistente, esta restricción es necesaria, pues cada sublote tendrá un tiempo de procesamiento diferente, dependiendo del tamaño del sublote, la máquina y el producto.

Las restricciones (12) a (15) describen la naturaleza de las variables. Las restricciones (12) y (13) indican que los instantes temporales iniciales de la ejecución de los *setups* y del procesado de los sublotes tienen que ser positivos. Por ende, el *makespan* será siempre positivo, por cómo está definido en (7). Las restricciones (14) y (15) indican la naturaleza binaria de las variables z y Q.

3.2.3 LSJSP con turnos

Este modelo incluye la restricción de turnos. Todos los lotes deben comenzar y acabar su procesado dentro de un turno (ventana temporal). Este modelo incluye 18 restricciones.

Objetivo

$$\min C \tag{16}$$

Restricciones

$$y_{R^h ju} \geq x_{R^{h-1} j,j,u} + p_{R^{h-1} j,j,u} \qquad \forall\, h \in (2,\ldots,M)\ in\ R_j;\ \forall\, j \in J;\ \forall\, u \in U \tag{17}$$

$$y_{mju} \geq x_{mj,u-1} + p_{mj,u-1} \qquad \forall\, m \in (M \wedge R_j);\ \forall\, j \in J;\ \forall\, u \in U \backslash \{0\} \tag{18}$$

$$z_{mklju} + z_{mjukl} = 1 \qquad \forall\, m \in \left(M \wedge R_j \wedge R_k\right); \forall j,k \in J; \forall u,l \in U \quad j \neq k \vee u \neq l \tag{19}$$

$$y_{mju} + V \cdot \left(1 - z_{mklju}\right) \geq \left(x_{mkl} + p_{mkl}\right) \qquad \forall\, m \in \left(M \wedge R_j \wedge R_k\right); \forall j,k \in J; \forall u,l \in U \quad j \neq k \vee u \neq l \tag{20}$$

$$x_{mju} \geq y_{mju} + s_{mj} \cdot Q_{ju} \qquad \forall\, m \in (M \wedge R_j);\ \forall\, j \in J;\ \forall\, u \in U \tag{21}$$

$$C \geq x_{mju} + p_{mju} \qquad \forall\, m \in M;\ \forall\, j \in J;\ \forall\, u \in U \tag{22}$$

$$\sum_{u}^{U} q_{ju} = d_j \qquad \forall\, j \in J \tag{23}$$

$$q_{ju} \leq V \cdot Q_{ju} \qquad \forall\, j \in J;\ \forall\, u \in U \tag{24}$$

$$Q_{ju} \leq q_{ju} \qquad \forall\, j \in J;\ \forall\, u \in U \tag{25}$$

$$p_{mju} = p_{mj} \cdot q_{ju} \qquad \forall\, m \in M;\ \forall\, j \in J;\ \forall\, u \in U \tag{26}$$

$$\sum_{j=0}^{J} \sum_{u=0}^{U} (s_{mj} + p_{mju}) \cdot W_{moju} \leq SHT \qquad \forall\, m \in M;\ \forall\, o \in O \tag{27}$$

$$\sum_{o=0}^{O} W_{moju} = Q_{ju} \qquad \forall\, m \in (M \wedge R_j);\ \forall\, j \in J;\ \forall\, u \in U \tag{28}$$

$$y_{mju} \geq W_{moju} \cdot o \cdot SHT \qquad \forall\, m \in M;\ \forall\, o \in O;\ \forall\, j \in J;\ \forall\, u \in U \tag{29}$$

$$\left(x_{mju} + p_{mju}\right) \cdot W_{moju} \leq (o+1) \cdot SHT \qquad \forall\, m \in M;\ \forall\, o \in O;\ \forall\, j \in J;\ \forall\, u \in U \tag{30}$$

$$x_{mju} \geq 0 \qquad \forall\, m \in M;\ \forall\, j \in J;\ \forall\, u \in U \tag{31}$$

$$y_{mju} \geq 0 \qquad \forall\, m \in M;\ \forall\, j \in J;\ \forall\, u \in U \tag{32}$$

$$z_{mklju} \in \{0,1\} \qquad \forall\, m \in M;\ \forall\, k,j \in J, k \neq j \tag{33}$$

$$Q_{ju} \in \{0,1\} \qquad \forall\, j \in J;\ \forall\, u \in U \tag{34}$$

El modelo es idéntico al anterior salvo por las restricciones 27 – 30.

La restricción (27) indica que el tiempo de procesado de los lotes que pasan por la máquina m en el turno o no puede ser mayor al tiempo determinado previamente de duración de un turno (SHT).

La restricción (28) obliga a asociar a cada lote que no está "vacío" al menos un turno en cada máquina m por la que tiene que pasar ese lote. Es decir, si a ese lote se le asocian unidades a procesar, se debe asegurar se asocia con al menos un turno en cada máquina que está en su ruta de proceso.

Las restricciones (29) y (30) indican la ventana temporal en la que se puede procesar el lote. Esto se hace limitando el instante de inicio del setup y el instante de finalización del procesado del lote en máquina perteneciente a su ruta de proceso.

3.2.4 LSJSP con tiempos de setup dependientes de la secuencia

En este MILP se incorporan restricciones adicionales (en total se tienen 19 restricciones) para tener en cuenta los tiempos de setup que dependen de la secuencia de las operaciones. Esta característica es crucial en entornos productivos donde el tiempo de preparación de las máquinas o recursos varía según el orden en que se procesen los trabajos. Este enfoque mejora la capacidad de adaptación del modelo a diferentes escenarios industriales, facilitando su aplicación en una amplia gama de sectores donde los tiempos de setup secuenciales son una variable determinante.

En el modelo LSJSP base, mediante la restricción (4) se conoce si un trabajo precede a otro, pero no de forma inmediata. Entre un lote predecesor y otro sucesor, en la misma máquina, pueden existir uno o más lotes que se procesan entre estos dos. A modo de ejemplo: en la Figura 11 se visualiza claramente esta característica. Tras resolver el modelo para una determinada instancia, se tiene $z_{11001} = 1$ y $z_{10110} = 0$, por lo tanto, se conoce que el lote 0 (L0) del trabajo 1 (P1) precede al lote 1 (L1) del trabajo 0 (P0), pero no se conoce si esta precedencia es inmediata. Uno o varios trabajos pueden procesarse entre estos dos.

Figura 11. Precedencias en el modelo LSJSP. Se conoce que P1-L0 precede a P0-L1 pero no si lal precedencia es inmediata. En este caso, otros lotes se procesan entre estos 2. [Fuente: elaboración propia].

Para superar esta limitación y disponer de información exacta sobre las precedencias de los trabajos, en este

modelo se introducen 2 trabajos "dummy" o ficticios para representar los nodos iniciales y finales.

Se modifican los siguientes conjuntos para reflejar la inclusión de los trabajos ficticios:

- $J = \{0, 1, \dots, j+1\}$ siendo $\{0, j+1\}$ trabajos ficticios cuya ruta de proceso incluye todas las máquinas.
- $U_j = \{U_0, U_1, \dots, U_{\{j+1\}}\}$ siendo $|U_0| = \left|U_{\{j+1\}}\right| = 1$ (un lote para cada trabajo ficticio en cada máquina).

Estos trabajos ficticios no se dividen en varios lotes, por lo tanto, solo se les asigna uno. En la Figura 12 (que complementa la explicación ofrecida más adelante de las restricciones (46) y (47)) se representa a modo de ejemplo el boceto de un programa de producción en el que se incluyen los trabajos ficticios. Son artificios para "obligar" a todos los trabajos reales, no ficticios, a ser precedidos y sucedidos por otros trabajos, permitiendo saber exactamente cual precede y cual sucede a cada uno. Para que los trabajos ficticios siempre sean los primeros y últimos en la secuencia de cada máquina, se impone que el trabajo ficticio 0 no pueda tener predecesores y que el trabajo ficticio $j+1$ no pueda tener sucesores.

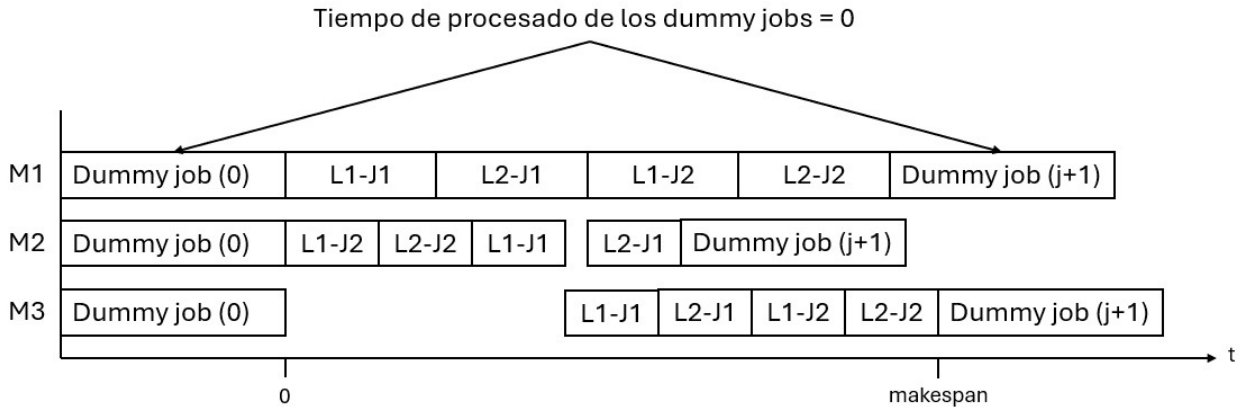

Figura 12. Representación de los dummy job en la secuencia de producción de un job shop con 3 máquinas y 2 trabajos, cada uno dividido en 2 lotes. [Fuente: elaboración propia].

Además se añade la variable c_{mju} para facilitar la inclusión de restricciones.

Objetivo

$$\min C \tag{35}$$

Restricciones

$$y_{R^h ju} \geq x_{R^{h-1} j,j,u} + p_{R^{h-1} j,j,u} \quad \forall\, h \in (2,\ldots,M)\ in\ R_j;\ \forall\, j \in J;\ \forall\, u \in U \tag{36}$$

$$y_{mju} \geq x_{mj,u-1} + p_{mj,u-1} \quad \forall\, m \in (M \wedge R_j);\ \forall\, j \in J;\ \forall\, u \in U \backslash \{0\} \tag{37}$$

$$y_{mju} \geq (x_{mkl} + p_{mkl}) - V * (1 - z_{mklju}) \quad \forall\, m \in (M \wedge R_j \wedge R_k);\ \forall j,k \in J;\ \forall u,l \in U;\ j \neq k \vee u \neq l \tag{38}$$

$$x_{mju} \geq x_{mkl} + p_{mkl} + s_{mjk} \cdot Q_{ju} - V * (1 - z_{mklju}) \quad \forall\, m \in (M \wedge R_j \wedge R_k);\ \forall j,k \in J;\ \forall u,l \in U;\ j \neq k \vee u \neq l \tag{39}$$

$$y_{mju} \geq x_{mju} - s_{mjk} \cdot Q_{ju} - V * (1 - z_{mklju}) \quad \forall\, m \in (M \wedge R_j \wedge R_k);\ \forall j,k \in J;\ \forall u,l \in U;\ j \neq k \vee u \neq l \tag{40}$$

$$x_{mju} \geq y_{mj,u} \quad \forall\, m \in (M \wedge R_j);\ \forall\, j \in J;\ \forall\, u \in U \tag{41}$$

$$x_{m,j+1,u} \geq c_{mj,u} \quad \forall\, m \in (M \wedge R_j);\ \forall\, j \in J, j \neq 1;\ \forall\, u \in U \tag{42}$$

$$x_{mju} \geq c_{m0u} \quad \forall\, m \in (M \wedge R_j);\ \forall\, j \in J, j \neq 0;\ \forall\, u \in U \tag{43}$$

$$C \geq c_{mju} \quad \forall\, m \in M;\ \forall\, j \in J;\ \forall\, u \in U \tag{44}$$

$$c_{mju} \geq x_{mju} + p_{mju} \quad \forall\, m \in M;\ \forall\, j \in J;\ \forall\, u \in U \tag{45}$$

$$\sum_{j \in J \backslash \{0\}} \sum_{u \in U} z_{mklju} = 1 \quad \forall\, m \in (M \wedge R_j \wedge R_k);\ \forall\, k \in J\{n+1\};\ \forall\, l \in U;\ j \neq k \vee u \neq l \tag{46}$$

$$\sum_{k \in J \backslash \{n+1\}} \sum_{l \in U} z_{mklju} = 1 \quad \forall\, m \in (M \wedge R_j \wedge R_k);\ \forall\, j \in J\{0\};\ \forall\, u \in U;\ j \neq k \vee u \neq l \tag{47}$$

$$\sum_{u}^{U} q_{ju} = d_j \quad \forall\, j \in J \tag{48}$$

$$q_{ju} \leq V \cdot Q_{ju} \quad \forall\, j \in J;\ \forall\, u \in U \tag{49}$$

$$Q_{ju} \leq q_{ju} \quad \forall\, j \in J;\ \forall\, u \in U \tag{50}$$

$$p_{mju} = p_{mj} * q_{ju} \quad \forall\, m \in M;\ \forall\, j \in J;\ \forall\, u \in U \tag{51}$$

$$x_{mju} \geq 0 \qquad \forall\, m \in M;\ \forall\, j \in J;\ \forall\, u \in U \qquad (52)$$

$$y_{mju} \geq 0 \qquad \forall\, m \in M;\ \forall\, j \in J;\ \forall\, u \in U \qquad (53)$$

$$z_{mklju} \in \{0,1\} \qquad \forall\, m \in M;\ \forall\, k,j \in J, k \neq j \qquad (54)$$

$$Q_{ju} \in \{0,1\} \qquad \forall\, j \in J;\ \forall\, u \in U \qquad (55)$$

Respecto al modelo LSJSP base, se elimina la restricción (4).

Se modifican las restricciones (6) a la que se le añade la nueva naturaleza de los tiempos de setup secuenciales y la restricción (7) que se descompone en dos restricciones (44), (45) para tener en cuenta la nueva variable incluida c_{mju}. Esto se hace principalmente para facilitar la compresión de los mecanismos del modelo, aunque no es necesario.

Se añaden las restricciones (39) a (47), marcadas a color en el modelo. Por razones de brevedad, solo se explican estas restricciones en detalle. El resto están explicadas en el apartado 3.2.2.

En la restricción (39) se consideran los tiempos de setup dependientes de la secuencia, alojados en la programación entre el makespan y el instante de inicio de 2 operaciones procesadas en la misma máquina, una después de otra. Facilita la determinación del instante de inicio del procesado después de completar el setup.

Las restricciones (40) y (41) sirven para determinar el instante de inicio del tiempo de setup de cada lote teniendo en cuenta cuál es el lote predecesor en cada máquina.

Con la restricción (42) se impone que el trabajo ficticio que representa el nodo final del grafo tiene que "empezar" cuando han acabado todos los anteriores. Es decir, su instante de inicio siempre será mayor o igual que el instante de finalización de todos los demás.

Con la restricción (43) se impone que todos los lotes tienen que procesarse después de que "acabe" el trabajo ficticio inicial. Se trata de un artificio similar al anterior para imponer que en todas las máquinas este trabajo sea el primero en la secuencia.

Como se ha comentado antes, las restricciones (44) y (45) son la descomposición de la restricción (7). Esta descomposición es fundamental para incluir las 2 restricciones anteriores.

Las restricciones (46) y (47) permiten tener un control absoluto de las precedencias entre lotes, sabiendo exactamente quien es el sucesor y quien es el predecesor de cada lote.

La restricción (46) especifica que todos los lotes, en cada máquina en la que se procesan, tienen un sucesor menos el trabajo ficticio final. Conjuntamente se impone que el trabajo ficticio inicial no puede ser un sucesor.

La restricción (47) especifica que todos los lotes, en cada máquina en la que se procesan, tiene un predecesor. El trabajo ficticio final no puede ser predecesor y el trabajo ficticio inicial no tiene predecesor.

3.2.5 LSJSP con tiempos de setup dependientes de la secuencia y turnos

En este modelo se simplifican las restricciones del modelo anterior, eliminando aquellas que no son necesarias. En concreto, se omiten las restricciones (41), (42) y (43). Las restricciones (44) y (45) se vuelven a componer en una sola restricción. Adicionalmente, se añaden restricciones de turnos, de forma que los lotes solo puedan ser procesados (inicializados y terminados) en determinadas ventanas temporales.

Este modelo está compuesto por 20 restricciones. Todas ellas han sido explicadas en los modelos ya presentados y por razones de brevedad no serán descritas de nuevo.

Objetivo

$$\min C \tag{56}$$

Restricciones

$$y_{R^h ju} \geq x_{R^{h-1} j,j,u} + p_{R^{h-1} j,j,u} \qquad \forall\, h \in (2,\dots,M)\ in\ R_j;\ \forall\, j \in J;\ \forall\, u \in U \tag{57}$$

$$y_{mju} \geq x_{mj,u-1} + p_{mj,u-1} \qquad \forall\, m \in (M \wedge R_j);\ \forall\, j\backslash\{0, n+1\} \in J;\ \forall\, u \in U\backslash\{0\} \tag{58}$$

$$y_{mju} \geq (x_{mkl} + p_{mkl}) - V \cdot (1 - z_{mklju}) \qquad \forall\, m \in (M \wedge R_j \wedge R_k);\ \forall j,k \in J;\ \forall u,l \in U;\ j \neq k \vee u \neq l \tag{59}$$

$$x_{mju} \geq x_{mkl} + p_{mkl} + s_{mjk} \cdot Q_{ju} - V \cdot (1 - z_{mklju}) \qquad \forall\, m \in (M \wedge R_j \wedge R_k);\ \forall j,k \in J;\ \forall u,l \in U;\ j \neq k \vee u \neq l \tag{60}$$

$$y_{mju} \geq x_{mju} - s_{mjk} \cdot Q_{ju} - V \cdot (1 - z_{mklju}) \qquad \forall\, m \in (M \wedge R_j \wedge R_k);\ \forall j,k \in J;\ \forall u,l \in U;\ j \neq k \vee u \neq l \tag{61}$$

$$C \geq x_{mju} + p_{mju} \qquad \forall\, m \in M;\ \forall\, j \in J;\ \forall\, u \in U \tag{62}$$

$$\sum_{j \in J\backslash\{0\}} \sum_{u \in U} z_{mklju} = 1 \qquad \forall\, m \in (M \wedge R_j \wedge R_k);\ \forall\, k \in J\{n+1\};\ \forall\, l \in U;\ j \neq k \vee u \neq l \tag{63}$$

$$\sum_{k \in J\backslash\{n+1\}} \sum_{l \in U} z_{mklju} = 1 \qquad \forall\, m \in (M \wedge R_j \wedge R_k);\ \forall\, j \in J\{0\};\ \forall\, u \in U;\ j \neq k \vee u \neq l \tag{64}$$

$$\sum_{j=0}^{J} \sum_{u=0}^{U} (x_{mj} + p_{mju} - y_{mju}) \cdot W_{moju} \leq SHT \qquad \forall\, m \in M;\ \forall\, o \in O \tag{65}$$

$$\sum_{o=0}^{O} W_{moju} = Q_{ju} \qquad \forall\, m \in (M \wedge R_j);\ \forall\, j \in J;\ \forall\, u \in U \tag{66}$$

$$y_{mju} \geq W_{moju} \cdot o \cdot SHT \qquad \forall\, m \in M;\ \forall\, o \in O;\ \forall\, j \in J;\ \forall\, u \in U \tag{67}$$

$$(x_{mju} + p_{mju}) \cdot W_{moju} \leq (o+1) \cdot SHT \qquad \forall\, m \in M;\ \forall\, o \in O;\ \forall\, j \in J;\ \forall\, u \in U \tag{68}$$

$$\sum_{u}^{U} q_{ju} = d_j \qquad \forall\, j \in J \tag{69}$$

$$q_{ju} \leq V \cdot Q_{ju} \qquad \forall\, j \in J;\ \forall\, u \in U \tag{70}$$

$$Q_{ju} \leq q_{ju} \qquad \forall\, j \in J;\ \forall\, u \in U \tag{71}$$

$$p_{mju} = p_{mj} \cdot q_{ju} \qquad \forall\, m \in M;\ \forall\, j \in J;\ \forall\, u \in U \tag{72}$$

$$x_{mju} \geq 0 \qquad \forall\, m \in M;\ \forall\, j \in J;\ \forall\, u \in U \qquad (73)$$

$$y_{mju} \geq 0 \qquad \forall\, m \in M;\ \forall\, j \in J;\ \forall\, u \in U \qquad (74)$$

$$z_{mklju} \in \{0,1\} \qquad \forall\, m \in M;\ \forall\, k,j \in J, k \neq j \qquad (75)$$

$$Q_{ju} \in \{0,1\} \qquad \forall\, j \in J;\ \forall\, u \in U \qquad (76)$$

4 DISEÑO DEL ALGORITMO GENÉTICO

En este capítulo se detalla el diseño de un algoritmo genético específico para resolver los 4 problemas o escenarios del LSJSP modelados en el capítulo anterior.

En el apartado 4.1 se habla del flujo de diseño del algoritmo. Esto es el proceso que se sigue para diseñar el algoritmo satisfactoriamente. De igual manera que una casa no se puede empezar a construir por el tejado, un algoritmo requiere definir una serie de parámetros y funcionalidades antes de ser implementado.

En el apartado 4.2 se ofrece la notación usada en el capítulo. En el apartado 4.3 se ofrece el diseño de la codificación del cromosoma. El apartado 4.4. trata la decodificación y evaluación del cromosoma, como base para el correcto funcionamiento de este tipo de algoritmos. En los apartados 4.5 a 4.8 se trata específicamente el diseño del algoritmo genético.

4.1 Flujo de diseño

Antes de detallar el flujo de proceso del diseño, cabe decir que el diseño y la implementación de un algoritmo son dos procesos que se pueden realizar conjuntamente. Hablando desde la experiencia, cuando se diseña una funcionalidad, es conveniente implementar el código correspondiente y hacer pruebas mínimas para validarla. Por ejemplo, cuando se implementa un módulo para la generación aleatoria de cromosomas, conviene comprobar que la estructura de datos generada y la información que contiene es correcta y coherente, antes de seguir diseñando e implementado otros módulos. Aunque en este capítulo solo se trate el proceso de diseño, es importante dejar claro que el proceso es complementario con el proceso de implementación.

A la hora de implementar una metaheurística es crucial desgranar cuáles son las funcionalidades necesarias (que

posteriormente se integrarán en la metaheurística) para:

- Codificar las posibles soluciones factibles.
- Evaluar las soluciones (restricciones y función objetivo).
- Implementar la estrategia de búsqueda.
- Implementar operadores de búsqueda (mutación y cruce en este caso).
- Implementar mecanismos de intensificación y diversificación.
- Representar e interpretar las soluciones decodificadas

En el diagrama de flujo mostrado en la Figura 13 se muestran el flujo de diseño del algoritmo genético en cuestión, en el cual se incluyen las funcionalidades que se han considerado necesarias.

Este flujo es apropiado para obtener un producto mínimo viable, un algoritmo capaz de resolver el problema con soluciones de suficiente calidad (en las que se observe una mejoría notable respecto a soluciones aleatorias) y que cumplen todas las restricciones. A partir de aquí se puede mejorar el algoritmo y estudiar en profundidad su comportamiento ante diferentes instancias de diferentes tamaños. Para ello se debe estudiar el efecto de los hiperparámetros del algoritmo (tamaño de la población, número de generaciones, probabilidad de uso de cada operador genético, etc) y la inclusión/omisión de operadores genéticos.

Los últimos pasos del diseño son la modificación del módulo de evaluación y decodificación. Más en concreto, se quieren implementar 2 parámetros que se puedan activar y desactivar para indicarle al algoritmo el tipo de decodificación deseado. Es decir, si queremos que tenga en cuenta las restricciones de turnos y si queremos que considere tiempos de setup secuenciales.

Se busca que la decodificación se lleve a cabo, según el valor de estos parámetros, para cualquiera de los 4 escenarios modelados en el capítulo anterior.

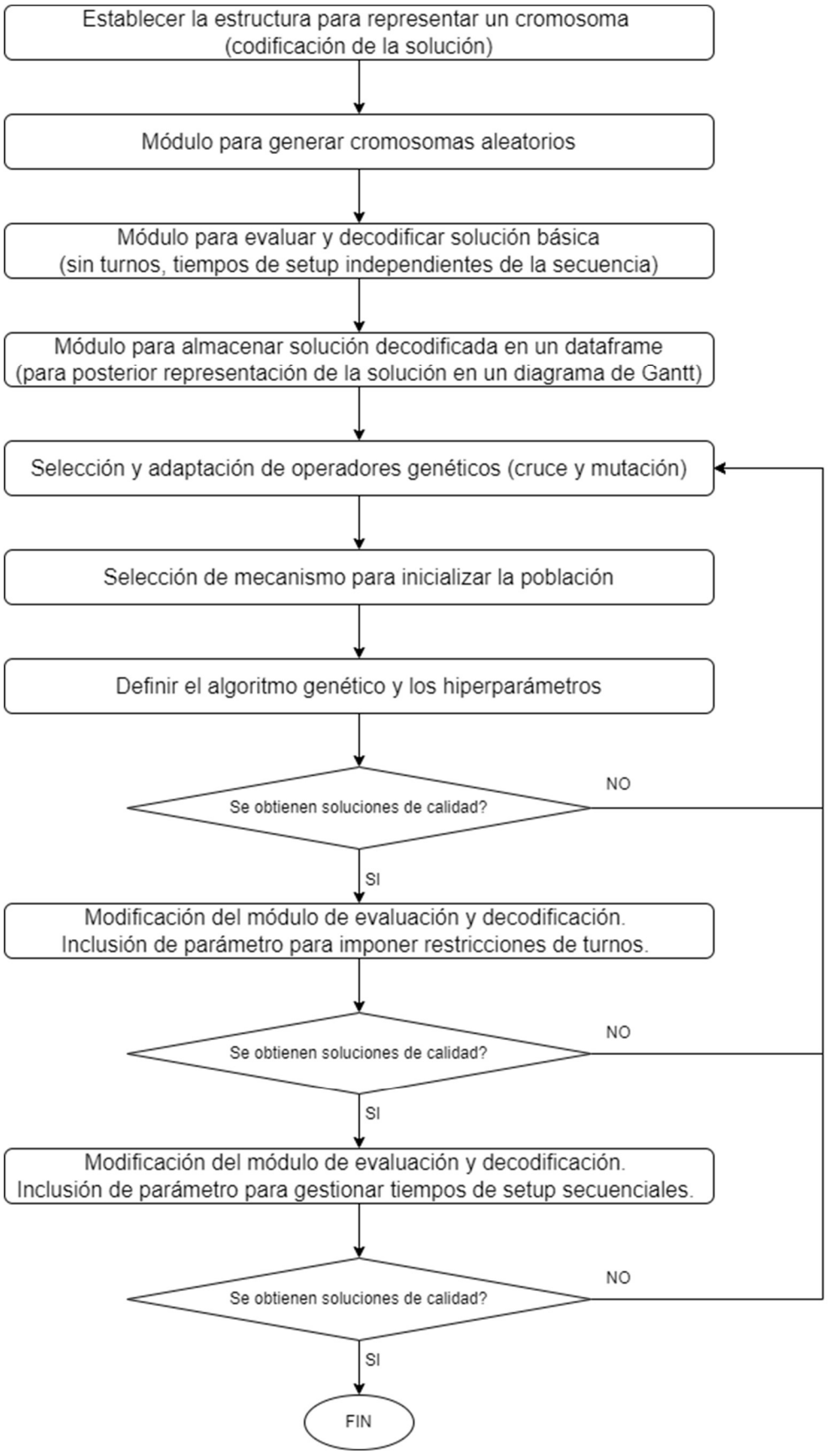

Figura 13. Flujo de diseño del algoritmo genético. [Fuente: elaboración propia]

A continuación se detalla el diseño de cada funcionalidad incluida en el diagrama de flujo anterior.

4.2 Notación

La notación usada para el diseño del algoritmo genético es la mostrada en la Tabla 2.

k	Índice de iteración (generación)
G_k	Población inicial aleatoria
G_s	Número de generaciones.
M_k	Población seleccionada (padres)
R_k	Población recombinada y mutada (hijos)
P_s	Tamaño de la población
T_s	Tamaño del torneo
C_p	Probabilidad de recombinación (Crossover)
M_p	Probabilidad de mutación
D_t	Umbral de diversificación
F_t	Contador de generaciones sin mejora
j	Trabajo o producto
m	Máquina
u	Lote
SHT	Duración de un turno
d_j	Demanda de cada trabajo
d_{ju}	Demanda asignada (tamaño) de cada lote
s_{mj}	Tiempo de setup independiente del trabajo j en la máquina m
s_{mjk}	Tiempo de setup dependiente del trabajo k precedido por el trabajo j en la máquina m
α_{ju}	Tamaño porcentual de cada lote (codificado)
$\alpha_{ju}{}'$	Tamaño en unidades de cada lote (decodificado)
RR_{ju}	Seguimiento de la ruta de cada lote
RHS	Segmento derecho del cromosoma
LHS	Segmento izquierdo del cromosoma
y_{mju}	Instante de inicio del setup de cada lote en cada máquina
x_{mju}	Instante de inicio de procesado de cada lote en cada máquina
c_{mju}	Instante de finalización de cada lote en cada máquina
δ	Coeficiente de escala de penalización

Tabla 2. Notación para el algoritmo genético.

4.3 Representación del cromosoma

La codificación de la solución para este problema no es para nada intuitiva, ya que son varias las variables que se pretenden optimizar de manera simultánea. Se debe determinar:

1. El tamaño de cada lote.
2. El número de lotes óptimo de cada trabajo o producto.
3. La secuencia de programación de los lotes.

La idea que se puede tener a priori es que es necesario tener un vector o matriz que contiene toda la información, pero cuyo tamaño es variable, ya que el número total de lotes a programar es a priori desconocido e irá variando según se desplace el algoritmo por el espacio de búsqueda.

No obstante, gestionar un array de tamaño variable en un algoritmo genético no es sencillo. Resultaría en casuísticas complejas cuando se aplicasen operadores de recombinación. Implementar elitismo y combinar soluciones de distinto tamaño también sería complejo.

Por otro lado, con el uso de un solo array se estaría mezclando información completamente diferente. Cada subproblema tiene una naturaleza matemática diferente. La secuencia de programación (3) viene dada por el orden en el que se indexan los elementos. El tamaño de cada lote (1) y el número de lotes de cada trabajo (2) se expresan con un valor numérico.

Por lo tanto, los operadores genéticos aplicados a cada subproblema deben ser completamente diferentes. Para el subproblema de secuenciación se buscan operadores que permuten elementos que componen un array. Para el subproblema de tamaño se buscan elementos que aumenten o disminuyan un valor contenido en un rango factible.

Atendiendo a estos impedimentos, se opta por representar el cromosoma con la misma estrategia que usan Defersha y Chen [17], adaptándola al caso del entorno job shop clásico.

El cromosoma está compuesto por 2 segmentos:

1. **LHS (*Left hand side* o segmento izquierdo).** En este segmento se representa el tamaño de los lotes de cada trabajo. Para cada trabajo j se tienen tantos valores α_{ju} como lotes máximos se hayan indicado. Cada α_{ju} toma un valor entre 0 y 1. Cada valor α_{ju}, junto con el resto valores α_{ju} asociados al mismo trabajo se normalizan para expresar el porcentaje de demanda asignado a cada lote.

   Por ejemplo, si se tienen 3 lotes de un trabajo j con los valores $\alpha_{j1} = 0{,}25$, $\alpha_{j2} = 1$ y $\alpha_{j3} = 0{,}75$ el

porcentaje de demanda asignada a cada lote vendría dado por:

$$porcentaje\ de\ asignación\ al\ lote\ (j,u) = \frac{\alpha_{ju}}{\sum_{u=1}^{U} \alpha_{ju}}$$

Se tendría la asignación del 12,5%, 50% y 37,5% de la demanda a los lotes 1, 2 y 3 respectivamente del trabajo j.

Conocida la demanda (cantidad a procesar) de cada trabajo, el tamaño real de cada lote vendrá dado por la siguiente expresión:

$$d_{ju} = \frac{\alpha_{ju}}{\sum_{u=1}^{U} \alpha_{ju}} \cdot d_j$$

Evidentemente, esta expresión puede arrojar números reales no enteros, por lo que se debe implementar una estrategia para obtener números enteros basados en esos valores. Esta estrategia se explica en detalle en el siguiente apartado ya que forma parte del algoritmo de decodificación del cromosoma.

En la Figura 14. Representación del segmento izquierdo del cromosoma (LHS) para una instancia con número máximo de lotes $U = 3$. $\alpha_{ju} \in [0,1]$.Figura 14 se ofrece la estructura de representación de este segmento del cromosoma.

Figura 14. Representación del segmento izquierdo del cromosoma (LHS) para una instancia con número máximo de lotes $U = 3$. $\alpha_{ju} \in [0,1]$. [Fuente: elaboración propia]

2. **RHS (*Right hand side* o segmento derecho).** En este segmento se representa la secuencia de operaciones. Cada gen del cromosoma es un par de índices trabajo – lote (j,u), representando un lote. Cada lote de cada trabajo estará en la lista tantas veces como máquinas haya en la ruta de proceso de ese trabajo.

   Por ejemplo, si la ruta de proceso del trabajo 1 fuese [3, 1, 2], el lote 1 del trabajo 1 ($j = 1$,$u = 1$) se incluiría 3 veces en el segmento, representando el número de operaciones que se realizan sobre ese lote (o lo que es igual, número de máquinas por las que pasa ese lote). Por lo tanto, cada gen representa una

operación.

Se puede entender este segmento del cromosoma como la lista de trabajos pendientes que, junto con el segmento izquierdo decodificado y la ruta de proceso de cada trabajo, se usa para hacer una programación semiactiva.

En la Figura 15 se muestra una representación del segmento derecho del cromosoma.

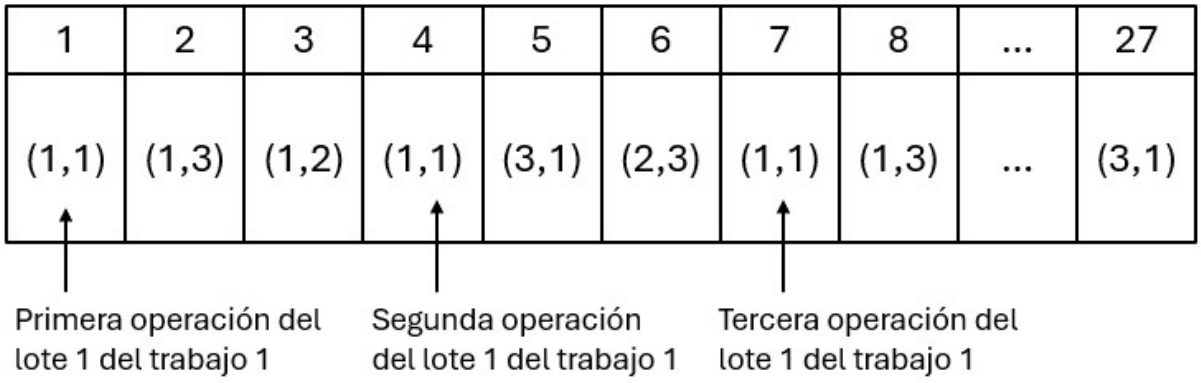

Figura 15. Representación del segmento derecho del cromosoma (RHS) para una instancia de 3 máquinas, 3 trabajos y 3 lotes. Todos los lotes pasan por todas las máquinas. [Fuente: elaboración propia]

Un vector formado por la concatenación de dos segmentos forma un cromosoma, tal y como se muestra en la

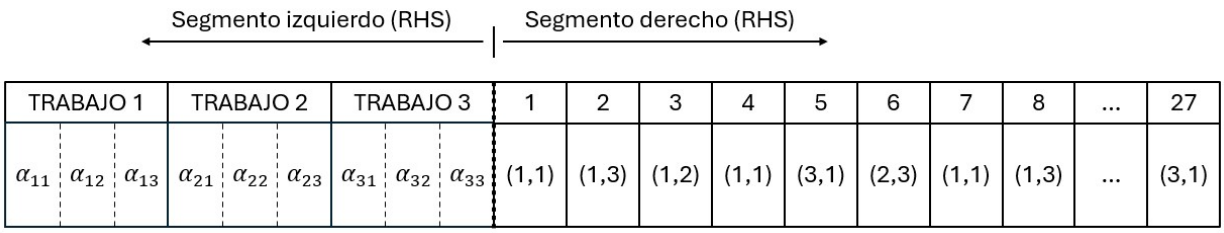

Figura 16. Representación de la solución (cromosoma) usada en el algoritmo genético. [Fuente: elaboración propia]

4.4 Decodificación y evaluación del cromosoma

Como ya se ha comentado, el problema LSJSP es un problema doble. Se deben resolver dos subproblemas de distinta índole de forma conjunta. Por ello, la representación del cromosoma se divide en dos segmentos. Con la decodificación del cromosoma sucede lo mismo. Primero se determina el tamaño real de cada lote (decodificación del segmento izquierdo) y luego se hace un programa semiactivo (decodificación del segmento derecho). El cromosoma se evalúa con los resultados del programa semiactivo, en concreto con el makespan y, en el caso de considerar turnos, con el makespan y una penalización que aplica cuando un lote ocupa un tiempo total superior a la duración de un turno.

4.4.1 Distribución de la demanda (LHS)

Se decodifica el segmento izquierdo para obtener el tamaño real (en unidades) de cada lote. A esto se le denomina distribución de la demanda. Consiste en la normalización de los valores α_{ju} de cada trabajo j y la asignación de un valor numérico entero a cada lote, que representa el número de unidades del trabajo j que lo componen.

Pseudocódigo para distribuir la demanda del cromosoma.

Para *cada trabajo* j

$suma\ tamaños = \sum_{u=1}^{U} \alpha_{ju}$

Si $suma\ tamaños \neq 0$

$normalizar\ \alpha_{ju}\ \forall\ U$

Para *cada lote* u

$\alpha'_{ju} = int(\alpha_{ju} \cdot d_j) = d_{ju}$

$suma\ inicial = \sum_{u=1}^{U} \alpha_{ju}'$

$residuo = suma\ inicial - d_j$

Si $residuo > 0$

Para *cada lote* u

$aumenta\ \alpha_{ju}'\ en\ 1\ unidad$

$disminuye\ residuo\ en\ 1\ unidad\ hasta\ que\ residuo = 0$

Sino

$Distribuye\ la\ demanda\ por\ igual\ entre\ todos\ los\ U - 1\ lotes$

$último\ lote \leftarrow d_j -$ *la suma del tamaño de los anteriores*

4.4.2 Programa semiactivo (RHS)

Conocido el tamaño de cada lote, se decodifica el segmento derecho dando lugar a un programa semiactivo de producción. Un programa factible se denomina **programa semiactivo** si ningún trabajo u operación puede ser finalizado con antelación sin cambiar el orden en el que se procesan en cualquiera de las máquinas [24]. Se obtienen los instantes de inicio y fin de procesado de cada lote, así como el makespan global.

Pseudocódigo para obtener un programa semiactivo <u>sin restricciones de turnos</u> (tiempos de inicio y finalización de cada lote).

---

1. $RR_{ju} \leftarrow R_j$ ***Para*** *cada trabajo* j, ***Para*** *cada lote* u

2. ***Para*** $cada(j,u)$ *en RHS*

3. ***Si*** $\alpha_{ju}' \neq 0$

4. $RR_{ju}[0] = m$

5. ***Si*** *m es la* ***primera máquina*** *de la ruta y está* ***vacía*** (0 *lotes procesados*)

6. $y_{mju} = 0$

7. ***Si*** *m es la* ***primera máquina*** *de la ruta y* ***NO*** *está* ***vacía*** (≥ 1 *lote procesados*)

8. $y_{mju} = tiempo\ de\ finalización\ del$ último *lote procesado en m*

9. ***Si*** *m* ***NO*** *es la* ***primera máquina*** *de la ruta y está* ***vacía*** (0 *lotes procesados*)

10. $y_{mju} = tiempo\ de\ finalización\ del\ lote\ (j,u)\ en\ máquina\ anterior$

11. ***Si*** *m* ***NO*** *es la* ***primera máquina*** *de la ruta y* ***NO*** *está* ***vacía*** (≥ 1 *lote procesados*)

12. $y_{mju} = max\ (tiempo\ de\ finalización\ del$ último *lote procesado en m,*

13. $tiempo\ de\ finalización\ del\ lote\ (j,u)\ en\ máquina\ anterior$)

14. $c_{mju} = y_{mju} + s_{mj} + p_{mj} \cdot \alpha_{ju}'$

15. $actualiza\ makespan\ global$

16. $Procesados_m \leftarrow (j,u)$

17. $Elimina\ el\ primer\ elemento\ de\ RR_{ju}$

Una vez se tienen todos los instantes de inicio de setup y finalización de cada lote, calcular el instante de inicio del procesado de cada lote es inmediato con la siguiente expresión:

$$x_{mju} = c_{mju} - p_{mj} \cdot \alpha_{ju}'$$

El pseudocódigo anterior es válido para hacer programaciones de acuerdo con el modelo base, sin restricciones de turnos y sin tiempos de setup dependientes de la secuencia.

La **implementación de las restricciones de turnos** requiere modificar ligeramente el algoritmo de

decodificación.

Pseudocódigo para obtener un programa semiactivo <u>con restricciones de turnos</u> (tiempos de inicio y finalización de cada lote).

$RR_{ju} \leftarrow R_j$ ***Para*** *cada trabajo* j, ***Para*** *cada lote* u

Para $cada(j,u)$ *en* RHS

Si $\alpha_{ju}{}' \neq 0$

$RR_{ju}[0] = m$

Si $s_{mj} + p_{mj} \cdot \alpha_{ju}{}' > SHT$

aplicar líneas 5 *a* 17 *del pseudocódigo* ***sin restricciones de turnos***

Sino

Si m *es la* ***primera máquina*** *de la ruta y está* ***vacía*** (0 *lotes procesados*)

$y_{mju} = 0$

Si m *es la* ***primera máquina*** *de la ruta y* ***NO*** *está* ***vacía*** ($\geq$ 1 *lote procesados*)

$c^* = tiempo\ de\ finalización\ del$ último *lote procesado en* m

Si *lote cabe en turno del lote predecesor en* m

$y_{mju} = c^*$

Sino

$Último\ turno = c^*//SHT$

$y_{mju} = SHT \cdot (último\ turno + 1)$

Si m ***NO*** *es la* ***primera máquina*** *de la ruta y está* ***vacía*** (0 *lotes procesados*)

$c^* = tiempo\ de\ finalización\ en\ máquina\ anterior\ en\ R_j$

Si *lote cabe en turno de proceso en máquina anterior*

$y_{mju} = c^*$

$\boldsymbol{Sino}\ y_{mju} = SHT \cdot (c^{*}//SHT + 1)$

$\boldsymbol{Si}\ m\ \boldsymbol{NO}\ es\ la\ \boldsymbol{primera\ máquina}\ de\ la\ ruta\ y\ \boldsymbol{NO}\ está\ \boldsymbol{vacía}\ (\geq 1\ lote\ proc.)$

$c_1^{*} =\ tiempo\ de\ finalización\ del\ último\ lote\ procesado\ en\ m$

$c_2^{*} =\ tiempo\ de\ finalización\ en\ máquina\ anterior\ en\ R_j$

$\boldsymbol{Si}\ c_2^{*} \geq c_1^{*}$

$\boldsymbol{Si}\ \ lote\ cabe\ en\ turno\ de\ proceso\ en\ máquina\ anterior$

$y_{mju} = c_2^{*}$

$\boldsymbol{Sino}$

$y_{mju} = SHT \cdot (c_2^{*}//SHT + 1)$

$\boldsymbol{Sino}$

$\boldsymbol{Si}$ lote cabe en turno del lote predecesor

$y_{mju} = c_1^{*}$

$\boldsymbol{Sino}\ y_{mju} = SHT \cdot (c_1^{*}//SHT + 1)$

$c_{mju} = y_{mju} + s_{mj} + p_{mj} \cdot LHS(j,u)$

$actualiza\ makespan\ global$

$Procesados_m \leftarrow (j,u)$

$Elimina\ el\ primer\ elemento\ de\ RR_{ju}$

Para implementar **tiempos de setup dependientes de la secuencia**, basta con cambiar s_{mj} por s_{mjk}, siendo este último el tiempo de setup en la máquina m cuando el trabajo j precede al trabajo k. Las precedencias en cada máquina se registran a través del vector $Procesados_m$. En los datos del problema se incluye el tiempo de setup $s_{m,-1,k}$ para representar el tiempo de setup de los trabajos que no son precedidos. Son aquellos trabajos que ocupan la primera posición en la secuencia de una máquina.

4.4.3 Función de aptitud

Para abordar los distintos casos modelados, se han definido dos funciones de aptitud (fitness), que se aplicarán en función de la existencia o no de restricciones de turnos:

- **Fitness sin turnos.** Corresponde directamente al makespan obtenido a partir de la decodificación del segmento RHS.

- **Fitness con turnos.** se basa también en el makespan obtenido de la misma manera, pero incluye una penalización adicional cuando un lote excede la duración establecida de un turno. La penalización se calcula según la siguiente fórmula:

$$penalty = \delta \cdot \sum_{j}^{J} \sum_{u}^{U} \max\left(0, s_{mj} + p_{mj} \cdot \alpha'_{ju} - SHT\right)$$

Donde:

- δ es un coeficiente de escala que determina la importancia de la penalización en comparación con el makespan.
- s_{mj} (o s_{mjk}) es el tiempo de setup correspondiente a cada lote en una secuencia determinada.
- p_{mj} es el tiempo de proceso unitario del trabajo j en la máquina m.
- α'_{ju} es el tamaño real expresado en unidades de cada lote ju.
- SHT es el tiempo establecido de duración de un turno.

Por cada unidad de tiempo en la que se excede la duración establecida de un turno, se añade una penalización proporcional a δ en la función de aptitud. Para garantizar que la restricción de turnos tenga prioridad sobre la optimización del makespan, δ debe ser suficientemente grande. Dado que el valor del makespan varía según el tamaño de la instancia, δ también debe ajustarse en consecuencia, de manera que siempre se priorice el cumplimiento de la restricción.

4.5 Inicialización de la población

Se inicializa la población con el **método del torneo**. Este método de selección consiste en seleccionar aleatoriamente varios individuos de una población para que "compitan" entre ellos. El individuo con la mejor aptitud (fitness) en ese torneo es seleccionado para formar parte de la población.

En este caso, la población de la que se extraen los padres se genera aleatoriamente

La dinámica es:

1. Se elige el tamaño del torno, que indica cuántos individuos competirán en cada selección. Con el tamaño del torneo se controla la presión selectiva. Si el torneo es grande, será más probable que siempre se elijan los mejores individuos. Si es pequeño, se permite mayor diversidad.
2. Para cada torneo, se seleccionan aleatoriamente un grupo de individuos de la población.
3. De los individuos seleccionados, el que tiene la mejor aptitud es el "ganador" y se selecciona para ser

parte de la nueva población.

4. El proceso se repite hasta que se tiene una población nueva del tamaño deseado.

4.6 Operadores genéticos

Después de seleccionar soluciones de calidad (padres), se genera descendencia (hijos) haciendo pequeños cambios en estas soluciones. Estos cambios consisten en combinar y/o mutar información de los padres mediante operadores genéticos. En este caso se han implementado operadores diferentes para cada segmento de la solución.

4.6.1 Segmento LHS

Se tiene los siguientes operadores de **cruce**

- **Cruce de un solo punto a la izquierda (SPC-1).** Consiste en determinar un punto de corte aleatoriamente que divide a los cromosomas padres en dos partes, antes y después del punto. Se intercambia la información que queda a la izquierda del punto. En la Figura 17 se representa esta operación.

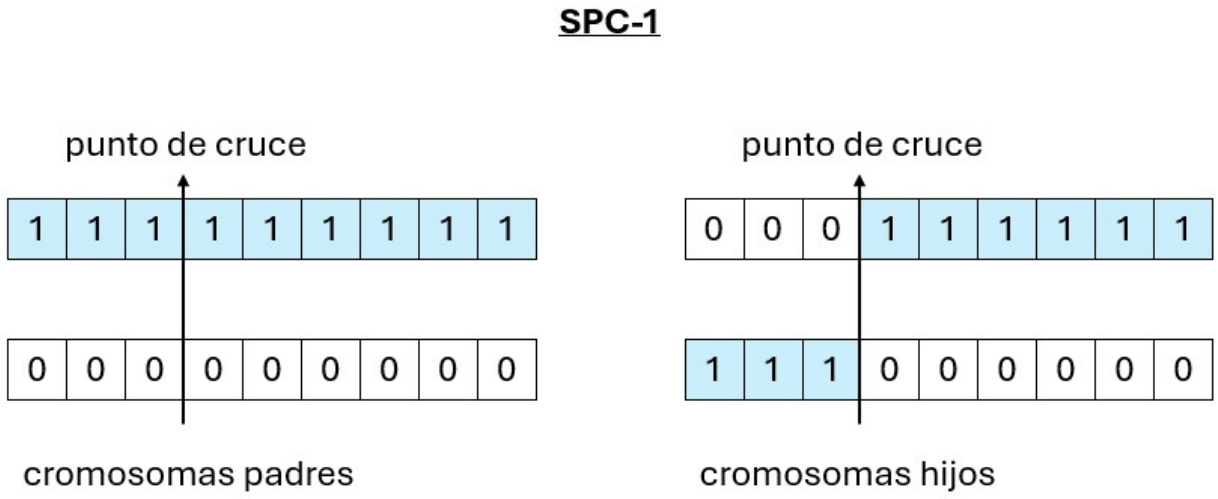

Figura 17. Operador genético SPC-1. [Fuente: elaboración propia].

- **Cruce de un solo punto a la derecha (SPC-2).** El mecanismo de cruce es el mismo de SPC-1 pero en vez de intercambiar la parte a la izquierda, se intercambia la parte a la derecha del punto de cruce. En la Figura 18 se representa esta operación.

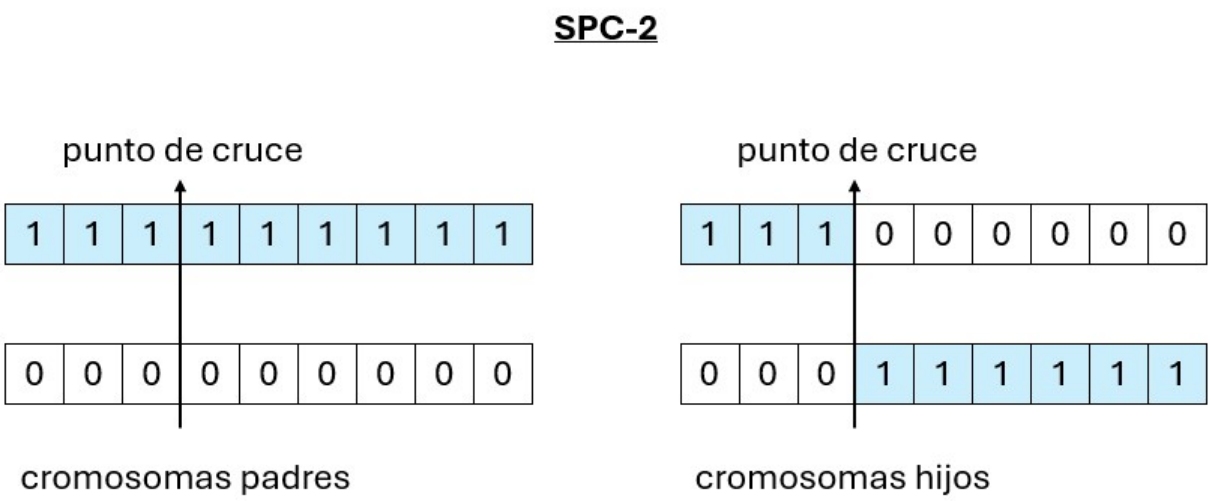

Figura 18. Operador genético SPC-2. [Fuente: elaboración propia].

Se tiene el siguiente operador de **mutación:**

- **Incremento/decremento controlado de lotes (SStM).** Este operador se aplica para aumentar o disminuir el valor de cada gen (tamaño de un lote) según una probabilidad pequeña σ_1 en una cantidad de paso θ. Por lo tanto, no todos los genes serán mutados, solo algunos y de forma aleatoria según una probabilidad $\sigma_1 = 0.2$. Este es un valor de referencia y puede ser modificado para variar el comportamiento del operador genético.

  Si el operador se aplica a un gen, la probabilidad de que aumente o disminuya su valor es igual, 50%. Las fórmulas que determinan los cambios son:

  - Para **incrementar** el valor del gen: $\alpha_{ju} = \min\{1, \alpha_{ju} + \delta\}$.
  - Para **decrementar** el valor del gen: $\alpha_{ju} = \max\{0, \alpha_{ju} - \delta\}$.

  Así se garantiza que el valor del gen siempre se mantiene dentro del intervalo [0, 1].

  La **cantidad de paso $\boldsymbol{\theta}$** se calcula cada vez que se aplica el operador de mutación.

$$\theta = \theta_{max} \cdot r$$

  Aquí:

- θ_{max} es un parámetro que controla el valor máximo que puede tener θ. En este caso se establece $\theta_{max} = 0.5$.m
- r es un número perteneciente al intervalo [0, 1].

4.6.2 Segmento RHS

En la codificación del segmento RHS, un mismo gen se repite tantas veces como número máximo de lotes se

haya indicado. Por ello y para poder usar los operadores de la librería de Python DEAP (explicada más adelante), se transforma el segmento RHS en una lista de números enteros que no se repiten. En otras palabras, se genera una lista con la posición de cada gen. Se aplica el operador genético y después se deshace la transformación, atendiendo a los cambios de posición.

Se tienen los siguientes operadores de **cruce**:

- **Cruce parcialmente emparejado (PMX).** Se eligen dos puntos de corte aleatorios dentro de los cromosomas, que definen el segmento de información que se intercambia entre los 2 padres para generar 2 hijos.

  Este operador presenta una complejidad, el emparejamiento parcial para la resolución de conflictos. Después de realizar el cruce, puede que un hijo contenga un valor ya presente en el segmento de intercambio y por lo tanto puede haber valores repetidos. Esto se resuelve realizando permutaciones, utilizando el mapeo de los padres para asegurar que los hijos contengan todos los valores de forma única. En la Figura 19 se ejemplifica un cruce PMX.

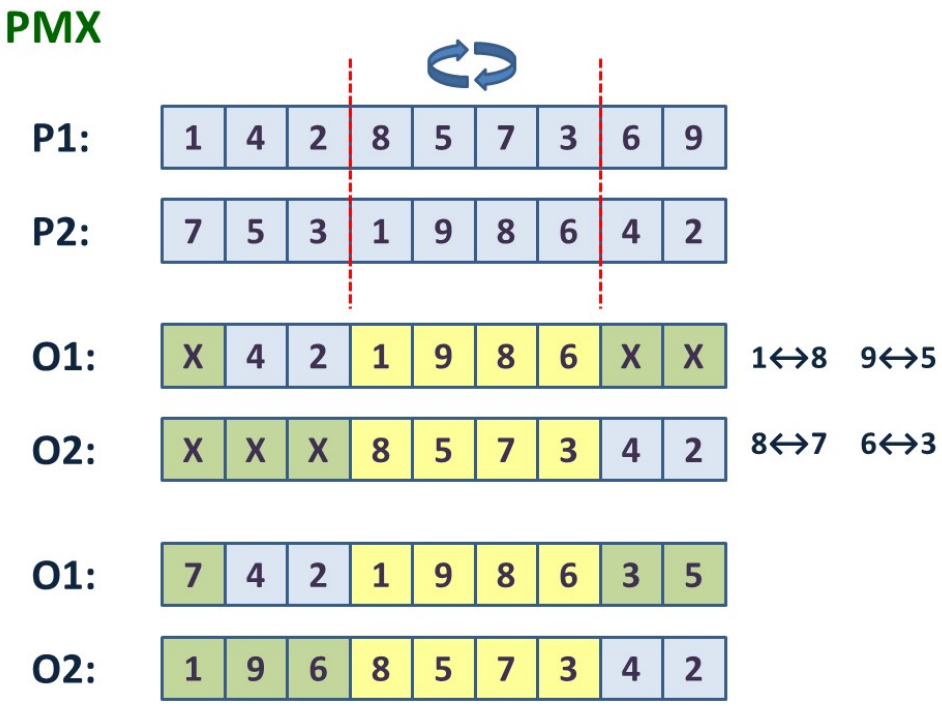

Figura 19. Operador genético PMX. [Fuente: [25]].

- **Cruce por orden (OX).** Se seleccionan dos puntos de corte aleatorios, que definen un segmento que será copiado de uno de los padres al hijo. Se copia el segmento del primer padre y los huecos restantes se rellenan con elementos del segundo padre en el orden en el que aparecen los elementos. Para no repetir valores, se ignoran los elementos que ya están en el hijo. El proceso se repite para generar el segundo hijo, usando el segundo padre como base para copiar el segmento y el primer padre para rellenarlo. En la Figura 20 se ejemplifica un cruce OX.

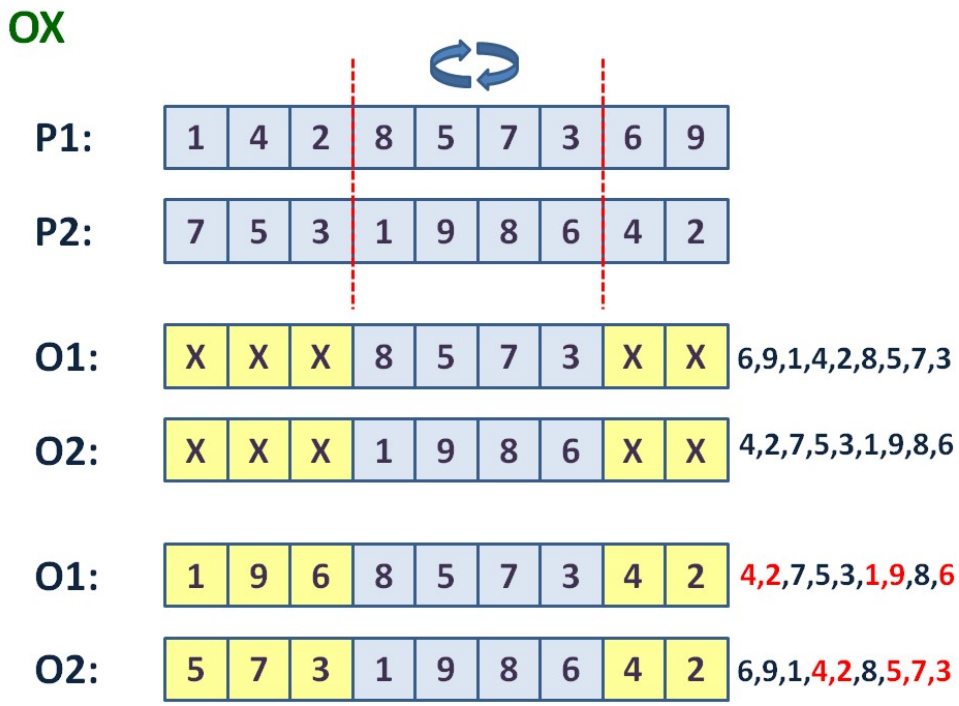

Figura 20. Operador genético OX. [Fuente: [25]].

- **Cruce a nivel de trabajo (JL).** Se elige aleatoriamente un trabajo del padre 1. Este trabajo se utiliza como base para crear el primer hijo. En el hijo 1, se copian las operaciones (lotes) del trabajo seleccionado manteniendo las mismas posiciones que tenían en el padre 1. A continuación se rellenan las posiciones vacías con los genes del padre 2, siguiendo el orden en el que aparecen, pero sin duplicar los lotes del trabajo que ya fue copiado anteriormente. El hijo 2 se hace de manera análoga, cambiando los papeles de los padres. El padre 2 será la base y el padre 1 será el que rellene los huecos vacíos. En la Figura 21 se ilustra el mecanismo de combinación de este operador.

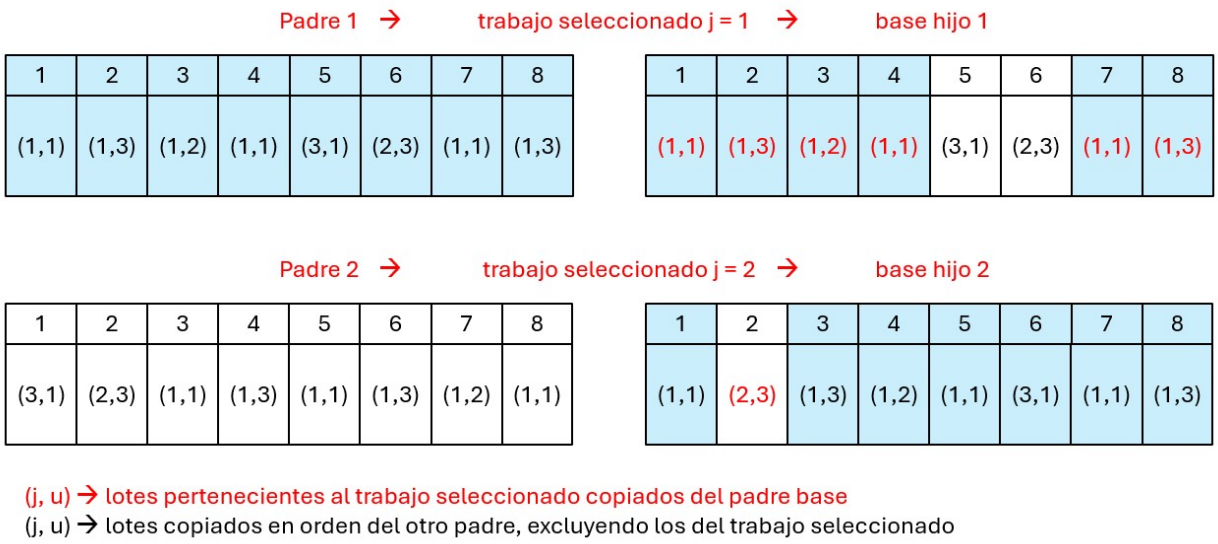

Figura 21. Operador genético JL. [Fuente: elaboración propia].

Se tiene el siguiente operador de **mutación:**

- **Mutación por reordenamiento de índices (MSI).** Este operador introduce variabilidad al barajar aleatoriamente algunos elementos dentro de un individuo, es decir, reorganiza ciertos índices de una secuencia. Para cada gen (índice) del cromosoma, se evalúa con una determinada probabilidad (en este caso 0.2) si será seleccionado o no. Después los genes seleccionados se mezclan aleatoriamente, cambiando la posición entre ellos. Finalmente, se copian estos genes en el cromosoma en el orden anterior. El resultado es el mismo cromosoma con algunos de sus genes cambiados de posición.

4.7 Hiperparámetros

El comportamiento del algoritmo genético cambia según el valor de una serie de parámetros que se definen antes de ejecutar el algoritmo. Dado que en el campo de la optimización y la optimización artificial hay algoritmo o modelos que van ajustando algunos parámetros automáticamente, cabe hacer una distinción entre parámetros e hiperparámetros. En general, los parámetros son ajustados en el transcurso del algoritmo y los hiperparámetros se seleccionan cuidadosamente antes de la ejecución.

La elección adecuada de los hiperparámetros repercute en la eficiencia del algoritmo y en la calidad de las soluciones generadas.

En este estudio se consideran **hiperparámetros**:

- **Probabilidades de cruce y mutación.** Probabilidades de que se apliquen operadores genéticos de cruce entre 2 padres u operadores de mutación en cada individuo.
- **Tamaño de la población.** El número total de individuos que componen la población y que se evalúan en cada generación.

- **Número de generaciones.** Número de iteraciones del algoritmo que se llevan a cabo, en este caso número de veces que se generan poblaciones descendientes.
- **Tamaño del torneo.** Número de individuos seleccionados aleatoriamente de la población que participan en cada torneo.

Además, se tienen 2 parámetros para cambiar la naturaleza del problema pero que no influyen directamente en la eficiencia del algoritmo genético. Son la selección de restricciones de turnos y de tiempos de setup dependientes de la secuencia. Dado que el efecto de los hiperparámetros que afectan al rendimiento del algoritmo puede cambiar según la naturaleza del problema, se tienen en cuenta.

En la Tabla 3 se agrupan todos los hiperparámetros, se indica su efecto y un valor de referencia base, con el que se ha probado en pruebas piloto que el algoritmo es válido y mejora la calidad de las soluciones.

Hiperparámetro	**Notación**	**Efecto**	**Valor de referencia**
Probabilidad de cruce	C_p	Rendimiento	0.8
Probabilidad de mutación	M_p	Rendimiento	0.2
Tamaño de la población	P_s	Rendimiento	150
Número de generaciones	G_s	Rendimiento	150
Tamaño del torneo	T_s	Rendimiento	3
Restricciones de turnos	-	Naturaleza del problema	-
Tiempos de setup dependientes de la secuencia	-	Naturaleza del problema	-

Tabla 3. Hiperparámetros del algoritmo genético. Efecto y valores de referencia.

4.8 Criterio de parada

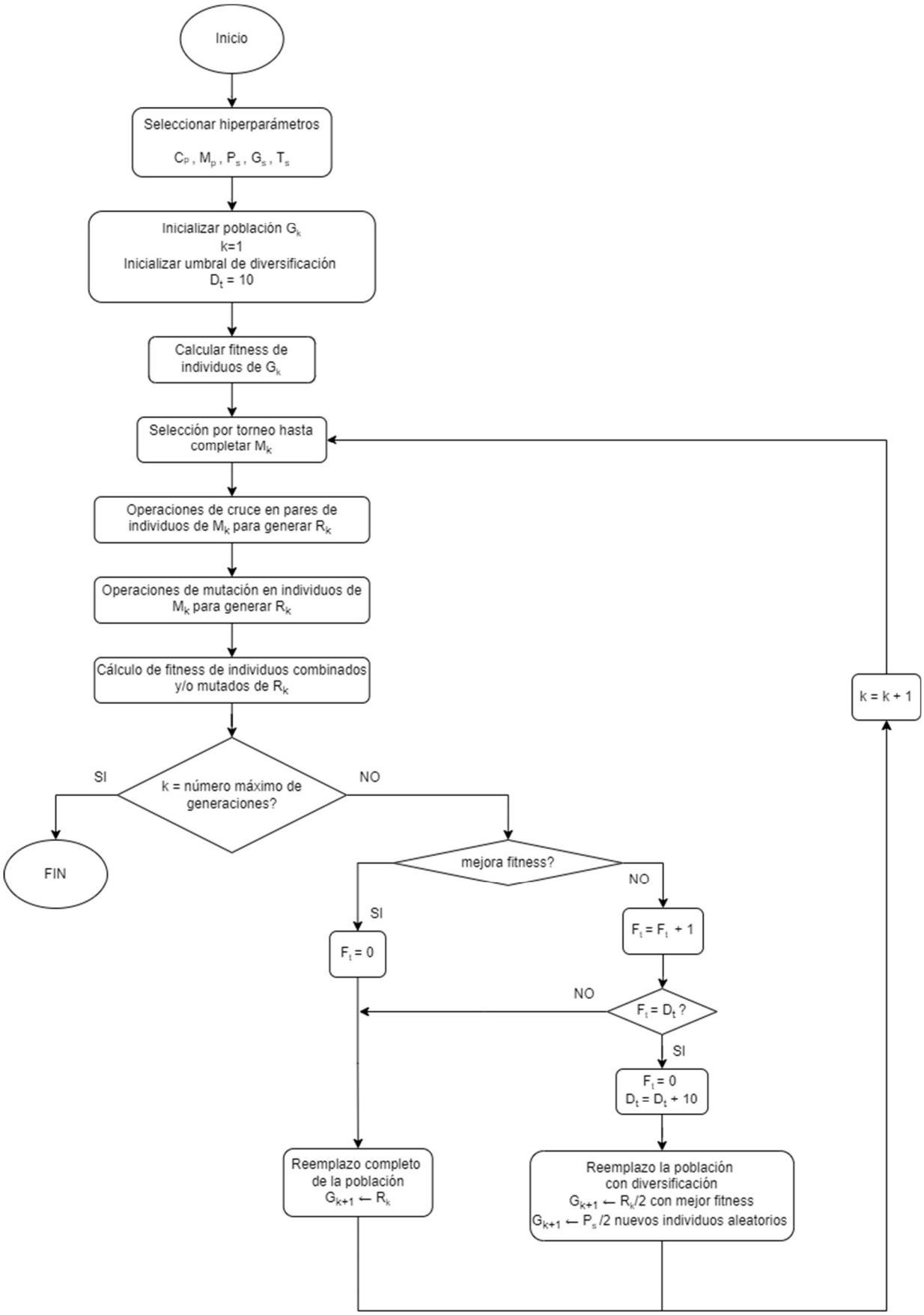

Figura 22. Diagrama de flujo del algoritmo genético. [Fuente: elaboración propia].

En la Figura 22 se ofrece el diagrama de flujo del algoritmo, en el que se ilustra el criterio de parada y el mecanismo de diversificación implementado.

En la particularización del algoritmo genético que se ha hecho, el criterio de parada es simple y sencillo, el número de generaciones. El valor de referencia son 50 generaciones, usado en instancias de tamaño pequeño. El número de generaciones debe crecer con el tamaño de la instancia, para facilitar la exploración del espacio de soluciones.

Se ha implementado un **mecanismo de diversificación** para ayudar al algoritmo a "salir" de mínimos locales y seguir mejorando la solución. El funcionamiento es el siguiente:

1. Se establece un umbral de diversificación $D_t = 10$ que representa el número de generaciones máximo en las que no mejora el fitness y que activa el mecanismo de diversificación.
2. Alcanzado este umbral, se elimina la mitad de la población generada que tiene peor fitness y se reemplaza con nuevos individuos generados aleatoriamente. De esta manera, la nueva generación está formada a partes iguales por los individuos generados en esa iteración con mejor fitness y por individuos nuevos aleatorios, que introducen material genético nuevo.
3. Cada vez que se activa este mecanismo, el umbral de diversificación D_t se aumenta en 10 unidades. Cómo cada vez es más complicado encontrar soluciones mejores, se busca la posibilidad de explorar cada vez más a fondo el espacio de soluciones fruto del material genético existente y los operadores genéticos implementados.

5 IMPLEMENTACIÓN EN PYTHON

En este capítulo se explica, en líneas generales, cómo se ha implementado el algoritmo en Python.

En el apartado 5.1 se expone cuáles son las librerías que se han usado y que funcionalidades ofrece cada una de ellas. En el apartado5.2 se detallan los módulos implementados, indicando el contenido y la funcionalidad de cada uno. También se ofrece la arquitectura del programa completo, en la que se ilustra como se relacionan los módulos entre sí. Por último, en el apartado 5.3 se ofrece el bloque de código principal, con el que se coordinan las funcionalidades de todos los módulos y se ejecuta el programa completo.

Python es un lenguaje de alto nivel conocido por su sintaxis simple y legible. Hoy en día, es uno de los lenguajes más populares. Cuenta con una enorme comunidad de desarrolladores que han creado numerosas bibliotecas. Es flexible, fácil de aprender y ampliamente utilizado, lo que lo convierte en una opción muy atractiva.

Para este trabajo se ha usado el entorno de desarrollo Virtual Studio Code (VS Code) desarrollado por Microsoft. Es gratuito y de código abierto. Soporta múltiples lenguajes, tiene un módulo para descargar extensiones fácilmente, se integra con Git y Github de forma sencilla, ofrece herramientas de depuración, tiene una terminal integrada para ejecutar código y es altamente personalizable.

La versión de Python usada es la 3.11.4.

5.1 Librerías

Se han usado las siguientes librerías:

- **NumPy (Numerical Python).** Es una librería que se usa para la manipulación de datos numéricos, álgebra lineal y cálculos científicos. En este caso se ha usado porque está optimizada para trabajar con arrays (matrices y vectores). De esta manera, es más sencillo manipular estas estructuras de datos y la ejecución del código es más rápida.

- **Pandas.** Esta librería está construida sobre NumPy. Ofrece estructuras de datos y herramientas para trabajar con datos estructurados. Aquí se han usado para manipular los valores de los parámetros de las soluciones decodificadas a través de DataFrames. Un DataFrame es la estructura de datos principal de Pandas, similar a una base de datos o una hoja de cálculo en Excel.
- **Random.** Es una biblioteca estándar de Python (viene incluido en la instalación de Python) que proporciona funciones para generar números aleatorios, mezclar listas o seleccionar elementos al azar, entre otras. Es fundamental para introducir variabilidad, como por ejemplo, en la selección de individuos.
- **Time.** Pertenece también al conjunto de librerías estándar de Python. Proporciona funciones para trabajar con el tiempo y medir la ejecución del programa en distintos períodos. Se usa para conocer el rendimiento del algoritmo.
- **Plotly.express.** Es un módulo de la biblioteca Plotly de Python que permite crear gráficos interactivos y atractivos de forma sencilla. En concreto, se ha usado el módulo plotly.express.timeline para hacer la representación gráfica de la solución (diagrama de Gantt). Es común encontrar en foros y repositorios de GitHub estos diagramas implementados en la librería plotly.figure_factory. Sin embargo, a medida que se añaden nuevas funcionalidades a plotly.express, plotly.figure_factory queda obsoleta y reemplazada por plotly.express.timeline.
- **DEAP (Distributed Evolutionary Algorithms in Python).** Está diseñada para facilitar la implementación de algoritmos evolutivos de optimización. La construcción y modificación del algoritmo genético es mucho más sencilla, limpia y rápida gracias a las herramientas y estructuras de datos que se ofrecen. Los componentes claves en esta aplicación son:
    - **Estructuras de datos** para generar poblaciones, seleccionar a los mejores individuos, etc.
    - **Operadores genéticos** de cruce y mutación, que al ser de código abierto se pueden modificar y/o complementar fácilmente.

5.2 Módulos

Para construir el algoritmo genético y desarrollar todas las funcionalidades necesarias, se han desarrollado varios módulos. Los módulos son archivos que contienen definiciones de funciones, clases y variables que se pueden reutilizar en otros programas. El propósito principal es organizar y estructurar el código, de forma que sea más fácil mantenerlo y reutilizarlo.

Los módulos son:

1. **Params.py.** Se encarga de generar parámetros aleatorios para el problema de programación de trabajos

en entornos tipo taller. Se define la clase `JobShopRandomParams` que encapsula los datos necesarios para crear instancias aleatorias del problema, como el número de trabajos, máquinas, tiempos de operación, etc. Estos datos son usados posteriormente por otros módulos para generar y evaluar cromosomas.

2. **chromosome_generator.py.** Contiene una función que se encarga de crear un cromosoma aleatorio. El cromosoma contiene una secuencia de genes que representan la información de un programa factible. Es clave para iniciar la población y/o añadir material genético nuevo.
3. **decoder.py.** Decodifica los cromosomas y los evalúa usando un algoritmo de programación semi activa. Contiene 3 funciones fundamentales.
    a. `decode_chromosome` decodifica un cromosoma según la demanda establecida. Devuelve el instante de inicio de preparación de cada lote y el instante de finalización.
    b. `get_chromosome_start_times` calcula los tiempos de inicio del procesado de cada lote, fundamental para construir el diagrama de Gantt del programa y diferenciar entre tiempos de setup y tiempos de procesado.
    c. `build_chromosome_results_df` genera un DataFrame con todos los datos necesarios para graficar el diagrama de Gantt.
4. **genetic_operators.py.** Implementa los operadores del algoritmo genético que son esenciales para evolucionar la población de soluciones.
5. **genetic_algorithm.py.** Es el módulo central que implementa el algoritmo genético. Coordina todo el proceso de evolución, evaluando cada cromosoma, aplicando los operadores genéticos, actualizando la población, etc.
6. **plotting_ga.py.** Se encarga de crear el gráfico de Gantt para mostrar el programa de producción de la mejor solución obtenida a través del algoritmo genético. Ayuda a interpretar los resultados que se obtienen en la decodificación del cromosoma.
7. **main.py.** Es el bloque de código principal, el que ejecuta el flujo completo. Coordina la generación de parámetros, la ejecución del algoritmo genético y la visualización de los resultados.

En la Figura 23 se ofrece la arquitectura del programa. En cada módulo se detallan las funciones implementadas más relevantes y las librerías usadas. Están conectados con flechas, a modo de representación del flujo de información entre ellos.

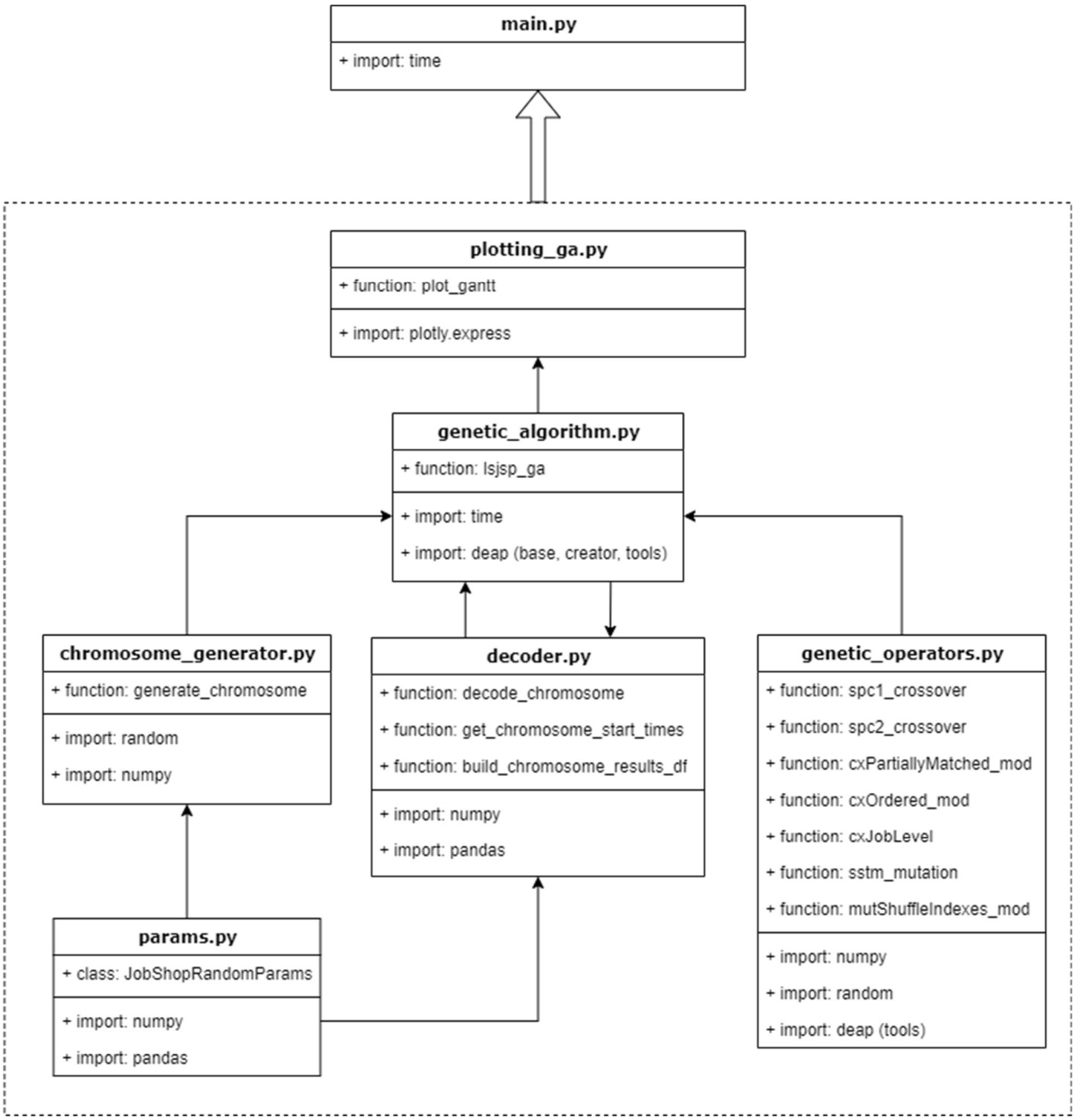

Figura 23. Módulos implementados en Python. funciones principales, librerías usadas y flujo. [Fuente: elaboración propia].

5.3 Bloque principal

Este es el código que se ha implementado en el módulo main.py para ejecutar el algoritmo genético y obtener un diagrama de Gantt de la mejor solución. Se pueden diferenciar 3 secciones:

En la primera, se importa la librería estándar *time* y todos los módulos.

En la segunda sección se definen los parámetros de la instancia que se va a generar aleatoriamente con los datos del entorno tipo taller, incluyendo la posible inclusión de las restricciones de turnos y la naturaleza de los tiempos de setup. Se genera la instancia y se imprimen por pantalla todos los parámetros.

En la tercera y última sección se definen dos hiperparámetros del algoritmo genético, el tamaño de la población y el número de generaciones. El resto de hiperparámetros se pueden cambiar en el módulo relativo al algoritmo. Se ejecuta el algoritmo genético y se imprime por pantalla el tiempo transcurrido hasta la parada y el mejor valor obtenido para la función de aptitud. Si no hay restricciones de turnos, este será el valor del makespan. En caso de que las haya, se imprime por pantalla cual es el valor del makespan y cuál es el valor de la penalización por el incumplimiento de la restricción.

El bloque principal de código, escrito en lenguaje Python es:

```
# import standard libraries
import time

# import all the modules (scripts) in the package
import chromosome_generator
import params
import decoder
import plotting_ga
import genetic_algorithm
from genetic_operators import *

# --------- JOB SHOP PARAMETERS AND CONSTRAINTS (CHANGE FOR DIFFERENT
SCENARIOS) ---------
# constraints
shifts_constraint = True
sequence_dependent = True

# Parameters
n_machines = 5  # number of machines
n_jobs = 6  # number of jobs
n_lots = 5  # number of lots
seed = 4  # seed for random number generator
demand = {i: 100 for i in range(0, n_jobs + 1)}  # demand of each job

# Create parameters object
my_params = params.JobShopRandomParams(
    n_machines=n_machines, n_jobs=n_jobs, n_lots=n_lots, seed=seed
)
my_params.demand = demand  # demand of each job
my_params.printParams(sequence_dependent=sequence_dependent,
save_to_excel=False)
```

```
# --------- GENETIC ALGORITHM ---------
# Genetic algorithm parameters (CHANGE FOR DIFFERENT GA RUNS)
pop_size = 100
n_generations = 100

# Genetic algorithm
start = time.time()
best_fitness, best_individual = genetic_algorithm.lsjsp_ga(
    my_params,
    population_size=pop_size,
    num_generations=n_generations,
    shifts=shifts_constraint,
    seq_dep_setup=sequence_dependent,
    plot=True,
)
end = time.time()
print("Time elapsed: ", end - start)
print("Best_fitness: ", best_fitness)
if shifts_constraint:
    makespan, penalty, _* = decoder.decode_chromosome(
        best_individual,
        my_params,
        shifts=shifts_constraint,
        seq_dep_setup=sequence_dependent,
    )
    print("Makespan: ", makespan)
    print("Penalty: ", penalty)
```

6 EXPERIMENTACIÓN NUMÉRICA Y VALIDACIÓN

En el presente capítulo se prueba y se valida el algoritmo para cada uno de los cuatro escenarios modelados en el capítulo 3. Además se prueba su rendimiento en una instancia más "grande" que la usada para validarlo y considerando el escenario más complejo (turnos y tiempos de setup dependientes de la secuencia) y más difícil de resolver mediante el correspondiente MILP propuesto.

En el apartado 6.1 se explica cómo se han generado las instancias del problema y el rango de valores usado para cada parámetro. Se ofrecen los datos de las instancias con las cuales se prueba y valida el modelo.

En el apartado 6.2 se muestran los resultados de varias simulaciones llevadas a cabo para cada escenario, prestando especial atención al gap de optimalidad obtenido. Se valida el rendimiento del algoritmo genético con esta métrica. En el apartado 6.3 se ofrecen los resultados obtenidos para el caso más complejo y con una instancia de tamaño considerable. Mediante la resolución del modelo MILP correspondiente, con los recursos disponibles, se desconoce el tiempo para encontrar una solución admisible. Por último, en el apartado 6.4 se exponen las conclusiones del capítulo, en las que se habla principalmente de las bondades y carencias del algoritmo.

Con esta experimentación el objetivo es validar el algoritmo, ver cómo responde ante distintas variaciones del problema genérico, superar las limitaciones de la resolución mediante modelos matemáticos y conocer las áreas de mejora.

Todos los modelos matemáticos se han implementado en Python y las soluciones óptimas se han obtenido usando el *solver* Gurobi. Los experimentos se llevan a cabo en un ordenador con un procesador Intel(R) Core (TM) i7-7700HQ CPU (2.80GHz, 2808 Mhz, 4 procesadores principales, 8 procesadores lógicos) con 16 GB de RAM.

6.1 Generación de instancias (datos del problema)

Para probar, evaluar y comparar el algoritmo genético, se generan instancias aleatorias de datos. Se controla la reproducibilidad de estas con una semilla y los rangos de valores de cada parámetro del problema.

El objetivo es por un lado tener el control absoluto en las dimensiones del problema y por otro lado tener la capacidad de generar problemas diferentes para evaluar modelos matemáticos y algoritmos.

Se intenta mantener una relación adecuada entre todos los parámetros del problema. Es decir, definir rangos de valores que tienen sentido y que pudieran darse en entornos reales.

Los datos de entrada para generar una instancia y el rango de valores de los ejemplos propuestos son:

- **Número de máquinas**. Entre 3 y 5.
- **Número de trabajos** (entendidos como productos). Entre 3 y 6.
- **Número máximo de sublotes de cada trabajo**. Entre 3 y 5.
- **Semilla** del generador de números aleatorios.
- **Demanda unitaria de cada trabajo**. Entre 50 y 200.

Los datos de salida de la instancia son los datos de entrada más los siguientes datos:

- **Matriz de tiempos de proceso unitarios**. Se establece el rango [1,20].
- **Matriz de tiempos de *setup* independientes de la secuencia**. Se establece el rango [50,100].
- **Secuencia de proceso de cada trabajo**. No incluye necesariamente todas las máquinas y una máquina no se incluye dos veces en la secuencia de un mismo trabajo.

Previo a este trabajo, se validó el modelo MILP más básico (tiempos de setup independientes de la secuencia y sin turnos) presentado aquí con diferentes instancias. Con el fin de evaluar y validar el algoritmo, analizar el efecto de los hiperparámetros y conocer el gap de optimalidad, se usan de nuevo estas instancias, añadiendo los tiempos de setup dependientes de la secuencia cuando corresponde.

En concreto, se trata con 2 instancias calificadas como “pequeña”, y “grande” por el coste computacional que supone su resolución exacta.

En este trabajo, se usa:

- La **instancia pequeña** para conocer el gap de optimalidad del algoritmo genético.
- La **instancia grande** para evaluar el rendimiento del algoritmo en problemas de mayor dimensión.

En la Tabla 4, y Tabla 5 se exponen los datos de estas instancias.

INSTANCIA PEQUEÑA

Tiempos de proceso

	Job 1	Job 2	Job 3
Máquina 0	4	15	16
Máquina 1	7	17	10
Máquina 2	9	5	8

Tiempos de *setup* independientes

	Job 1	Job 2	Job 3
Máquina 0	85	64	97
Máquina 1	88	66	59
Máquina 2	58	86	89

Secuencia		**Número de lotes máximo**	3 (varía según capacidad del MILP)
Job 1	[2, 0, 1]		
Job 1	[2, 1, 0]	**Semilla**	5
Job 2	[2]		

Tiempos de *setup* dependientes

Máquina 0	Job 1	Job 2	Job 3
Job -1	50	50	50
Job 1	0	88	58
Job 2	85	0	86
Job 3	64	66	0

Máquina 1	Job 1	Job 2	Job 3
Job -1	50	50	50
Job 1	0	66	65
Job 2	77	0	99
Job 3	98	57	0

Máquina 2	Job 1	Job 2	Job 3
Job -1	50	50	50
Job 1	0	63	97
Job 2	66	0	80
Job 3	77	61	0

Tabla 4. Parámetros de la instancia pequeña.

INSTANCIA GRANDE

Tiempos de proceso

	Job 1	Job 2	Job 3	Job 4	Job 5	Job 6
Máquina 0	4	15	16	7	17	10
Máquina 1	9	5	8	17	17	8
Máquina 2	13	16	18	8	17	13
Máquina 3	14	12	2	16	19	10
Máquina 4	11	10	10	2	19	8

Tiempos de *setup* independientes

	Job 1	Job 2	Job 3	Job 4	Job 5	Job 6
Máquina 0	85	64	97	88	66	59
Máquina 1	58	86	89	77	98	80
Máquina 2	66	57	62	65	99	89
Máquina 3	66	77	94	63	61	51
Máquina 4	97	80	70	72	68	59

Secuencia		**Número de lotes máximo**	10
Job 1	[3, 2, 4, 1]		
Job 1	[3, 1, 4, 2]	**Semilla**	5
Job 2	[2, 3, 0, 4, 1]		
Job 3	[4, 0, 2, 1, 3]		
Job 4	[0, 3, 2]		
Job 6	[4, 3]		

Tiempos de *setup* dependientes

Máquina 0	Job 1	Job 2	Job 3	Job 4	Job 5	Job 6
Job -1	50	50	50	50	50	50
Job 1	0	58	66	66	97	92
Job 2	85	0	57	77	80	91
Job 3	64	86	0	94	70	91
Job 4	97	89	62	0	72	51
Job 5	88	77	65	63	0	68
Job 6	66	98	99	61	68	0

Máquina 1	**Job 1**	**Job 2**	**Job 3**	**Job 4**	**Job 5**	**Job 6**
Job -1	50	50	50	50	50	50
Job 1	0	96	54	99	61	64
Job 2	66	0	88	57	93	77
Job 3	64	86	0	86	53	79
Job 4	55	91	91	0	51	83

Job 5	50	77	69	94	0	75
Job 6	66	81	69	95	53	0

Máquina 2	Job 1	Job 2	Job 3	Job 4	Job 5	Job 6
Job -1	50	50	50	50	50	50
Job 1	0	59	95	82	80	66
Job 2	57	0	74	62	68	50
Job 3	66	88	0	94	50	58
Job 4	80	97	63	0	89	76
Job 5	64	66	93	55	0	94
Job 6	76	55	90	64	57	0

Máquina 3	**Job 1**	**Job 2**	**Job 3**	**Job 4**	**Job 5**	**Job 6**
Job -1	50	50	50	50	50	50
Job 1	0	68	65	58	68	69
Job 2	80	0	56	82	67	99
Job 3	61	77	0	68	72	54
Job 4	55	79	85	0	69	68
Job 5	91	74	97	63	0	86
Job 6	72	83	70	78	93	0

Máquina 4	Job 1	Job 2	Job 3	Job 4	Job 5	Job 6
Job -1	50	50	50	50	50	50
Job 1	0	56	50	58	96	50
Job 2	70	0	58	60	55	52
Job 3	82	59	0	72	85	52
Job 4	60	80	58	0	93	99
Job 5	81	64	57	58	0	77
Job 6	65	52	62	82	93	0

Tabla 5. Parámetros de la instancia grande.

6.2 Validación y Gap de optimalidad

En el desarrollo de algoritmos metaheurísticos es fundamental evaluar su rendimiento. Una opción es emplear métricas que comparen las soluciones obtenidas con aquellas consideradas óptimas. Para validar la efectividad de estos algoritmos, una de las herramientas más utilizadas es el **gap de optimalidad**, el cual mide la diferencia entre la solución proporcionada por la metaheurística y la solución óptima obtenida a través de métodos exactos como la **programación entera mixta (MILP)**.

El gap de optimalidad se expresa generalmente como un porcentaje para cuantificar como de cerca o lejos está una solución heurística de la mejor solución posible. Esta medida sirve para evaluar la calidad de las soluciones

obtenidas y consecuentemente puede ser útil en la identificación de posibles mejoras en los hiperparámetros del algoritmo, como el tamaño de la población, las tasas de mutación o cruce, y el número de generaciones.

Dado que las metaheurísticas no garantizan la obtención de la solución óptima, el análisis del gap de optimalidad se convierte en un paso relevante en la validación de estas y de su rendimiento. En esta sección se ofrecen los resultados obtenidos por el algoritmo genético para una instancia pequeña y se comparan con soluciones óptimas obtenidas a partir de los modelos propuestos en el apartado 3.2. Los datos de la instancia se ofrecen en la Tabla 4.

La demanda influye significativamente en el tamaño del espacio de soluciones y, en consecuencia, en el coste computacional de encontrar el óptimo utilizando modelos MILP. Por esta razón, en cada escenario se selecciona un valor de demanda que se ajuste a la capacidad computacional disponible para resolver el problema de manera óptima en un tiempo razonable.

Para cada subproblema se corre el algoritmo genético 10 veces con los hiperparámetros mostrados en la Tabla 3 y se calcula la media, que se usará para calcular el gap de optimalidad.

6.2.1 LSJSP

Para este caso el valor de la **demanda es 200 unidades** para cada trabajo. La resolución del modelo MILP ofrece un makespan igual a **5380 unidades**.

Los resultados obtenidos con el algoritmo genético se muestran en la Tabla 6.

CASO 1

- Sin restricciones de turnos
- Tiempos de setup independientes de la secuencia

Simulación	Coste computacional [s]	Mejor Fitness (Makespan)	Gap de optimalidad
1	71.82302	5380	0.00% (min)
2	71.24322	5580	3.72%
3	70.30249	5902	9.70%
4	76.15932	5678	5.54%
5	74.18819	5959	10.76%
6	72.59195	6001	11.54% (max)
7	82.95092	5785	7.53%
8	77.59087	5881	9.31%
9	85.54608	5634	4.72%
10	92.97371	5805	7.90%
11	79.68218	5553	3.22%
12	82.26571	5787	7.57%
13	68.91836	5782	7.47%
14	71.73835	5772	7.29%
15	69.83709	5700	5.95%

Tabla 6. Resultados del algoritmo genético para el caso genérico con datos de una instancia pequeña.

Para una muestra de 15 simulaciones del algoritmo, se obtiene

- **Media** = 5746,6 unidades.
- **Desviación típica** = 164,67 unidades.
- **Gap de optimalidad** en el rango [0, 11.54%].

El gap de optimalidad de la media es:

$$Gap\ de\ Optimalidad = \frac{5746.6 - 5380}{5380} = 6{,}81\%$$

En base a los resultados obtenidos, se valida el algoritmo para el escenario genérico.

6.2.2 LSJSP con turnos

En este escenario es necesario reducir el valor de la **demanda a 100 unidades** para encontrar la solución óptima con el modelo matemático. Además, **el número de lotes máximo debe aumentar a 5**, sino no se encuentran soluciones factibles. La **solución óptima** presenta un makespan de **3304 unidades**.

Los resultados obtenidos con el algoritmo genético se muestran en la Tabla 7.

CASO 2

- Con restricciones de turnos
- Tiempos de setup independientes de la secuencia

Simulación	Coste computacional [s]	Mejor Fitness	Makespan	Penalty	Gap de optimalidad
1	151.2888	5431	5431	0	64.38%
2	150.4572	4954	4954	0	49.94%
3	135.5642	4679	4679	0	41.62%
4	137.3245	5708	5708	0	72.76% (max)
5	128.1541	4684	4684	0	41.77%
6	128.5837	4734	4734	0	43.28%
7	124.1782	4939	4939	0	49.49%
8	127.5618	4506	4506	0	36.38%
9	126.2991	5104	5104	0	54.48%
10	127.6128	4474	4474	0	35.41%
11	129.0984	5449	5449	0	64.92%
12	131.8549	5074	5074	0	53.57%
13	104.8537	4459	4459	0	34.96% (min)
14	115.0907	4969	4969	0	50.39%
15	105.5477	5089	5089	0	54.03%

Tabla 7. Resultados del algoritmo genético para el caso con turnos y tiempos de setup independientes de la secuencia con datos de una instancia pequeña.

Para una muestra de 15 simulaciones del algoritmo, se obtiene

- **Media** = 4950,2 unidades.
- **Desviación típica** = 375,07 unidades.
- **Gap de optimalidad** en el rango [34.96, 72.76%]

El gap de optimalidad de la media es:

$$Gap\ de\ Optimalidad = \frac{4950{,}2 - 3304}{3304} = 49{,}82\%$$

Se observa en este caso que el gap de optimalidad que presenta la media del mejor fitness es bastante elevado. Es claro que existen carencias en el mecanismo del algoritmo para manejar esta instancia en este caso en concreto. Se observa en la evolución de la solución que durante 100 o incluso más iteraciones el algoritmo no es capaz de mejorar. Aun así, en todas las simulaciones se mejora la mejor solución inicial y se cumple la restricción de turnos. Es por ello por lo que tiene sentido hacer uso del algoritmo para resolver el problema.

En el caso de implementar restricciones de turnos con tiempos de setup independientes, el espacio de búsqueda disminuye con respecto al caso más genérico, es más restringido. Quizás, para mejorar una solución de alta calidad se debiese pasar por una amplia zona restringida para llegar a una solución mejor. Este podría ser el problema por el cual el gap de optimalidad es tan elevado.

6.2.3 LSJSP con tiempos de setup dependientes de la secuencia

En este escenario se establece el valor de la **demanda en 100 unidades** para encontrar la solución óptima con el modelo matemático. **El número de lotes máximo se mantiene en 3.** El makespan de la **solución óptima** es **2729 unidades**.

Los resultados obtenidos con el algoritmo genético se muestran en la Tabla 8.

CASO 3

- Sin restricciones de turnos
- Tiempos de setup dependientes de la secuencia

Simulación	Coste computacional [s]	Mejor Fitness	Makespan	Penalty	Gap de optimalidad
1	74.34494	2809	2809	0	2.93% (min)
2	63.71181	3037	3037	0	11.29% (max)
3	67.24556	2869	2869	0	5.13%
4	67.80047	2809	2809	0	2.93% (min)
5	69.64733	3029	3029	0	10.99%
6	69.36814	2861	2861	0	4.84%
7	69.95311	2841	2841	0	4.10%
8	69.56102	2989	2989	0	9.53%
9	68.89762	2839	2839	0	4.03%
10	68.58642	2809	2809	0	2.93% (min)
11	69.56385	2851	2851	0	4.47%
12	67.10459	2985	2985	0	9.38%
13	69.56565	3013	3013	0	10.41%
14	70.29774	3002	3002	0	10.00%
15	66.16722	2998	2998	0	9.86%

Tabla 8. Resultados del algoritmo genético para el caso sin turnos y tiempos de setup dependientes de la secuencia con datos de una instancia pequeña.

Para una muestra de 15 simulaciones del algoritmo, se obtiene

- **Media** = 2916 unidades.
- **Desviación típica** = 91,17 unidades.
- **Gap de optimalidad** en el rango [2.93, 11.29%]

El gap de optimalidad de la media es:

$$Gap\ de\ Optimalidad = \frac{2916 - 2729}{2729} = 6{,}85\%$$

Igual que en el caso genérico, se obtiene un gap de optimalidad medio menor al 10%. Teniendo en cuenta que el coste computacional para obtener la solución óptima es de 455 segundos, se podría decir que el algoritmo es capaz de encontrar soluciones de calidad y en ocasiones muy cercanas al óptimo en tiempos casi 7 veces menores.

6.2.4 LSJSP con tiempos de setup dependientes de la secuencia y turnos

Para el caso más complejo desde el punto de vista del modelado MILP, la **demanda** debe ser reducida aún más. Se consideran **50 unidades** para encontrar la solución óptima. **El número de lotes máximo se reduce a 2** por el mismo motivo**.** El makespan de la **solución óptima** es **1867 unidades**.

Los resultados obtenidos con el algoritmo genético se muestran en la Tabla 9.

CASO 4

- Con restricciones de turnos
- Tiempos de setup dependientes de la secuencia

Simulación	Coste computacional [s]	Mejor Fitness	Makespan	Penalty	Gap de optimalidad
1	62.09503	1867	1867	0	0.00% (min)
2	66.00067	2295	2295	0	22.92%
3	65.73206	2280	2280	0	22.12%
4	61.61803	2383	2383	0	27.64% (max)
5	63.57672	1867	1867	0	0.00% (min)
6	64.60103	1867	1867	0	0.00% (min)
7	68.72868	2280	2280	0	22.12%
8	63.27432	2383	2383	0	27.64% (max)
9	63.05866	2383	2383	0	27.64% (max)
10	62.82253	2295	2295	0	22.92%
11	63.44417	2295	2295	0	22.92%
12	63.72799	2280	2280	0	22.12%
13	73.64437	2347	2347	0	25.71%
14	69.48473	2280	2280	0	22.12%
15	64.30708	1867	1867	0	0.00% (min)

Tabla 9. Resultados del algoritmo genético para el caso con turnos y tiempos de setup dependientes de la secuencia con datos de una instancia pequeña.

Para una muestra de 15 simulaciones del algoritmo, se obtiene

- **Media** = 2197,93 unidades.
- **Desviación típica** = 3,25 unidades.

- **Gap de optimalidad** en el rango [0, 27,64%]

El gap de optimalidad de la media es:

$$Gap\ de\ Optimalidad = \frac{2197{,}93 - 1867}{1867} = 17{,}72\%$$

Se observa que en más de una simulación se llega al óptimo del problema. Sin embargo, en tantas otras el algoritmo no es capaz de seguir evolucionando. Está claro que hay una carencia en cuanto a la exploración del espacio de búsqueda. El espacio de este problema en concreto es "pequeño". El algoritmo en pocas iteraciones alcanza soluciones de alta calidad, pero en muchos casos se estanca. Las razones por las cuales sucede esto pueden ser diversas y requieren un análisis más profundo del algoritmo.

Dado que la instancia resuelta mediante el MILP solo contempla 2 lotes máximos por cada trabajo, surge la idea de probar el algoritmo con la misma instancia pero aumentando el **número de lotes máximos a 3 unidades.** Dado que no se conoce el óptimo del problema, se calcula el gap de optimalidad con respecto a la mejor solución encontrada.

Los resultados obtenidos con el algoritmo genético, incrementando en una unidad el número de lotes máximos, se muestran en la Tabla 10.

CASO 4* (Óptimo desconocido, Gap respecto a mejor solución conocida*)

- Con restricciones de turnos
- Tiempos de setup dependientes de la secuencia

Simulación	Coste computacional [s]	Mejor Fitness	Makespan	Penalty	Gap relativo*
1	72.15137	2280	2280	0	25.83%
2	69.80837	2115	2115	0	16.72%
3	72.30183	2151	2151	0	18.71%
4	73.22729	2040	2046	0	12.91%
5	71.24688	2046	2046	0	12.91%
6	71.20992	1812	1812	0	0.00% (min)
7	72.66207	1867	1867	0	3.04%
8	73.38208	1867	1867	0	3.04%
9	71.81634	1897	1897	0	4.69%
10	72.38923	2070	2070	0	14.24%
11	70.79691	2053	2053	0	13.30%
12	69.4062	1867	2190	0	20.86%
13	71.44682	2309	2309	0	27.43% (max)
14	70.88947	2115	2115	0	16.72%
15	68.9223	2175	2175	0	20.03%

Tabla 10. Resultados del algoritmo genético para el caso con turnos y tiempos de setup dependientes de la

secuencia con datos de una instancia pequeña. Aumento del número de lotes máximos a 3 unidades.

Para una muestra de 15 simulaciones del algoritmo, se obtiene

- **Media** = 2044,27 unidades.
- **Desviación típica** = 154,65 unidades.
- **Máximo Gap relativo respecto a la mejor solución encontrada** = 27,43%]

El gap de optimalidad particular de la media es:

$$Gap\ relativo = \frac{2044{,}27 - 1812}{1812} = 12{,}82\%$$

Considerando un lote más para cada trabajo se ha conseguido mejorar la solución óptima obtenida (en un tiempo razonable) con el modelo MILP. Además, la media de soluciones obtenidas es menor. Con esto se refuerza la idea de que, aunque el algoritmo no sea capaz de darnos la mejor solución sí que es capaz de devolver soluciones de alta calidad gracias a la habilidad para manejar instancias de mayor tamaño.

6.3 Evolución en instancias grandes

Una de las limitaciones que presentan los modelos MILP propuestos es que no es posible obtener soluciones óptimas ni siquiera factibles cuando el tamaño de las instancias crece. La limitación es aún más notable cuando se consideran restricciones de turnos y/o tiempos de setup dependientes de la secuencia. Se pierde capacidad para manejar espacios de soluciones más grandes. Esta limitación se supera con el algoritmo genético propuesto.

En el apartado anterior, se obtuvo un mayor gap de optimalidad para los dos casos que incluyen restricciones de turnos. Por ello, resultaría interesante conocer el comportamiento del algoritmo en una instancia más grande y permitiendo muchas más generaciones y por lo tanto las oportunidades de mejorar. Además, es obvio que si el algoritmo se comporta bien considerando esta restricción también lo hará cuando no se considere, pues se restringe menos el espacio de búsqueda. Hay menos zonas "prohibidas" y por lo tanto es más complicado estancarse en mínimos locales.

Para esta experimentación se han usado los parámetros de la instancia "grande" ofrecidos en la Tabla 5. Se han usado los siguientes hiperparámetros:

- **Tamaño de la población** = 200.
- **Número de generaciones** = 1000

- **Demanda** unitaria de cada trabajo = 500.
- El **umbral de diversificación** aumenta de 1 unidad en 1 unidad, en vez de aumentar de 10 en 10. Esto se ha modificado porque se conoce de antemano que al algoritmo le cuesta mejorar cuando encuentra soluciones de calidad y porque el número de generaciones es elevado. Para fomentar aún más la diversificación, se disminuye la tasa de actualización del umbral.
- La **probabilidad de mutación** se aumenta a 0.3 por el mismo motivo, aumentar la diversificación.

Adicionalmente **se ha aumentado el tiempo que ocupa un turno a 1440 unidades** (si fuesen minutos, serían 24 horas).

Para **garantizar encontrar soluciones factibles**, donde se cumple la restricción de turnos es fundamental asegurarse de qué toda la demanda se puede distribuir en lotes cuyo tiempo total no excede el tiempo de un turno. Para ello se puede hacer una comprobación simple:

$$\frac{d_j}{número\ de\ lotes\ max} \cdot \max\left(p'_{mj}\right) \cdot \max\left(s'_{m^* j^* k}\right) \leq SHT$$

Donde:

- d_j es la demanda asignada a cada trabajo (igual para todos los trabajos en este caso)
- $\max\left(p'_{mj}\right)$ es el mayor tiempo de proceso unitario en la matriz de tiempos de proceso
- $\max\left(s'_{m^* j^* k}\right)$ es el mayor tiempo de setup correspondiente al trabajo j^* y la máquina m^* del mayor tiempo de proceso unitario. Aquí se expresa para tiempos de setup dependientes de la secuencia pero es igualmente válido para el caso de tiempos independientes.

Los resultados obtenidos son:

1. **Caso 2. Restricciones de turnos y tiempos de setup independientes de la secuencia.**

Tras 1000 generaciones, el fitness obtenido es mucho menor que el inicial. Como ya se ha explicado, este fitness está compuesto por el valor del makespan y una penalización a modo de restricción. En la mejor solución, el valor de la penalización es 0 y por lo tanto se satisface la restricción de turnos. En concreto se obtiene:

- **Makespan =** 49945.
- **Penalty** = 0.
- **Coste computacional =** 4882,57 s.

La evolución de la solución se muestra en la Figura 24.

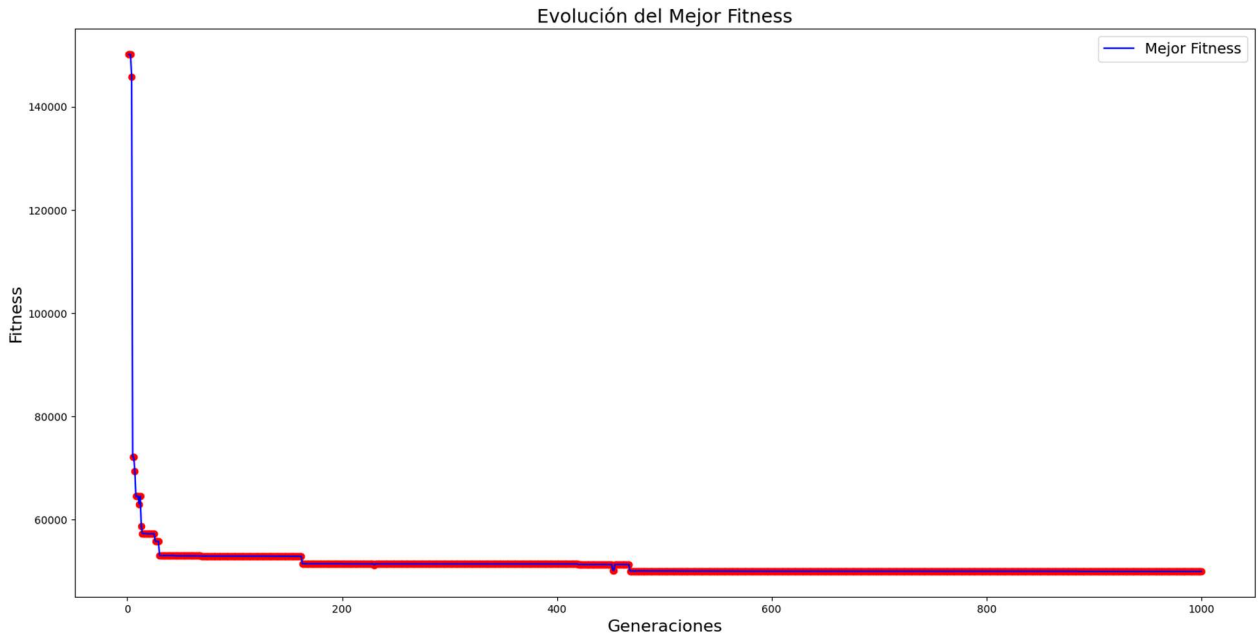

Figura 24. Evolución del mejor fitness en una instancia grande considerando restricción de turnos y tiempos de setup independientes de la secuencia. [Fuente: elaboración propia].

Se observa que en las primeras generaciones el algoritmo mejora notablemente, reduciendo el fitness a menos de la mitad del fitness inicial. Esta rápida evolución se debe al cumplimiento de la restricción de turnos, la cual está priorizada por encima del makespan.

Más allá de la rápida evolución en las primeras iteraciones, el algoritmo se estanca rápidamente y necesita muchas generaciones para realizar pequeñas mejoras. Está claro que existe una carencia de mecanismos que, mediante diversificación o intensificación, mejoren las soluciones encontradas.

El diagrama de Gantt obtenido se muestra en Figura 25. A simple vista es complicado obtener información de él, dado el tamaño y la cantidad de información que presenta. Sin embargo, es muy útil para hacerse una idea de la complejidad del puzzle que se está intentando resolver. Un puzzle que no consiste solo en poner las piezas en el sitio correcto sino también en darle un tamaño adecuado a esas piezas.

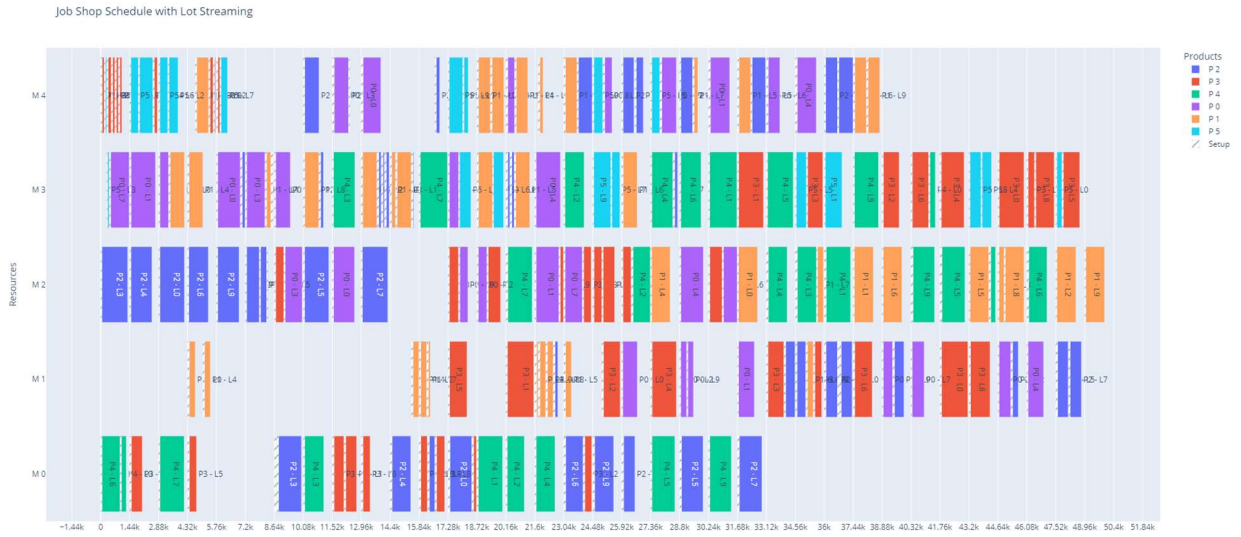

Figura 25. Diagrama de Gantt obtenido para la instancia grande tras 1000 generaciones considerando restricciones de turnos y tiempos de setup independientes de la secuencia. [Fuente: elaboración propia].

2. **Caso 4. Restricciones de turnos y tiempos de setup dependientes de la secuencia.**

En este caso se obtienen los siguientes resultados:

- **Makespan** = 47216.
- **Penalty** = 0.
- **Coste computacional** = 4990,24 s.

Análogamente al caso anterior se cumple estrictamente con la restricción de turnos, obteniendo una penalización nula. El valor del makespan obtenido es menor. Evidentemente, la naturaleza de los setups permite ahorrar tiempo con el procesando lotes pertenecientes al mismo trabajo en serie, uno detrás de otro en la misma máquina, y por lo tanto omitiendo la preparación en esa máquina en cuestión.

En instancias más pequeñas, se observa una tendencia muy fuerte a que todos los lotes del mismo trabajo se procesen en serie. Esto es porque los tiempos de setup son más relevantes en el makespan cuando la demanda es pequeña. En este caso, con una demanda más elevada, los tiempos de setup no tienen ese efecto tan notable en el makespan y por lo tanto se observa, que a pesar de que muchos lotes del mismo trabajo se procesan en serie, se intercalan lotes de otros trabajos también.

La evolución del mejor fitness en esta simulación se muestra en la Figura 26. En las primeras generaciones, la evolución es bastante rápida y de nuevo se asocia con el cumplimiento de la restricción de turnos. Después la

evolución es más lenta y conforme disminuye el fitness, se necesitan más generaciones para mejorar. Aquí la curva de evolución es más satisfactoria que en el caso anterior. Aunque se observa que la mejora no es sencilla, se producen muchas más mejoras a lo largo de las 1000 generaciones. Especialmente, en las primeras 250 generaciones, la mejora es más dinámica y no hay períodos largos de estancamiento.

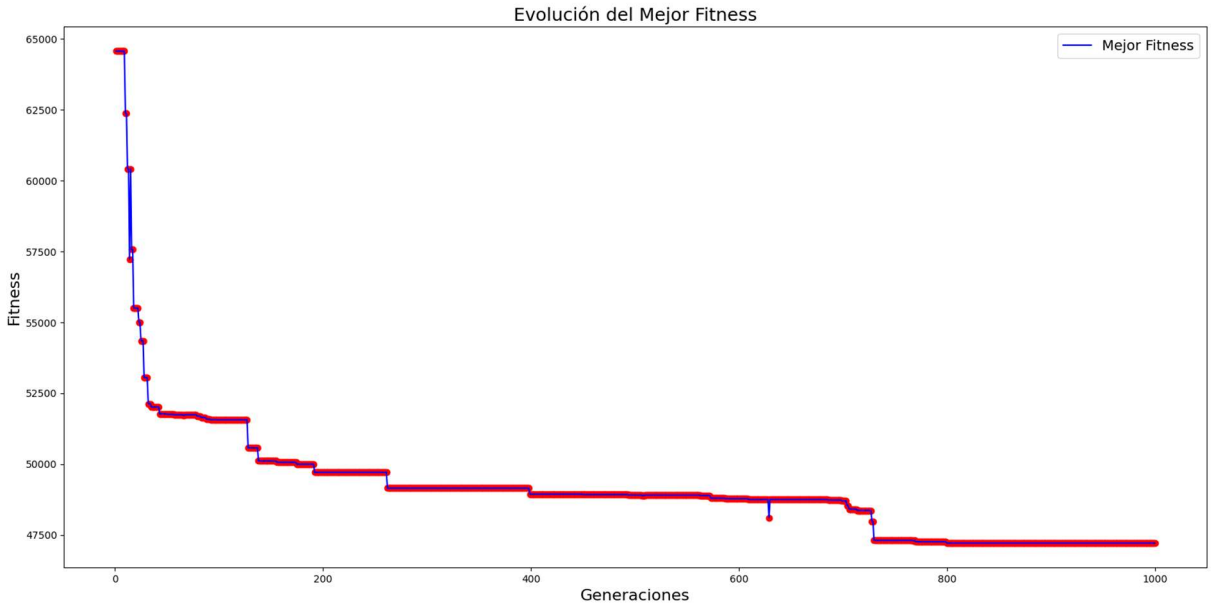

Figura 26. Evolución del mejor fitness en una instancia grande considerando restricción de turnos y tiempos de setup dependientes de la secuencia. [Fuente: elaboración propia].

Por complementar el caso anterior y reforzar la visión de la complejidad del problema, se muestra el diagrama de Gantt obtenido en la Figura 27.

Figura 27. Diagrama de Gantt obtenido para la instancia grande tras 1000 generaciones considerando restricciones de turnos y tiempos de setup dependientes de la secuencia. [Fuente: elaboración propia].

Obviamente, estos diagramas tal y como se ilustran aquí en formato fotografía son poco útiles en el día a día en la producción, pues es tanta la información que contienen y tan detallada que es complicado trabajar con ello. Sin embargo, los archivos html generados con los diagramas de Gantt permiten visualizar la información de una determinada región del diagrama así como los datos de cada lote concreto (tamaño, producto e instantes de inicio y fin). Por ejemplo, se ilustra en la Figura 28 un horizonte temporal de 7 días (asumiendo que la unidad de tiempo son minutos) extraído de este caso.

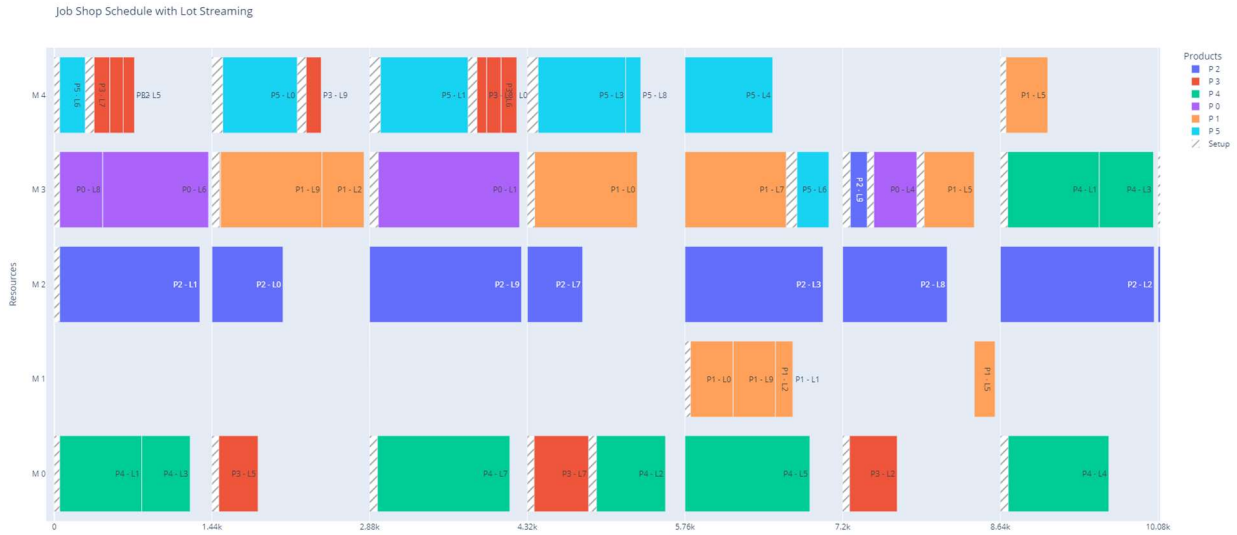

Figura 28. Extracto temporal de un diagrama de Gantt generado con *plotly.express.timeline*. [Fuente: elaboración propia].

6.4 Conclusiones de la experimentación

En el presente capítulo se ha evaluado el rendimiento del algoritmo genético implementado. Gracias a los modelos matemáticos desarrollados se ha podido obtener la solución óptima de una instancia pequeña para cada modelo y medir el gap de optimalidad de las soluciones que se obtienen con el algoritmo genético. Además se ha evaluado el rendimiento del algoritmo con instancias de mayor tamaño para los casos más complejos: aquellos que incluyen restricciones de turnos.

En la Tabla 11 se ofrece un resumen de los resultados obtenidos en la experimentación.

VALIDACIÓN Y GAP DE OPTIMALIDAD

Tamaño de la población	150	**Número de generaciones**	150

	GAP DE OPTIMALIDAD [%]			FITNESS				
	Media	Max	Min	Óptimo	Media	Sd	Demanda	Nº lotes max
CASO 1	6,81	11,54	0	5380	5746,6	164,67	200	3
CASO 2	49,82	72,76	34,96	3304	4950,2	375,07	100	5
CASO 3	6,85	11,29	2,93	2729	2916	91,17	100	3
CASO 4	17,72	27,64	0	1867	2197,93	3,25	50	2
CASO 4 (*)	12,82	27,43	0	1812	2044,27	154,65	50	3

RENDIMIENTO EN UNA INSTANCIA GRANDE

Tamaño de la población	200	**Número de generaciones**	1000

	Makespan	Penalty	Coste computacional [s]	Demanda	Nº lotes max
CASO 2	49945	0	4882,57	500	10
CASO 4	47216	0	4990,24	500	10

Tabla 11. Resumen de los resultados obtenidos en la experimentación.

Atendiendo a los resultados obtenidos, se observa que para una instancia pequeña obtiene un programa de producción cercano al óptimo cuando no se consideran restricciones con turnos. En los casos 2 y 4, en los que sí se consideran restricciones temporales relacionadas con el tamaño de los lotes, los resultados no son tan buenos.

Especialmente en el caso 2, donde se consideran turnos y tiempos de setup independientes de la secuencia, el

rendimiento del algoritmo deja mucho que desear. El gap de optimalidad medio es bastante grande comparado con el obtenido en los demás casos. Evaluando el rendimiento para este caso con una instancia más grande y observando la evolución de la mejor solución encontrada, se observa la carencia de mecanismos para mejorar. Apenas unas pocas generaciones después de empezar, el algoritmo necesita muchas generaciones para hacer pequeñas mejoras.

No obstante, con la implementación del algoritmo se supera una de las principales limitaciones de las técnicas de resolución basadas en modelos matemáticos: la capacidad de obtener soluciones factibles y de calidad para instancias de gran tamaño. Al resolver una instancia de gran envergadura, cuyo coste computacional necesario para encontrar una solución factible es desconocido (en modelos con turnos), el algoritmo genera soluciones razonables, con una calidad mucho mejor que la inicial. En el caso 4, a diferencia del caso 2, se evidencia cómo el algoritmo evoluciona de manera progresiva hacia soluciones cada vez mejores con relativa facilidad en las primeras 400 generaciones.

Considerando estos resultados y el comportamiento observado del algoritmo, se puede afirmar que el algoritmo es capaz de encontrar soluciones factibles en todos los casos y mejorarlas, llegando en algunas ocasiones a soluciones óptimas o muy cercanas al óptimo. Por otro lado, no se puede ignorar que en los casos con restricciones de turnos y especialmente cuando los tiempos de setup son independientes de la secuencia, el algoritmo carece de mecanismos de intensificación y diversificación para evolucionar hacia soluciones de mayor calidad. Sería necesario un análisis más detallado de los hiperparámetros y los mecanismos propios del algoritmo genético para sacar más conclusiones.

7 CONCLUSIONES

Con el presente proyecto se ha conseguido cumplir con el objetivo principal propuesto: desarrollar una metaheurística capaz de aportar soluciones para instancias grandes del problema LSJSP considerando restricciones de turnos y tiempos de setup dependientes de la secuencia, en un tiempo razonable.

La revisión de la literatura ha sido fundamental para entender mejor el problema de lotificación y secuenciación de trabajos conjunta, así como para obtener ideas y estrategias útiles para el diseño e implementación del algoritmo genético.

Gracias al modelado e implementación de varios modelos matemáticos de programación lineal entera mixta, se ha podido evaluar el rendimiento del algoritmo mediante el gap de optimalidad. Esta validación junto con la experimentación con una instancia de gran tamaño ha permitido conocer bondades, carencias y consecuentemente áreas de mejora. Se ha demostrado que el algoritmo genético desarrollado es capaz de evaluar y explorar el espacio de soluciones lo necesario para encontrar soluciones factibles cuya calidad es notable respecto a la calidad de las soluciones iniciales.

El grado de cumplimiento de los objetivos específicos establecidos al inicio del proyecto es satisfactorio. Se ha explicado en qué consiste el problema, se han revisado técnicas, metaheurísticas y estrategias para solucionarlo. Se ha hecho hincapié en el algoritmo genético por ser este el más común en la literatura y una de las metaheurísticas más flexibles y fácilmente integrables con otras técnicas Se han modelado matemáticamente todos los escenarios que se querían resolver y se han implementado en Python para obtener la solución óptima del problema en una instancia pequeña. Se ha explicado en profundidad el diseño e implementación del algoritmo genético para resolver el problema, así como el diseño, codificación, decodificación y evaluación de las soluciones. Finalmente se ha probado y validado el algoritmo en instancias de diferentes tamaños. Con todo, se han podido extraer conclusiones valiosas. Se facilita de este modo una base sólida sobre la que trabajar en futuras mejoras.

Este proyecto no pretende ofrecer el estudio detallado de los hiperparámetros y el comportamiento del algoritmo genético en cuestión, sino aportar una base sólida de teoría, modelado y código para resolver el problema LSJSP con restricciones de distinta naturaleza. Un trampolín para adentrarse en este pequeño nicho de la programación de la producción y desarrollar técnicas de optimización más potentes y escalables, aplicables en escenarios reales y específicos.

REFERENCIAS

[1] F. M. Vallejo Gómez de Travecedo, "Modelado MILP para la determinación conjunta del scheduling y lot-streaming en un entorno jobshop," Sevilla, 2024.

[2] D. Rooyani and F. Defersha, "A Two-Stage Multi-Objective Genetic Algorithm for a Flexible Job Shop Scheduling Problem with Lot Streaming," *Algorithms*, vol. 15, no. 7, 2022, doi: 10.3390/a15070246.

[3] F. T. S. Chan, T. C. Wong, and L. Y. Chan, "The application of genetic algorithms to lot streaming in a job-shop scheduling problem," *Int J Prod Res*, vol. 47, no. 12, 2009, doi: 10.1080/00207540701577369.

[4] F. M. Defersha and S. Bayat Movahed, "Linear programming assisted (not embedded) genetic algorithm for flexible jobshop scheduling with lot streaming," *Comput Ind Eng*, vol. 117, 2018, doi: 10.1016/j.cie.2018.02.010.

[5] A. Salazar-Moya and M. V. Garcia, "Lot streaming in different types of production processes: A prisma systematic review," *Designs (Basel)*, vol. 5, no. 4, 2021, doi: 10.3390/designs5040067.

[6] D. Lei and X. Guo, "Scheduling job shop with lot streaming and transportation through a modified artificial bee colony," *Int J Prod Res*, vol. 51, no. 16, 2013, doi: 10.1080/00207543.2013.784404.

[7] F. Xie, L. Li, L. Li, Y. Huang, and Z. He, "A decomposition-based multi-objective Jaya algorithm for lot-streaming job shop scheduling with variable sublots and intermingling setting," *Expert Syst Appl*, vol. 228, 2023, doi: 10.1016/j.eswa.2023.120402.

[8] C. Low, C. M. Hsu, and K. I. Huang, "Benefits of lot splitting in job-shop scheduling," *International Journal of Advanced Manufacturing Technology*, vol. 24, no. 9–10, 2004, doi: 10.1007/s00170-003-1785-9.

[9] S. Dauzère-Pérès and J. B. Lasserre, "Lot Streaming in job-shop scheduling," *Oper Res*, vol. 45, no. 4, 1997, doi: 10.1287/opre.45.4.584.

[10] J. M. Novas, "Production scheduling and lot streaming at flexible job-shops environments using constraint programming," *Comput Ind Eng*, vol. 136, 2019, doi: 10.1016/j.cie.2019.07.011.

[11] A. Bożek and F. Werner, "Flexible job shop scheduling with lot streaming and sublot size optimisation," *Int J Prod Res*, vol. 56, no. 19, 2018, doi: 10.1080/00207543.2017.1346322.

[12] S. Dauzère-Péres and J. B. Lasserre, "Integration of lotsizing and scheduling decisions in a job-shop," *Eur J Oper Res*, vol. 75, no. 2, 1994, doi: 10.1016/0377-2217(94)90085-X.

[13] S. Dauzere-Peres and J. B. Lasserre, "An iterative procedure for lot streaming in job-shop scheduling," *Comput Ind Eng*, vol. 25, no. 1–4, 1993, doi: 10.1016/0360-8352(93)90263-W.

[14] J. H. Holland, "Outline for a Logical Theory of Adaptive Systems," *Journal of the ACM (JACM)*, vol. 9, no. 3, 1962, doi: 10.1145/321127.321128.

[15] Ramón Garduño Juárez, "Conogasi," https://conogasi.org/articulos/algoritmos-geneticos/.

[16] H. Liu, L. S. Chen, and P. S. Lin, "Lot streaming multiple jobs with values exponentially deteriorating over time in a job-shop environment," *Int J Prod Res*, vol. 51, no. 1, 2013, doi: 10.1080/00207543.2012.657255.

[17] F. M. Defersha and M. Chen, "Jobshop lot streaming with routing flexibility, sequence-dependent setups, machine release dates and lag time," *Int J Prod Res*, vol. 50, no. 8, 2012, doi: 10.1080/00207543.2011.574952.

[18] J. Fan, C. Zhang, W. Shen, and L. Gao, "A matheuristic for flexible job shop scheduling problem with lot-streaming and machine reconfigurations," *Int J Prod Res*, vol. 61, no. 19, 2023, doi: 10.1080/00207543.2022.2135629.

[19] M. A. Salido, J. Escamilla, F. Barber, A. Giret, D. Tang, and M. Dai, "Energy-aware Parameters in Job-shop Scheduling Problems," in *IJCAI 2013 Workshop on Constraint Reasoning, Planning and Scheduling Problems for a Sustainable Future*, 2013.

[20] J. Para, J. Del Ser, and A. J. Nebro, "Energy-Aware Multi-Objective Job Shop Scheduling Optimization with Metaheuristics in Manufacturing Industries: A Critical Survey, Results, and Perspectives," 2022. doi: 10.3390/app12031491.

[21] Y. Sato and M. Sato, "Using Dominated Solutions at Edges to the Diversity and the Uniformity of Non-dominated Solution Distributions in NSGA-II," *SN Comput Sci*, vol. 3, no. 6, 2022, doi: 10.1007/s42979-022-01303-w.

[22] A. S. Manne, "On the Job-Shop Scheduling Problem," *Oper Res*, vol. 8, no. 2, 1960, doi: 10.1287/opre.8.2.219.

[23] W. Y. Ku and J. C. Beck, "Mixed Integer Programming models for job shop scheduling: A computational analysis," *Comput Oper Res*, vol. 73, 2016, doi: 10.1016/j.cor.2016.04.006.

[24] A. Sprecher, R. Kolisch, and A. Drexl, "Semi-active, active, and non-delay schedules for the resource-constrained project scheduling problem," *Eur J Oper Res*, vol. 80, no. 1, 1995, doi: 10.1016/0377-2217(93)E0294-8.

[25] "Operadores Genéticos." Accessed: Sep. 23, 2024. [Online]. Available: https://creationwiki.org/Genetic_algorithm