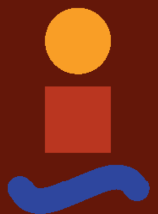
Trabajo Fin de Grado
Ingeniería de Organización Industrial

Programación de la producción en entornos sanitarios: resolución mediante colonias de abejas artificiales

Autor: Isabel Cámara Flores
Tutor: Víctor Fernández-Viagas Escudero
Tutor externo: José Manuel Molina Pariente

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



Sevilla, 2020

Trabajo Fin de Grado
Ingeniería de Organización Industrial

Programación de la producción en entornos sanitarios: resolución mediante colonias de abejas artificiales

Autor:
Isabel Cámara Flores

Tutores:
Víctor Fernández-Viagas Escudero
Profesor Contratado Doctor

José Manuel Molina Pariente
Investigador Postdoctoral

Dpto. de Organización Industrial y Gestión de Empresas I
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla
Sevilla, 2020

Trabajo Fin de Grado: Programación de la producción en entornos sanitarios: resolución mediante colonias de abejas artificiales

Autor: Isabel Cámara Flores

Tutores: Víctor Fernández-Viagas Escudero. José Manuel Molina Pariente

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

Presidente:

Vocales:

Secretario:

Acuerdan otorgarle la calificación de:

Sevilla, 2020

El Secretario del Tribunal

AGRADECIMIENTOS

En primer lugar, me gustaría agradecer a mi tutor, Víctor Fernández-Viagas Escudero, su dedicación a pesar de las adversidades que ha provocado el estado de alarma. Aunque esta situación incrementó su trabajo como docente, disponer de su tiempo siempre que lo he necesitado ha sido crucial. También me gustaría reconocer el esfuerzo de los profesores que, con paciencia, han respondido a todas mis inquietudes y me han inculcado la pasión por la Organización Industrial.

Sin embargo, sin el apoyo incondicional de mis familiares, en especial de mi madre y de mi abuela, todo hubiera sido más complicado. Mamá, gracias por el sacrificio. Eres mi fuente de inspiración y mereces todo el éxito que consiga. A ti abuela, te agradezco los valores que me transmitiste, pero sobre todo la confianza que depositaste en mí. Sin haber podido verme terminar, sabías que iba a conseguirlo y te fuiste orgullosa.

Por último, me gustaría destacar el papel que han tenido en este camino mis amigos, quienes me han ayudado a desconectar en situaciones de colapso que servían de impulso para continuar. En especial a María, Irene y Nazaret, quienes además de esto, han compartido esta experiencia conmigo, haciendo que todos los momentos duros hayan merecido la pena. Para mí sois mucho más que unas compañeras de clase.

Isabel Cámara Flores
Sevilla, 2020

RESUMEN

El problema de decisión abordado trata de modelar y analizar cuál es el mejor algoritmo, entre cinco propuestas, para establecer la planificación de las operaciones pendientes de la especialidad de Cirugía Plástica y Quemaduras Mayores del Hospital Universitario "Virgen del Rocío" (Sevilla) durante un horizonte temporal. Se aplican cinco versiones de la metaheurística bio-inspirada *"Artificial Bee Colony"* a un conjunto de instancias que simulan diversos escenarios. Se usa una adaptación de la versión básica y cuatro modificaciones de esta, lo que nos permite observar la influencia de las fases del algoritmo en la programación de los quirófanos.

En cada una de las propuestas, se pretende establecer una programación de las operaciones de calidad que cumpla con las restricciones en un tiempo computacional razonable. El objetivo principal consiste en operar con mayor brevedad a los pacientes más graves para reducir el retraso de cada intervención respecto a su fecha límite. Así, se consigue minimizar la tardanza ponderada en función de la prioridad de cada paciente que se opera. Adicionalmente, se añade un componente a la Función Objetivo para mejorar la solución en caso de que dos programas tengan la misma tardanza. En particular, como componente secundario de la Función Objetivo se intenta reducir el número de pacientes que no se operan en el intervalo que se programa debido a que el día de operación establecido sobrepasa el horizonte de planificación. De esta forma, se logra operar al mayor número de pacientes posibles, en caso de que la tardanza no pueda ser reducida más.

ÍNDICE

Agradecimientos ..... *vii*
Resumen ..... *ix*
índice ..... *xi*
Índice de Figuras ..... *xiii*
Índice de Tablas ..... *xv*
1 Introducción *1*
1.1 Programación de la producción **1**
1.2 Algoritmos aproximados **4**
1.3 Programación de la producción en el sector sanitario **5**
2 Descripción del problema *7*
2.1 Restricciones **7**
2.2 Objetivo **8**
3 Metodología *9*
3.1 Determinación del horario quirúrgico **9**
3.2 Ordenación lista de espera **10**
3.3 ABC Básico **12**
3.4 Mejora 1 ..... **16**
3.5 Mejora 2 ..... **20**
3.6 Mejora 3 ..... **21**
3.7 Mejora 4 ..... **22**
4 Experimentación *25*
4.1 Generación de las instancias del problema **25**
4.2 Determinación de los parámetros del algoritmo **27**
4.3 Evaluación de las alternativas **29**
5 Conclusiones *33*
6 Bibliografía *35*
7 Anexos *37*
7.1 Código función principal **37**
7.2 Código funciones auxiliares **39**

ÍNDICE DE FIGURAS

Figura 1. Niveles toma de decisiones (*Entender ITIL 2011 Normas y mejores prácticas para avanzar hacia ISO 20000 - Los niveles decisionales*, n.d.) 1
Figura 2. Diagrama de Gantt (Framinan et al., 2014) 2
Figura 3. Solución de un problema de programación (Framinan et al., 2014) 3
Figura 4. Flujo de trabajo máquinas paralelas (Framinan et al., 2014) 4
Figura 5. Intercambio de información en algoritmo de inteligencia colectiva (Ioannou, 2017) 5
Figura 6. Ejemplo solución 20 trabajos y 3 máquinas paralelas (Framinan et al., 2014) 9
Figura 7. Diferencia exploración y explotación (*Metaheurísticas | Peralta-Abarca | Inventio, la génesis de la cultura universitaria en Morelos*, n.d.) 11
Figura 8. Pseudocódigo algoritmo Artifical Bee Colony (Elaboración propia) 12
Figura 9. Pseudocódigo inicialización población algoritmo ABC básico (Elaboración propia) .. 12
Figura 10. Inserción aleatoria (Elaboración propia) 13
Figura 11. Intercambio adyacente (Elaboración propia) 13
Figura 12. Intercambio de pares (Elaboración propia) 14
Figura 13. Pseudocódigo fase de la abeja obrera algoritmo ABC básico 14
Figura 14. Pseudocódigo fase abeja observadora algoritmo ABC básico (Elaboración propia) .. 15
Figura 15. Pseudocódigo fase abeja exploradora algoritmo ABC básico (Elaboración propia) .. 15
Figura 16. Pseudocódigo NEH (Pan et al., 2014) 16
Figura 17. Pseudocódigo inicialización población mejora 1 (Elaboración propia) 17
Figura 18. Pseudocódigo fase abeja obrera mejora 1 (Elaboración propia) 19
Figura 19. Pseudocódigo fase abeja observadora mejora 1 (Elaboración propia) 19
Figura 20. Pseudocódigo fase abeja exploradora mejora 1 (Elaboración propia) 20
Figura 21. Pseudocódigo fase abeja exploradora mejora 2 (Elaboración propia) 21
Figura 22. Pseudocódigo fase abeja obrera mejora 3 (Elaboración propia) 22
Figura 23. Pseudocódigo fase abeja observadora mejora 3 (Elaboración propia) 22
Figura 24. Pseudocódigo fase abeja obrera mejora 4 (Elaboración propia) 23
Figura 25. Pseudocódigo fase abeja observadora mejora 4 (Elaboración propia) 23

ÍNDICE DE TABLAS

Tabla 1.Valores destacables de RDI de cada porcentaje de empeoramiento 28
Tabla 2. Valores destacables RDI de cada porcentaje de empeoramiento (Segunda calibración) 29
Tabla 3. Valores destacables de RPD de cada algoritmo 30
Tabla 4. Valores destacables de RDI de cada algoritmo 31
Tabla 5. Valores destacables de RPD de cada algoritmo (Segunda experimentación) 32
Tabla 6. Valores destacables de RDI de cada algoritmo (Segunda experimentación) 32

1 INTRODUCCIÓN

“Lo que caracteriza al hombre de ciencia no es la posesión del conocimiento o de verdades irrefutables, sino la búsqueda desinteresada e incesante de la verdad”
- Karl Popper -

1.1 Programación de la producción

La gestión de la producción es un proceso fundamental de toma de decisiones en los sistemas de fabricación y producción, así como en la industria de servicio, para garantizar la entrega de bienes con la máxima calidad al mínimo coste y tiempo. Como se muestra en la Figura 1, estas decisiones están jerarquizadas en niveles que deben coordinarse (Cardoen et al., 2010).

En el nivel estratégico se determinan los propósitos y las metas de la organización, por ejemplo, entrar o salir de los mercados. Estas decisiones son imposibles de revisar en un horizonte que no sea menor a un año, por eso, se consideran decisiones a largo plazo que necesitan establecerse teniendo en cuenta la evolución del mercado y los recursos internos de la empresa. Posteriormente se toman las decisiones a nivel táctico o a medio plazo, donde se diseña la estructura para implementar las decisiones estratégicas.

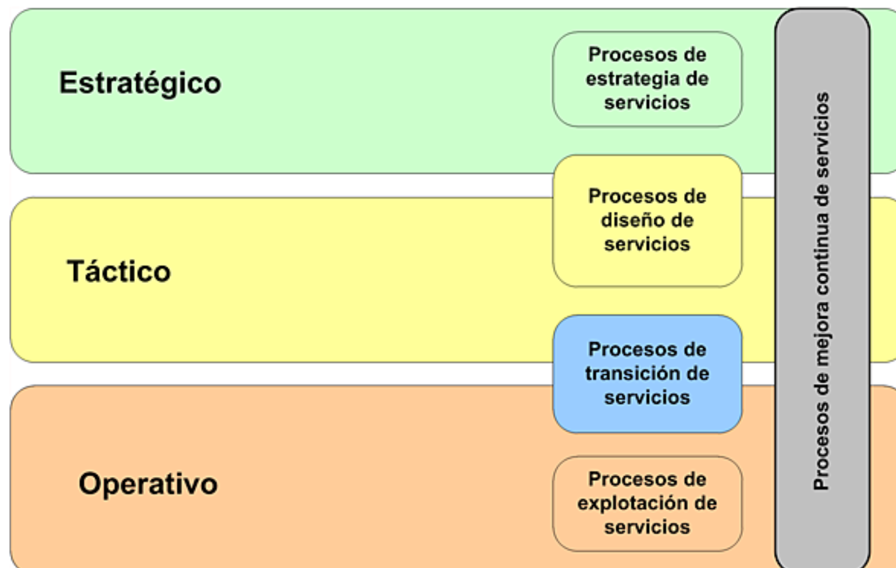


Figura 1. Niveles toma de decisiones (*Entender ITIL 2011 Normas y mejores prácticas para avanzar hacia ISO 20000 - Los niveles decisionales*, n.d.)

Finalmente, en el nivel operativo o a corto plazo se definen los procedimientos de las actividades resultantes de la planificación táctica y se programa la producción. Aunque en el pasado no formaba parte de las decisiones básicas de la gestión, es en estos procesos donde la empresa obtiene el beneficio gracias al valor añadido que poseen, por lo que en la actualidad es cada vez

más frecuente optimizarlos en función a unos criterios. A diferencia de la producción basada en la economía de escala del pasado, para optimizar la denominada producción ajustada del presente, que precisa una mayor flexibilidad en la gama y los volúmenes, es necesario tener en cuenta las características de las tareas y de los recursos. Solo así se consigue procesar un conjunto de trabajos haciendo un uso eficiente de los recursos necesarios. Generalmente, cada tarea o grupo de tareas tiene unas características específicas: tiempo de procesamiento, prioridad, fecha en la que como mínimo puede iniciarse o como máximo puede completarse, etc. Los recursos, que pueden ir desde máquinas en un taller hasta personas del equipo de trabajo, también presentan particularidades: especialización, velocidad, capacidad, etc.

Los criterios de optimización pueden ser regulares, si son función no decreciente de los tiempos de finalización de las tareas, o no regulares en caso contrario. Un ejemplo de criterio regular sería minimizar el tiempo medio de finalización de las tareas, y uno no regular, reducir al máximo el tiempo medio que las tareas han sobrepasado la fecha de vencimiento.

El resultado de la programación es un cronograma detallado que ayuda a mantener la planificación y el control de las operaciones para constituir un buen servicio al cliente. En él se determinan dónde y cuándo debe comenzar cada trabajo, es decir, se define un programa. En numerosos problemas, especialmente de criterio regular, la solución óptima es un programa semiactivo en el que cada operación se realiza tan pronto como sea posible, por tanto, las tareas no podrán ser completadas con anterioridad sin cambiar el orden de procesamiento. Tradicionalmente, el cronograma se representa con un diagrama de Gantt como el de la Figura 2. El eje horizontal siempre representa la escala de tiempo, pero las posibles representaciones del eje vertical dan lugar a dos tipos distintos. En el caso en el que el eje vertical represente las máquinas donde se realizan cada uno de los trabajos, el Gantt será de tipo orientado a máquinas. Si en cambio se representan los trabajos que se realizan, será un diagrama de Gantt orientado a trabajos.

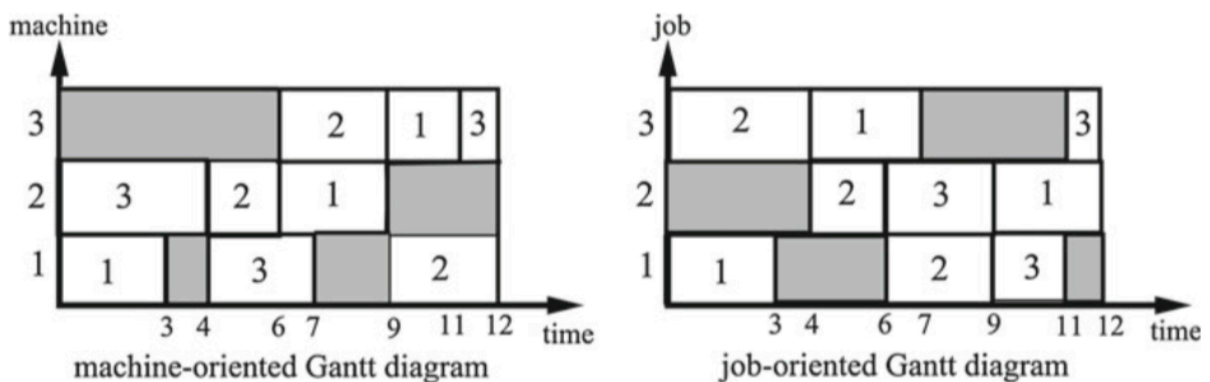


Figura 2. Diagrama de Gantt (Framinan et al., 2014)

Para determinar una buena solución de un problema de programación en la que la asignación de los recursos se haga de forma eficiente, el tomador de decisiones suele usar herramientas de apoyo a la decisión. Estas herramientas aplican un método (algoritmo) para resolver, en primer lugar, un modelo que recoge de forma simplificada las distintas instancias o situaciones que puedan plantearse. Un modelo de programación es una abstracción formal de un problema de decisión de programación generado teniendo en cuenta unas tareas, restricciones y criterios específicos. En concreto, un modelo está definido por un conjunto de trabajos finito y conocido que deben procesarse en un conjunto de máquinas que también se conoce. En principio, a cada trabajo hay que realizarle una serie de operaciones determinadas cuyo orden está establecido mediante una ruta de procesamiento. Además, cada una de estas tareas suele presentar un tiempo de

procesamiento, una fecha de lanzamiento o instante a partir del cual se puede comenzar a procesar, una fecha de entrega en la que tendría que estar idealmente terminada, y una prioridad o importancia. Tras desarrollar el algoritmo que resuelve el modelo, se implementa en la situación específica con los datos reales y se obtiene la solución del problema de programación en cuestión. El resumen de este proceso se muestra en la Figura 3.

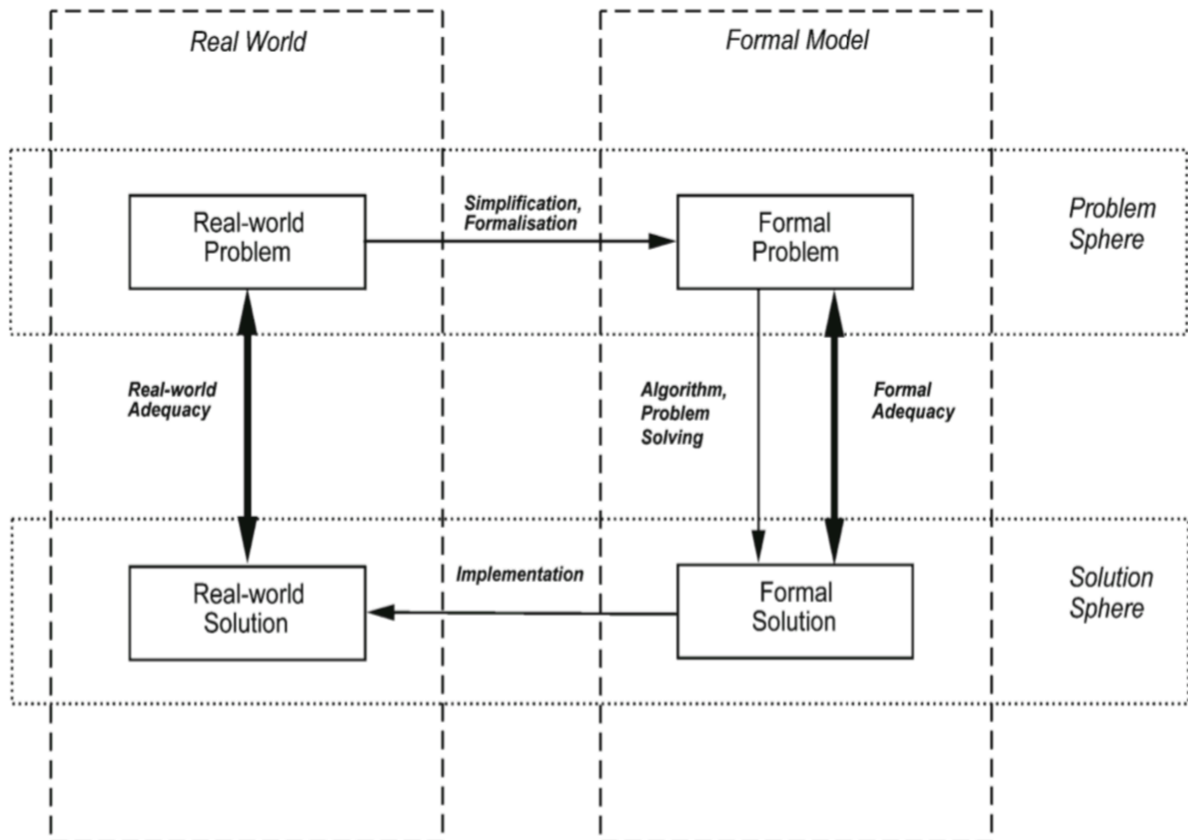


Figura 3. Solución de un problema de programación (Framinan et al., 2014)

Este Trabajo Fin de Grado está centrado en el desarrollo de cinco algoritmos que resuelven un modelo para establecer la programación de los quirófanos de un hospital. Se pretende determinar cuál es el más adecuado en varias instancias o situaciones que pudiesen plantearse, pero no se llegará a implementar con los datos reales para comprobar el efecto que finalmente tiene en la situación real.

Aunque se trate de un proceso de decisión abierto, porque tiene relación con otros procesos anteriores, en el modelo se descarta el preoperatorio y el postoperatorio y se visualiza como un problema de decisión cerrado para centrarnos únicamente en los quirófanos. En el caso de la programación de los quirófanos, las máquinas o quirófanos multifuncionales son idénticos y trabajan de forma paralela. Por ello, las rutas de procesamiento para los trabajos son muy sencillas. Como se muestra en la Figura 4, cada trabajo (paciente en el contexto actual) tiene que ser procesado una única vez en alguna de las máquinas disponibles (quirófanos). Las restricciones y los criterios se describen posteriormente en la sección 2.1 y 2.2.

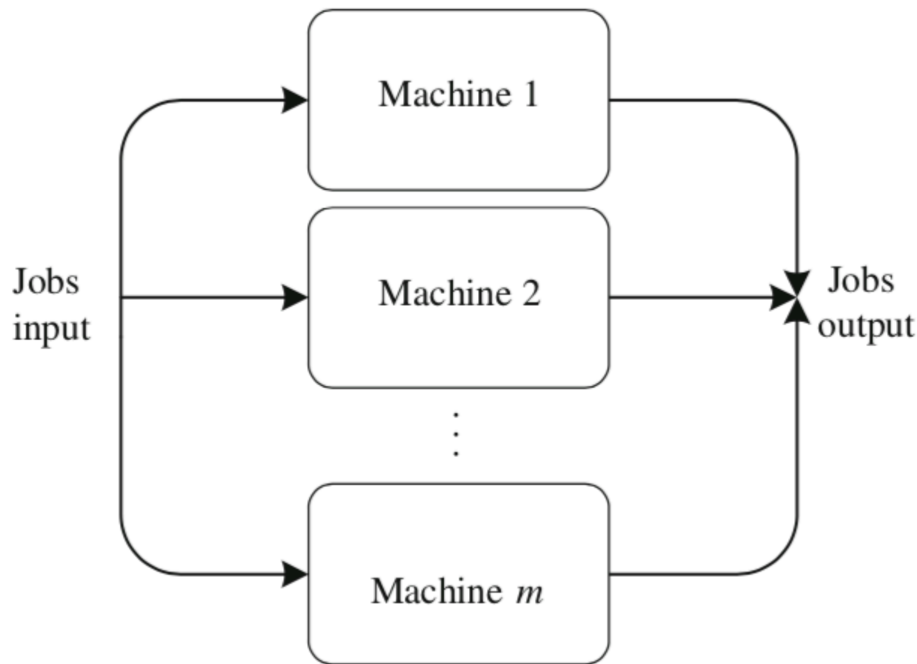


Figura 4. Flujo de trabajo máquinas paralelas (Framinan et al., 2014)

1.2 Algoritmos aproximados

Desde la perspectiva de la complejidad computacional existen dos tipos de problemas: los polinomiales (P), que son más sencillos, y los no polinomiales (NP-Hard). La complejidad de un problema de máquinas paralelas de tipo combinatorio suele ser bastante elevada. El número de soluciones aumenta considerablemente debido a la multiplicidad y diversidad de los elementos, por la gran variedad de restricciones, y por la incertidumbre. Si a eso le sumamos la competencia basada en el tiempo y la necesidad de establecer un sistema flexible que responda con rapidez a los cambios, suele ser frecuente la aplicación de métodos aproximados que resuelven el problema dejando al margen la búsqueda de la solución óptima, lo cual reduce el tiempo computacional. Con este enfoque se consigue que una solución inicial se vaya perfeccionando mediante la búsqueda de vecindades. La vecindad de una solución es el conjunto de todas las soluciones posibles del espacio de soluciones que pueden ser generadas con una leve modificación o movimiento local. Sin embargo, aunque los métodos aproximados son procedimientos que generan una solución de calidad, no puede demostrarse formalmente. Por ello, en problemas no convexos donde se presentan máximos o mínimos locales, no podemos tener la certeza de saber si la solución corresponde con un óptimo local o global.

En los métodos aproximados se distingue entre heurística y metaheurística. Las metaheurísticas son procedimientos diseñados de manera genérica que con pocas modificaciones pueden adaptarse a cualquier problema específico. Por el contrario, las heurísticas se diseñan para un problema en concreto, aunque es cierto que en ocasiones puede aplicarse con éxito fuera su alcance. Dentro de las metaheurísticas se enmarcan los métodos basados en poblaciones, como los algoritmos genéticos, colonia de hormigas, enjambre de abejas, etc. Estos algoritmos trabajan con una población formada por un conjunto de soluciones admisibles que son las que se van perfeccionando. En todos los casos, para que el algoritmo quede claramente definido hay que determinar:

- Representación de la solución: en entornos de programación suelen ser una secuencia o permutación de los trabajos.

- Inicialización: un procedimiento metaheurístico está aplicado a una población inicial. En algunas ocasiones se genera de manera aleatoria y en otras se inicializa a partir de heurísticas constructivas como puede ser una regla de despacho. Aunque es cierto que la aletoriedad puede suponer una baja calidad, en ocasiones las heurísticas también podrían limitar el proceso de búsqueda dificultando la posibilidad de encontrar la solución óptima global.

- Proceso de búsqueda de vecindario: Se define el proceso de búsqueda de vencindario junto al operador usado, el criterio de terminación y el criterio de aceptación de vecindad.

En este trabajo se profundiza en algoritmos que se basan en la inteligencia colectiva (*Swarm intelligence*) representada en la Figura 5, es decir, en el comportamiento de un colectivo que no está centralizado. Estos procedimientos presentan unos agentes que solucionan un problema complejo actuando de manera sencilla. Generalmente están inspirados en el comportamiento de la naturaleza (bio-inspirados) ya sea de un animal, un ciclo evolutivo, etc. Son algoritmos bastante genéricos y se adaptan con facilidad a cualquier tipo de problema, pero hay que tener en cuenta que debido a la gran cantidad de parámetros de los que dependen, con reacciones poco previsibles, se corre el riesgo de estancarse en un óptimo local.

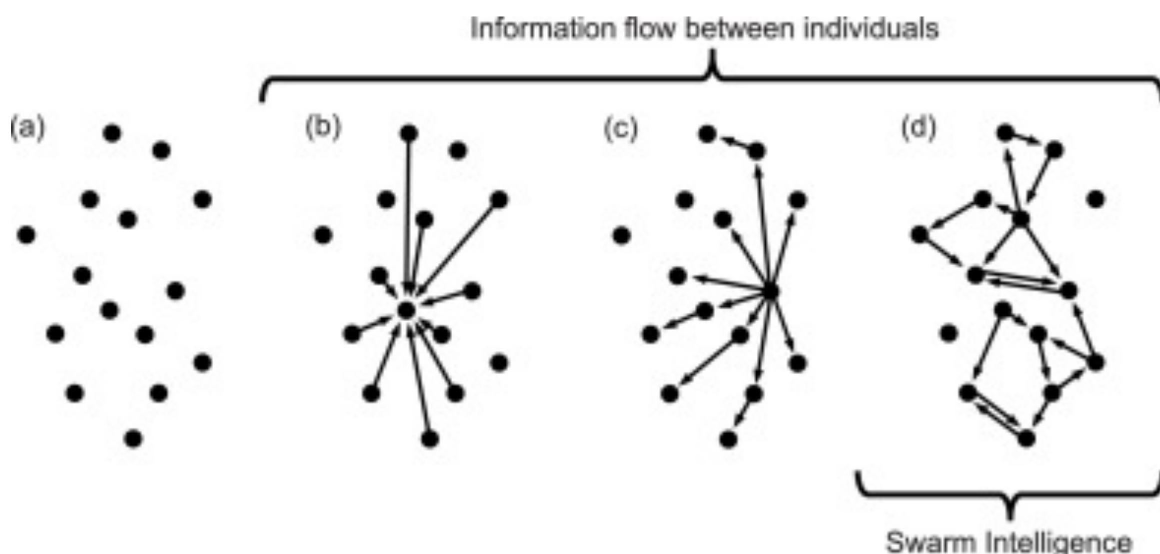


Figura 5. Intercambio de información en algoritmo de inteligencia colectiva (Ioannou, 2017)

1.3 Programación de la producción en el sector sanitario

Debido al incremento de población de avanzada edad y a la existencia de presupuestos restrictivos (Fei et al., 2010), las organizaciones de atención médica buscan realizar sus servicios de manera eficiente reduciendo al máximo los costes asociados asegurando la calidad. Para conseguir reducir estos costes es importante gestionar adecuadamente los recursos del hospital, especialmente los quirófanos, pues es la instalación que presenta una mayor partida en el presupuesto (Macario et al., 1995). En cuanto a la calidad, centrándonos en el contexto de los quirófanos, está presente si

se presta un excelente servicio consiguiendo la satisfacción del cliente. Para eso es necesario evitar esperas entre el instante en el que se determina la necesidad de una operación hasta que se realiza.

Las decisiones relacionadas con la gestión del quirófano, al igual que cualquier proceso productivo, suelen desglosarse en tres niveles de decisión: estratégico (largo plazo), táctico (medio plazo) y operacional (corto plazo). En el nivel estratégico las variables de decisión son el número y los tipos de cirugías que se planifican, y en el táctico se determinan los recursos requeridos. Posteriormente en el nivel operativo se determina el quirófano y la fecha de intervención de cada cirugía de la lista de espera, es decir, el horario quirúrgico. (Molina-Pariente, Fernandez-Viagas, et al., 2015).

El Trabajo Fin de Grado desarrollado se enmarca en el Hospital Virgen del Rocío de Sevilla (España), tercer hospital nacional con mayor actividad y presupuesto, en el que se atienden alrededor de 42.500 cirugías anuales (Molina-Pariente, Hans, et al., 2015). En concreto, el documento está focalizado en la Especialidad de Cirugía Plástica y Quemaduras mayores en donde, a diferencia de otras especialidades, la programación de los pacientes en los servicios quirúrgicos se toma con la ayuda de herramientas de apoyo a la decisión (ASSYST). Con esta herramienta administrativa se trata de encontrar un orden apropiado de intervención de los pacientes definiendo los tiempos en los que se deben iniciar cada operación en los quirófanos con el objetivo de encontrar el valor óptimo de un índice de desempeño. A pesar de que en el hospital ya existen modelos de decisión que permiten realizar análisis en función de los distintos escenarios posibles, la principal motivación del estudio consiste en diseñar y/o adaptar otros métodos aproximados al problema de estudio.

El trabajo está organizado de la siguiente forma. En la sección 2 se describe formalmente el problema en cuestión. La adaptación e implementación de la metodología empleada para resolver el problema de forma aproximada aparece descrita en la sección 3, y los resultados experimentales obtenidos se presentan en la sección 4. Finalmente, en la sección 5 se expresan las conclusiones y se exponen posibles puntos de partida para seguir trabajando en el desarrollo del trabajo en un futuro.

2 DESCRIPCIÓN DEL PROBLEMA

"El valor de una idea radica en el uso de la misma"
- Tomas A. Edison -

Contando con las cirugías electivas y de urgencia, la especialidad de Cirugía Plástica y Quemaduras Mayores del Hospital Virgen del Rocío realiza alrededor de 3.000 intervenciones por año (Molina-Pariente, Hans, et al., 2015). Dentro de la lista de espera de los quirófanos existen dos tipos de pacientes: de atención electiva o programada y aquellos con casos urgentes. Este estudio se centra en la gestión de la lista de espera del primer tipo de paciente, ya que las emergencias deben ser realizadas de manera inmediata sin necesidad de programarse debido a su gravedad.

La especialidad cuenta 4 quirófanos multifuncionales para realizar las operaciones. Sin embargo, las cirugías de urgencia tienen reservado de forma permanente uno de ellos, donde se usan recursos adicionales (recursos quirúrgicos urgentes). Por tanto, en el problema objeto de estudio solo se cuenta con 3. Los quirófanos no están siempre disponibles. Los lunes y martes de mañana pueden usarse cualquiera de ellos, pero tanto los lunes y martes por la tarde, como los miércoles, jueves y viernes de mañana, solo pueden usarse dos. Así mismo, los miércoles, jueves y viernes de tarde solo hay un único quirófano donde se puede operar. El horario para un turno de mañana es de 8:30 a 15:00, y para uno de tarde de 15:00 a 20:00.

En la etapa de consulta previa a la cirugía, el encargado de la toma de decisiones asigna a cada paciente en la lista de espera un cirujano especializado que se encargará de realizar la operación ya que presenta las habilidades necesarias. Así mismo, basándose en datos históricos y en las particularidades del paciente en cuestión, el tomador de decisiones establece la duración estimada y la prioridad de la intervención. En relación con esta prioridad, se determina una fecha de vencimiento relativa que se intenta no superar, es decir, el instante en el que como máximo se debería iniciar la operación. A partir de este momento queda determinada la fecha de lanzamiento del paciente tras haber superado todas las pruebas médicas pertinentes, pues se recoge el día en el que se determina la necesidad de realizar la operación. Si bien el problema en realidad se trata de un problema estocástico debido a la posible variabilidad en los datos de entrada (duraciones, fechas de vencimiento, etc.), en la práctica se resolverá simplificándolo a un caso determinista en donde cada paciente posee sus propios parámetros para poder obtener soluciones en tiempos asumibles.

En el problema se pretende resolver la programación de los quirófanos de la especialidad durante una semana que presenta turnos de mañana y tarde, con el objetivo de conseguir que las cirugías de los pacientes en cuya consulta previa se haya determinado la necesidad de alguna operación se realicen en uno de los quirófanos de los que dispone el hospital.

2.1 Restricciones

Como consecuencia de la planificación táctica previa al problema que se aborda, los cirujanos no pueden operar en todos los turnos del horizonte de planificación, por lo que cuando se determina

el turno en el que se opera un paciente, debe cumplirse que sea uno de los que tenga asignado el cirujano asociado.

Además de las limitaciones tácticas, existen limitaciones intrínsecas del entorno como consecuencia de los tiempos máximos de operatividad de un cirujano y de un quirófano. En el caso del cirujano, el límite máximo está fijado en 6 horas y media, y en el caso del quirófano en 6 horas y media para los turnos de mañana, y 4 para los turnos de tarde. Así mismo, debido a que el cirujano solo puede estar operando a un único paciente a la vez, se debe evitar en todo momento la superposición de operaciones en distintos quirófanos del mismo turno que compartan el mismo cirujano.

Finalmente, existen unos límites mínimos y máximos a la hora de establecer una operación. Nunca se puede iniciar antes de la fecha en la que se lance. De igual forma, una operación podrá ser planificada como máximo el ultimo día del horizonte de planificación. Si no es posible establecerla en este intervalo, porque no se cumplan todas las restricciones anteriores, la operación no se realiza dentro del periodo que se está programando.

Los recursos humanos e instrumentales necesarios para comenzar la operación no suponen ninguna restricción, pues están disponibles en todo momento.

2.2 Objetivo

Para poder comenzar con la planificación o programación de los quirófanos es necesario establecer, en primer lugar, un orden de la lista de los pacientes en espera. Esta ordenación o secuencia se obtendrá con la metodología desarrollada en el apartado 3.2.

Una vez se conozca la ordenación, lo ideal sería ir estableciendo para cada uno de los pacientes en ese orden, un turno de operación y un quirófano que cumpla con las restricciones y esté en el periodo de tiempo establecido por la fecha de lanzamiento y la fecha de vencimiento. Sin embargo, cuando existe una elevada congestión no siempre es posible. Por ello, el objetivo del presente documento consiste en determinar una ordenación con la que se realice el mayor número de operaciones de forma que se minimice la suma la tardanza ponderada según los valores de prioridad de cada intervención. Es decir, además de minimizar tardanza, se pretende reducir al máximo la cantidad de pacientes que no se operan dentro del horizonte de planificación. La tardanza de un paciente se define como el período de tiempo entre la fecha de vencimiento de la operación y la fecha de en la que finalmente se ejecuta la cirugía.

3 METODOLOGÍA

Lo que no es útil para la colmena, no es útil para la abeja.
- Marco Aurelio -

Como se ha mencionado en el apartado 2.2, el primer paso para establecer la planificación de los quirófanos consiste en determinar una ordenación de la lista de los pacientes en espera. En este entorno productivo en concreto, donde cada intervención se realiza en una etapa constituida por múltiples recursos o quirófanos en paralelo (parallel machine), el problema estaría correctamente resuelto si además de determinar la secuencia u ordenación de los pacientes, se define una regla de asignación a los quirófanos. De esa forma, se conocería la secuencia de pacientes en cada uno de los quirófanos como en el ejemplo de la Figura 6, y quedaría establecido el horario quirúrgico de cada intervención.

	1	2	3	4	5	6	7
Machine 1	7	13	20	14	1	3	6
Machine 2	12	11	5	19	10	8	
Machine 3	2	9	18	17	15	16	4

Figura 6. Ejemplo solución 20 trabajos y 3 máquinas paralelas (Framinan et al., 2014)

La regla de asignación que se usa consiste en asignar la operación al quirófano en el cual el cirujano asociado haya trabajado durante más tiempo. Esta regla se utiliza siempre que exista más de un quirófano que cumpla con las restricciones donde se pueda realizar la operación, lo cual permite condensar las operaciones en un mismo quirófano para evitar tiempos de inactividad.

3.1 Determinación del horario quirúrgico

Para establecer el horario quirúrgico de los pacientes por cada posible ordenación de la lista de espera que se plantee, como se conoce la regla de asignación, basta con seguir un procedimiento sencillo.

En primer lugar, se determina el turno de operación más próximo posible para cada uno de los pacientes en el orden establecido. Un paciente podrá ser intervenido como mínimo el día del lanzamiento de su operación. Por otro lado, el turno debe permitir que la intervención pueda

realizarse sin interrupciones. Para ello debe cumplirse que el cirujano asociado disponga de horas de trabajo suficientes. Es decir, que se trate de uno de los turnos en los que tácticamente se ha decidido que puede operar, y que además no se supere el límite máximo de horas que puede estar trabajando. Hasta que no se cumplan las dos condiciones anteriores, se atrasa la operación al turno siguiente siempre y cuando esté dentro del horizonte de planificación. Si queda fuera, el paciente no será operado, lo cual penaliza con un 0.001 el valor de la Función Objetivo. La Función Objetivo es la variable que mide la calidad de la solución del problema siguiendo el criterio establecido en el apartado 2.2. Por tanto, mientras más pequeño sea el número de pacientes que no se operan en el intervalo que se está programando, menor será valor de la Función Objetivo y, por consiguiente, mejor solución.

Si por el contrario se encuentra un turno lo más próximo posible en el que se cumplan las restricciones anteriores, se asocia el paciente a uno de los quirófanos que estén disponibles en ese momento siguiendo la regla de asignación. Para ello ha de cumplirse que sea uno de los quirófanos habilitados ese turno, y que además disponga de tiempo suficiente como para realizar la operación sin interrupción, teniendo en cuenta la duración de esta. Finalmente, debe verificarse que no existe solape con otra operación ya planificada en otro quirófano del mismo turno que comparta cirujano con la operación que se planifica. Si no existe ningún quirófano que cumpla con estas nuevas restricciones, se retrasa el turno de operación hasta encontrar un turno que cumpla con todas las restricciones y que disponga de un quirófano en el que se pueda asignar la operación, siempre que esté dentro del horizonte de planificación. Si queda fuera, el paciente no será operado en el intervalo que se programa, lo cual penaliza incrementando un 0.001 el valor de la Función Objetivo.

El horario quirúrgico queda establecido en el momento que se establece el quirófano en el que se realiza la intervención. Si la operación se establece con retraso respecto a su fecha límite, se suma a la Función Objetivo la tardanza ponderada por la prioridad, pues el objetivo principal es encontrar una lista de pacientes con la que no se produzcan demoras. Mientras menor sea la tardanza de los pacientes que se operan, menor será el valor de la Función Objetivo y, por tanto, mejor solución.

3.2 Ordenación lista de espera

Con el fin de encontrar una secuencia u ordenación de pacientes que proporcione una buena solución al problema, es decir, un buen resultado de la Función Objetivo, se aplican cinco variaciones de la metaheurística de las colonias de abejas artificiales (“Artificial Bee Colony”) con las que se obtienen cinco secuencias distintas que serán analizadas posteriormente para determinar la más efectiva.

La metaheurística propuesta por Karaboga & Basturk (2007) e inspirada en la inteligencia colectiva (*Swarm intelligence*) y autoorganización de las abejas en la búsqueda de alimento, es la aconsejada siguiendo el contexto concreto en el que se desarrolla la investigación. Al tratarse de un algoritmo poblacional, trabaja con una población constituida por varios individuos o posibles ordenaciones de pacientes, lo cual permite recorrer con mucha expansión el sistema de decisión. Además, se demostró que el rendimiento del algoritmo ABC era tan competitivo como otros algoritmos basados en población empleando menos parámetros de control (Karaboga & Basturk, 2007) (Karaboga & Basturk, 2008).

Cada fuente de alimento o punto de comida posible representa un individuo de la población. Mediante ensayos y comparaciones entre las distintas fuentes de alimento encontradas se generan nuevas fuentes mejoradas y se descartan las menos efectivas, lo que permite crear generaciones

que superan a sus predecesoras siguiendo el método evolutivo. Para descartar o aceptar una solución el método compara la puntuación o *score* de cada individuo, que representa el valor de la Función Objetivo de la secuencia. Debido a que se trata de un enfoque heurístico y evolutivo, es probable que no encontremos una secuencia con la que se obtenga la Función Objetivo óptima, pero como consecuencia del autoperfeccionamiento basado en los principios de la Inteligencia Artificial, el resultado estará bastante próximo al óptimo.

La metodología está basada en la comunicación de unos agentes, o de ahora en adelante abejas, que mediante bailes consiguen interactuar para determinar dónde se encuentran las fuentes de alimento. En esta interacción se definen distintos roles o clases de abejas, cada una con una función:

- Abejas obreras (*employee*): exploran el vecindario de la fuente de alimento asignada de manera que, si se encuentra una fuente de alimento mejor, cambian la posición hacia el nuevo punto y comparten la información con las abejas observadoras mediante el baile. Es decir, por cada fuente de alimento hay una abeja obrera y su función es evaluar y modificar las fuentes actuales para mejorarlas.

- Abejas observadoras (*onlooker*): examinan el vecindario de las fuentes de comida ya encontradas por las abejas obreras a partir de la información que estas le transmiten mediante el baile, y en el caso de encontrar una mejor, modifican la posición de las obreras.

- Abeja exploradora (*scout*): tratan de buscar nuevas posibles fuentes de alimento considerando el espacio de soluciones completo del problema, evitando así el estancamiento en un óptimo local.

Los distintos roles dan lugar a las tres fases de la metodología que varían según la versión en la que nos encontremos. Como se observa en el pseudocódigo de la Figura 8, estas fases son aplicadas a una población inicial a partir de un proceso de evolución iterativo hasta que se complete un criterio de parada, generalmente establecido en función del tiempo. El tamaño de la población inicial (PS) viene determinado por los individuos que la componen, los cuales son generados también de distinta forma en función de la versión.

Para conseguir encontrar soluciones de calidad de forma eficiente sin quedar atrapado en regiones específicas del espacio de soluciones, en cada ciclo se fomenta tanto la explotación como la exploración. Mientras que la explotación pretende mejorar las soluciones ya encontradas, la exploración trata de diversificar la búsqueda. En cualquier versión de esta metaheurística en concreto, la abeja obrera y observadora son las encargadas de explotar la solución y la abeja exploradora es la que diversifica. La tercera fase se aplica tras haber realizado la explotación de forma repetitiva durante un número de ensayos. La diferencia entre exploración y explotación se muestra en la Figura 7.

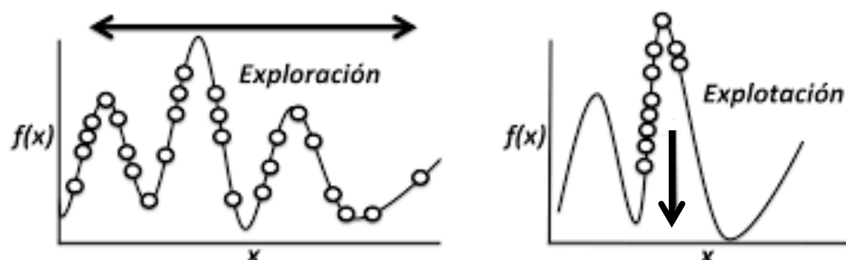
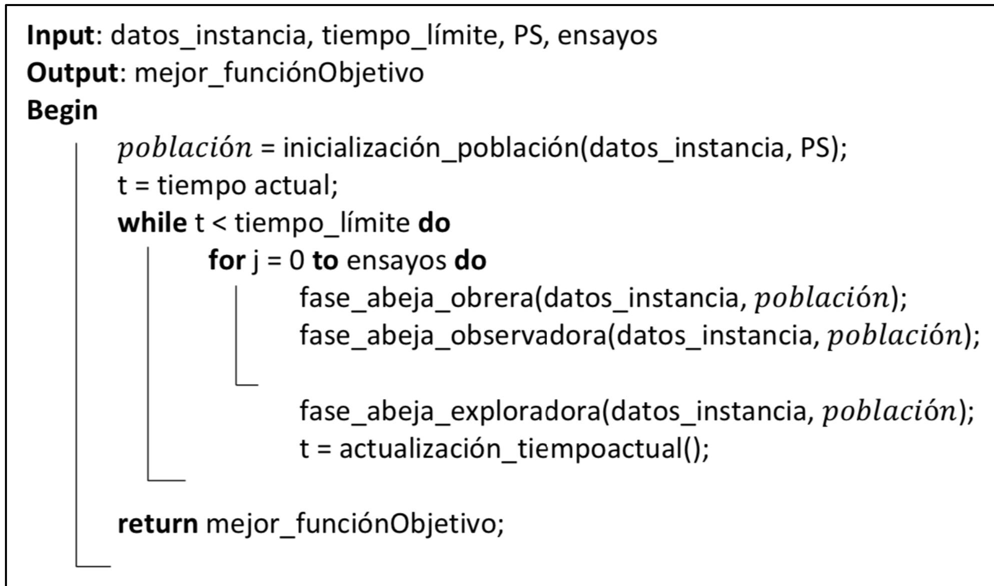


Figura 7. Diferencia exploración y explotación (*Metaheurísticas | Peralta-Abarca | Inventio, la génesis de la cultura universitaria en Morelos*, n.d.)



```
Input: datos_instancia, tiempo_límite, PS, ensayos
Output: mejor_funciónObjetivo
Begin
    población = inicialización_población(datos_instancia, PS);
    t = tiempo actual;
    while t < tiempo_límite do
        for j = 0 to ensayos do
            fase_abeja_obrera(datos_instancia, población);
            fase_abeja_observadora(datos_instancia, población);

            fase_abeja_exploradora(datos_instancia, población);
            t = actualización_tiempoactual();

    return mejor_funciónObjetivo;
```

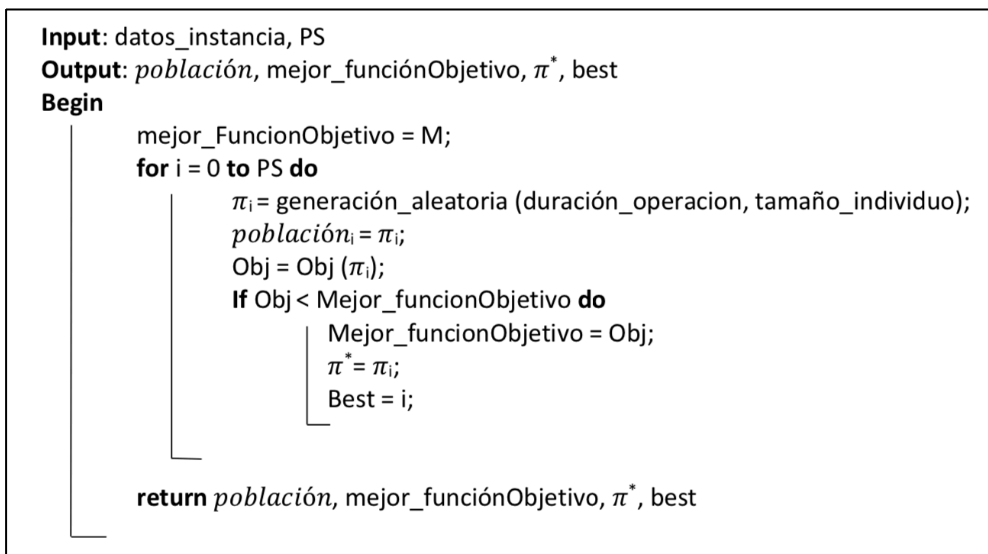
Figura 8. Pseudocódigo algoritmo Artifical Bee Colony (Elaboración propia)

3.3 ABC Básico

El primer algoritmo que se aplica para determinar una secuencia u ordenación de los pacientes en lista de espera, es una adaptación de la metaheurística ABC original propuesta por Karaboga & Basturk (2007). Se adapta el algoritmo para poder aplicarlo en un problema de secuenciación como es el caso de la programación de los quirófanos.

3.3.1 Determinación de la población inicial

Antes de que los agentes realicen sus funciones, se determina la población o conjunto de soluciones inicial. En este caso, siguiendo el proceso que se muestra en el pseudocódigo de la Figura 9[1], se inicializa la población generando vectores aleatorios $\{\pi_1, \pi_{2...}, \pi_{PS}\}$ hasta completar su tamaño. Cada vector recoge una secuencia u ordenación de la lista de espera de los pacientes.



```
Input: datos_instancia, PS
Output: población, mejor_funciónObjetivo, π*, best
Begin
    mejor_FuncionObjetivo = M;
    for i = 0 to PS do
        πi = generación_aleatoria (duración_operacion, tamaño_individuo);
        poblacióni = πi;
        Obj = Obj (πi);
        If Obj < Mejor_funcionObjetivo do
            Mejor_funcionObjetivo = Obj;
            π* = πi;
            Best = i;

    return población, mejor_funciónObjetivo, π*, best
```

Figura 9. Pseudocódigo inicialización población algoritmo ABC básico (Elaboración propia)

[1] Las variables π^* y best recogen la mejor secuencia hasta el momento y su posición en la población, que serán usadas en la fase de la abeja observadora.

3.3.2 Fases del algoritmo

Una vez que se obtiene la población inicial, se continúa inicializando cada uno de los agentes o abejas, es decir, se procede con la aplicación de las tres fases de la metodología de manera iterativa para explorar nuevas soluciones, hasta que se complete un criterio de terminación basado en un tiempo máximo de ejecución.

- Fase de la abeja obrera: En la fase de la abeja obrera del algoritmo básico, para cada una de las soluciones de la población inicial, se busca un vecino (π_{new}) combinando el individuo actual (π_u) con un individuo seleccionado al azar ($\pi_{\mathbf{q}}$) de la siguiente forma:

  $\boldsymbol{\pi_{new} = \pi_u + (\pi_u - \pi_q) \cdot r}$

  donde **r** es un número real uniformemente distribuido entre 1 y -1.

  Sin embargo, este método no se puede aplicar en un problema basado en permutación. Para este tipo de problemas, los vecindarios comúnmente utilizados se definen mediante inserción, intercambio adyacente e intercambio de pares (Schiavinotto & Stützle, 2007), cuyos ejemplos se muestran en la Figura 10, 11 y 12 respectivamente. La inserción extrae un trabajo de la posición aleatoria u, para incorporarlo en la v que también se elige aleatoriamente. En el intercambio adyacente, un trabajo en la posición aleatoria u se intercambia con el de la posición u+1. Finalmente, en el intercambio de pares, los trabajos que se intercambian son los de la posición u y la posición v, ambas elegidas de manera aleatoria. No obstante, en esta fase se utiliza un operador de intercambio adyacente múltiple que se demostró que obtenía mejores resultados (Pan et al., 2013). En él, se realizan dos intercambios adyacentes. Uno entre la posición u y u+1 y otro entre la posición v y v+1. Tanto u como v son elegidas de manera aleatoria.

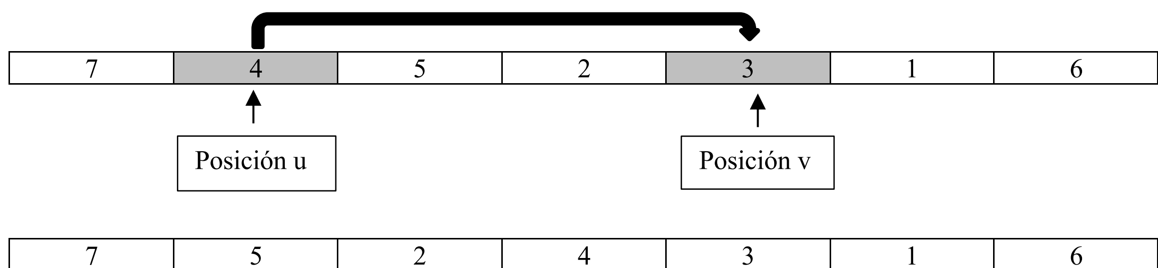


Figura 10. Inserción aleatoria (Elaboración propia)

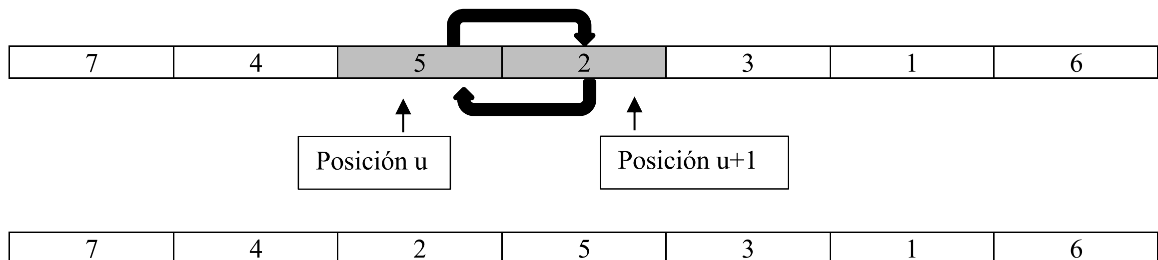


Figura 11. Intercambio adyacente (Elaboración propia)

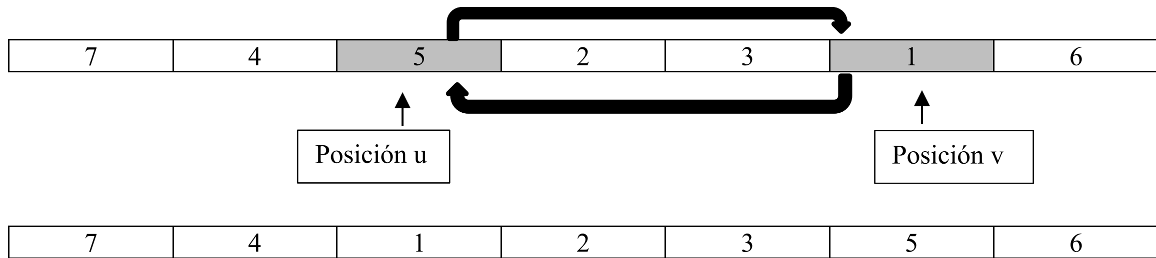


Figura 12. Intercambio de pares (Elaboración propia)

En la fase de la abeja obrera del algoritmo ABC básico, se acepta la incorporación en la población del nuevo vecino solo si presenta mejor valor de la Función Objetivo que el individuo a partir del cual se ha generado. En otras palabras, se aplica un mecanismo de selección codicioso para elegir la nueva solución o mantener la solución anterior. El resumen del proceso se muestra en la Figura 13.

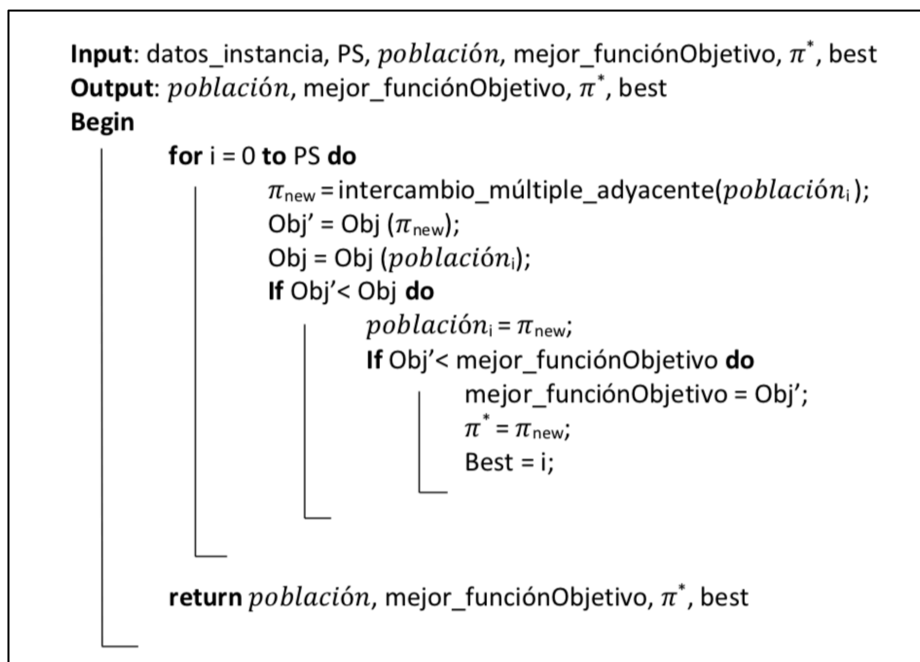


Figura 13. Pseudocódigo fase de la abeja obrera algoritmo ABC básico
(Elaboración propia)

- <u>Fase de la abeja observadora:</u> Tras realizar la fase de la abeja obrera, la abeja observadora evalúa la información que le han proporcionado y selecciona la fuente de alimento que presente mayor cantidad de néctar, es decir, la solución que presente mayor aptitud. Generalmente la probabilidad de que una abeja observadora seleccione una fuente de alimento viene determinada por la siguiente ecuación:

$$\xi_u = \frac{f_u}{\sum_{u=1}^{PS} f_u}$$

Donde f_u representa el valor de aptitud de π_u. Cuanto más alto es, mayor probabilidad de que la abeja observadora seleccione esa solución.

En el problema basado en permutación, la solución más apta será aquella que presente menor valor de la Función Objetivo. Por ello, la fase de la abeja observadora se aplica al mejor individuo (π*) que se obtiene tras realizar la fase de la abeja obrera. Como se muestra en la Figura 14, al mejor individuo se le aplica el mismo procedimiento que en la fase anterior, es decir, a partir de él se genera un nuevo vecino mediante intercambio adyacente múltiple, que se incorpora a la población siempre que presente menor valor de la Función Objetivo. [2]

```
Input: datos_instancia, población, mejor_funciónObjetivo, π*, best
Output: población, mejor_funciónObjetivo, π*, best
Begin
    π_new = intercambio_múltiple_adyacente(π*);
    Obj' = Obj (π_new);
    Obj = Obj (π*);
    If Obj'< Obj do
        población_best = π_new;
        mejor_funciónObjetivo = Obj';
        π* = π_new;

    return población, mejor_funciónObjetivo, π*
```

Figura 14. Pseudocódigo fase abeja observadora algoritmo ABC básico
(Elaboración propia)

- <u>Fase de la abeja exploradora:</u> Una vez que se hayan realizado un número de ensayos de explotación local, se inicia la fase de exploración global. La fase exploradora recorre todos los individuos de la población comprobando que hayan sido mejorados durante alguno de los ensayos, es decir, se comprueba que el individuo que ocupa cada posición en la población es distinto al individuo que ocupaba esa posición en la población inicial que se generó en el apartado 3.1.1. Cada individuo que no se haya mejorado se elimina. En la Figura 15 se puede ver que en el algoritmo básico, la abeja exploradora genera en su lugar un nuevo individuo al azar que es incorporado directamente a la población.

```
Input: datos_instancia, PS, población, mejor_funciónObjetivo, π*, best
Output: población, mejor_funciónObjetivo, π*, best
Begin
    for i = 0 to PS do
        If población_i no ha sido modificado en algún ensayo do
            π_new = generación_aleatoria (duración_operacion, tamaño_individuo);
            población_i = π_new;
            Obj = Obj (π_new);
            If Obj < mejor_funcionObjetivo do
                Mejor_funcionObjetivo = Obj;
                π* = π_new;
                Best = i;

    return población, mejor_funciónObjetivo, π*, best
```

Figura 15. Pseudocódigo fase abeja exploradora algoritmo ABC básico
(Elaboración propia)

[2] Se sigue actualizando π^* para poder usarse en futuras fases de la abeja observadora de los siguientes ciclos. Best también se actualiza si sufre alguna modificación, solo que en esta fase no ocurre.

3.4 Mejora 1

El segundo algoritmo empleado para la determinación de la secuencia o lista ordenada de pacientes es una adaptación del algoritmo ABC propuesto por Pan et al. (2014) aplicado al problema de programación híbrida de flowshop con minimización del makespan, es decir, del tiempo final en el que todas las tareas son completadas. El flowshop híbrido es un entorno de producción que se caracteriza por que las tareas pasan por cada etapa en el mismo orden. Todos los trabajos tienen la misma ruta, pero cada etapa podría estar constituida por máquinas paralelas idénticas, proporcionales o distintas.

3.4.1 Determinación de la población inicial

Como no se conoce la solución óptima, no existe un criterio que asegure con certeza la proximidad a esta, pero para mejorar el espacio de soluciones inicial y asegurar la calidad y diversidad, en esta versión de la metodología la población se inicia de manera semi-aleatoria como se puede ver en el pseudocódigo de la Figura 17. Es decir, los individuos que la componen (PS) serán generados aleatoriamente a excepción de los cuatro primeros, que se generan de la siguiente forma:

- Primer individuo: En ocasiones aplicar heurísticas puede proporcionar buenas soluciones iniciales cuando se procede posteriormente con una metaheuristica (Framinan et al., 2014). En este caso, para generar la primera solución de la población se aplica la heurística NEH (Nawaz et al., 1983) cuyo pseudocódigo se muestra en la Figura 16.

  La heurística consiste un procedimiento constructivo de secuenciación constituido por tres pasos. En el primer paso se ordenan los pacientes con el criterio LPT (*Longest Processing Time first*), de forma que las duraciones de las operaciones de los pacientes estén ordenadas de mayor a menor. En el segundo paso se elige la permutación que genere mejor valor de la Función Objetivo entre el primer y segundo paciente de la ordenación inicial, lo cual da lugar a la generación de la primera secuencia parcial. Finalmente, en la tercera fase (fase de inserción) se construye la secuencia de forma progresiva, incorporando de uno en uno cada trabajo en el orden previamente establecido. Los trabajos se sitúan en la mejor posición de la secuencia parcial que se va generando. En caso de empate, se elige la última secuencia que se haya generado.

Procedure NEH
Generate a seed sequence $\beta = (\beta_1, \beta_2, ..., \beta_{PS})$ by using the LPT rule
$\pi := (\beta_1)$
for $h := 2$ **to** PS **do** %(the NEH enumeration procedure)
Take job β_h from β and test it in all of the h possible slots of π
Insert job β_h in π at the slot that results in the lowest objective value
endfor
return π

Figura 16. Pseudocódigo NEH (Pan et al., 2014)

```
Input: datos_instancia, PS
Output: población, mejor_funciónObjetivo
Begin
    mejor_funciónObjetivo = M;
    π0 = generación_NEH(duración_operacion, tamaño_individuo);
    población0= π0;
    Obj = Obj(π0 );
    if Obj < mejor_funciónObjetivo do
        mejor_funciónObjetivo = Obj;

    π1 = generación_NEH(duración_operacion, tamaño_individuo);
    población1= π1;
    Obj = Obj(π1);
    if Obj < mejor_funciónObjetivo do
        mejor_funciónObjetivo = Obj;

    π2 = generación_NEH(duración_operacion, tamaño_individuo);
    población2= π2;
    Obj= Obj(π2);
    if Obj < mejor_funciónObjetivo do
        mejor_funciónObjetivo = Obj;

    π3 = generación_NEH(duración_operacion, tamaño_individuo);
    población3= π3;
    Obj = Obj(π3);
    if Obj < mejor_funciónObjetivo do
        mejor_funciónObjetivo = Obj;

    contador = 4;
    while contador < PS do
        πcontador = generación_aleatoria(tamaño_individuo);
        comprobación_repetición(πcontador, población);
        if πcontador no se repite con ningún individuo de la población do
            poblacióncontador = πcontador;
            contador = contador +1;
            Obj= Obj(poblacióncontador);
            if Obj < mejor_funciónObjetivo do
                mejor_funciónObjetivo = Obj;

    return población, mejor_funciónObjetivo
```

Figura 17. Pseudocódigo inicialización población mejora 1 (Elaboración propia)

- Segundo individuo: Para intentar reducir la suma de la tardanza de los pacientes, el segundo individuo es generado a partir de la regla de despacho EDD (*Earlier Due Date first*), es decir, se obtiene una secuencia en la que los pacientes que tengan menor fecha de vencimiento se procesan primero.

- Tercer individuo: La regla de despacho usada para generar la tercera secuencia es la LPT, por tanto, el trabajo con mayor tiempo de proceso se procesa primero.

- Cuarto individuo: Debido a que la prioridad es un factor que incrementa la Función Objetivo, la cuarta secuencia se ha generado ordenando los pacientes de mayor a menor nivel de prioridad para intentar así aproximarse al óptimo del problema.

3.4.2 Fases del algoritmo

Del mismo modo que en algoritmo ABC básico, una vez que se obtiene la población inicial, se desarrollan las tres fases del algoritmo de manera iterativa hasta completar un tiempo máximo de ejecución.

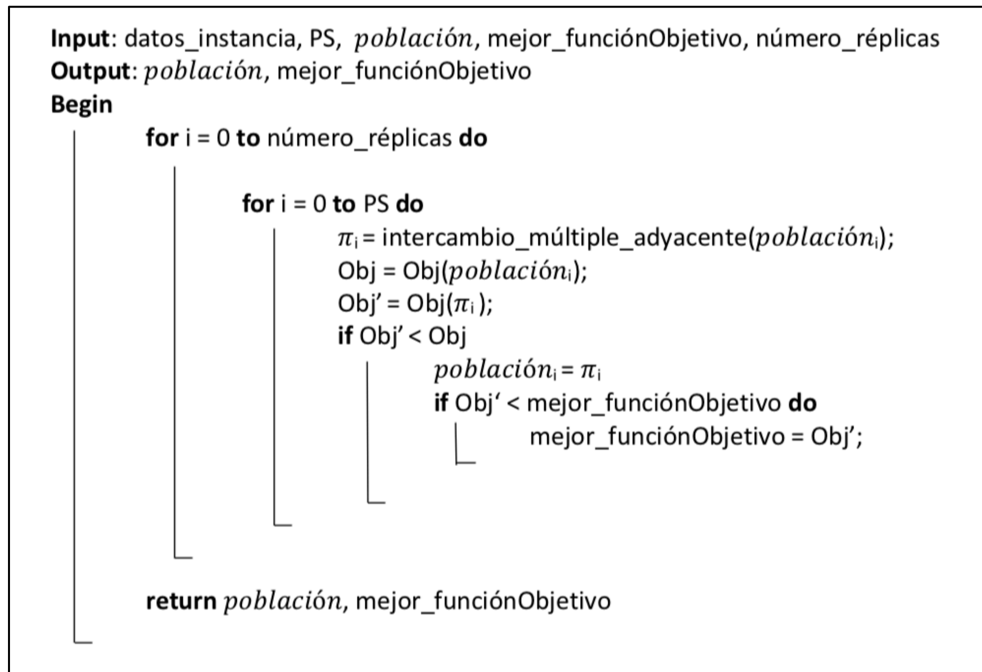
- Fase de la abeja obrera: Para la búsqueda en el vecindario de cada una de las soluciones de la población inicial, se usa el mismo método que el que se adaptó en la versión básica para el problema basado en permutación. Los vecinos se definen mediante intercambio adyacente múltiple, uno entre la posición u y u+1, y otro entre la posición v y v+1. Tanto u como v son elegidas de manera aleatoria.

  Así mismo, en este caso también se acepta incorporar el nuevo vecino en la población solo si presenta mejor valor de la Función Objetivo que el individuo a partir del cual se ha generado, siguiendo un mecanismo codicioso.

  El algoritmo ABC favorece la exploración global como resultado de explorar las vecindades de cada una de las secuencias que conforman la población. Es decir, numerosas áreas pequeñas del espacio de soluciones del problema se exploran simultáneamente. Para equilibrar la exploración y la explotación presentes, a diferencia de la versión básica, la fase de la abeja obrera de esta versión se repite durante un número de réplicas antes de comenzar con la segunda fase, como se observa en el pseudocódigo de la Figura 18.

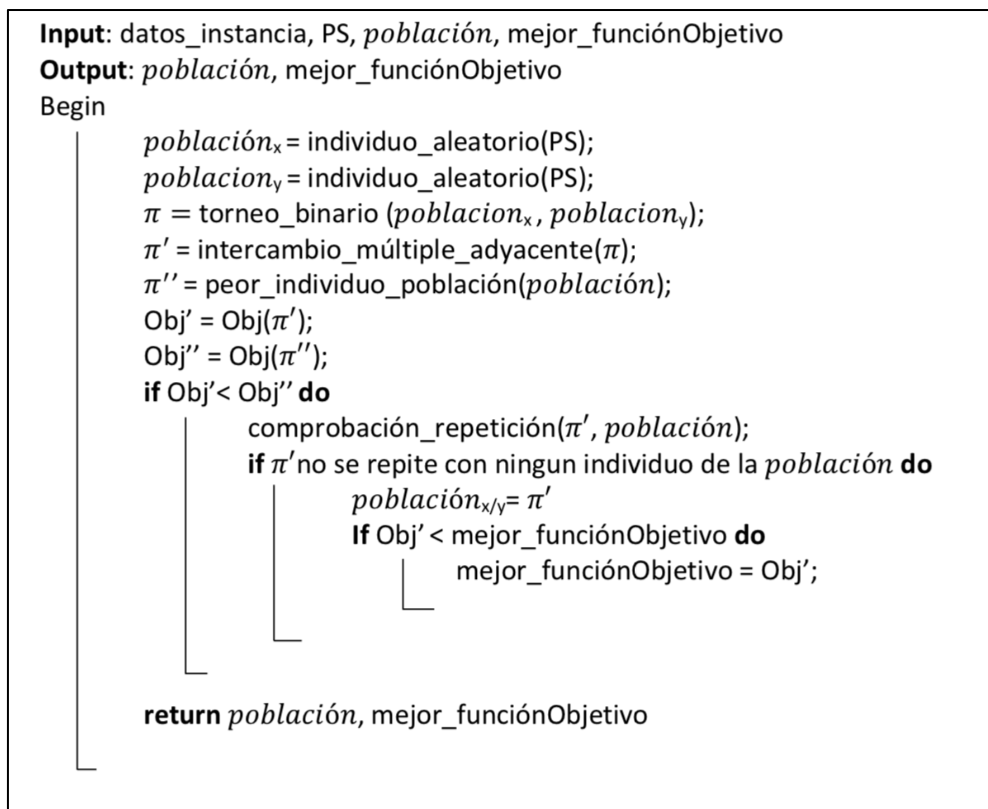
- Fase de la abeja observadora: Tras realizar la fase de la abeja obrera, la segunda fase no se aplica a la mejor solución que se obtiene de la fase anterior como ocurría en el ABC básico, sino que se seleccionan dos individuos aleatorios de la población resultante, y se elige el que mejor valor de la Función Objetivo presente, es decir, se realiza un torneo binario. Este tipo de torneo es ampliamente usado para la selección de individuos debido a su simplicidad y capacidad para escapar de los óptimos locales. Sin embargo, es cierto que la búsqueda en el espacio de soluciones es limitada porque solo se explora la vecindad de la mejor solución de entre dos elegidas al azar.

  Después de la selección, la abeja observadora realiza el mismo procedimiento que en la fase anterior. A partir de este individuo se genera un vecino mediante intercambio adyacente múltiple. En este caso, el vecino será aceptado solo si es mejor que el peor individuo de la población, si además no se repite con ninguno de los individuos presentes en la población a la hora de la incorporación. El resumen de la fase se muestra en el pseudocódigo de la Figura 19.



```
Input: datos_instancia, PS, población, mejor_funciónObjetivo, número_réplicas
Output: población, mejor_funciónObjetivo
Begin
    for i = 0 to número_réplicas do
        for i = 0 to PS do
            πi = intercambio_múltiple_adyacente(poblacióni);
            Obj = Obj(poblacióni);
            Obj' = Obj(πi );
            if Obj' < Obj
                poblacióni = πi
                if Obj' < mejor_funciónObjetivo do
                    mejor_funciónObjetivo = Obj';
    return población, mejor_funciónObjetivo
```

Figura 18. Pseudocódigo fase abeja obrera mejora 1 (Elaboración propia)



```
Input: datos_instancia, PS, población, mejor_funciónObjetivo
Output: población, mejor_funciónObjetivo
Begin
    poblaciónx = individuo_aleatorio(PS);
    poblaciony = individuo_aleatorio(PS);
    π = torneo_binario (poblacionx , poblaciony);
    π' = intercambio_múltiple_adyacente(π);
    π'' = peor_individuo_población(población);
    Obj' = Obj(π');
    Obj'' = Obj(π'');
    if Obj'< Obj'' do
        comprobación_repetición(π', población);
        if π'no se repite con ningun individuo de la población do
            poblaciónx/y= π'
            If Obj' < mejor_funciónObjetivo do
                mejor_funciónObjetivo = Obj';
    return población, mejor_funciónObjetivo
```

Figura 19. Pseudocódigo fase abeja observadora mejora 1 (Elaboración propia)

- Fase de la abeja exploradora: En el algoritmo ABC básico, llegados a este punto, si una solución perteneciente a la población no ha experimentado mejoras durante la aplicación de los ensayos, se abandonaba para introducir en su lugar una nueva solución generada

aleatoriamente. Sin embargo, ese mecanismo reducirá la eficiencia, pues en la mayoría de los casos, la solución generada aleatoriamente será probablemente peor o mucho peor que la eliminada que ha sobrevivido a varias generaciones y es bastante prometedora.

Para solucionar este problema, el nuevo individuo a incorporar en la población será la secuencia que presente mejor valor de la Función Objetivo de entre un conjunto de vecinos generados tras realizar una inserción a la secuencia eliminada tal y como se muestra en el pseudocódigo de la Figura 20. En la inserción se extrae un trabajo de la posición u, elegida de forma aleatoria, para introducirlo en la v que también se elige aleatoriamente. Se realizarán varios vecinos para conseguir que la probabilidad de trasladarse a una región prometedora del espacio de soluciones que hasta ahora no se había explorado sea elevada.

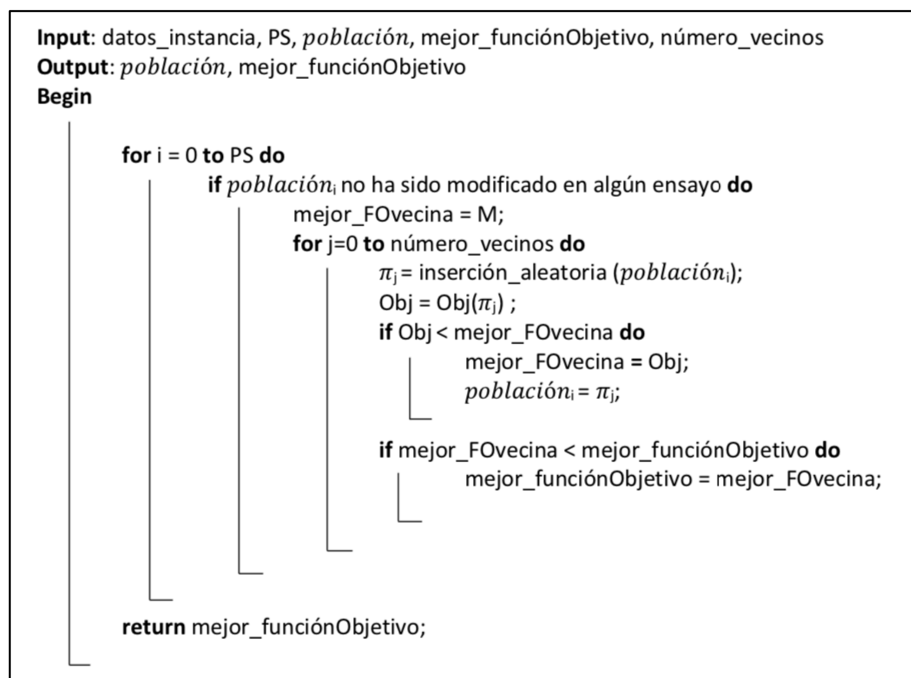


Figura 20. Pseudocódigo fase abeja exploradora mejora 1 (Elaboración propia)

3.5 Mejora 2

El tercer algoritmo presentado corresponde con la mejora 2, que será una modificación de la adaptación del algoritmo propuesto por Pan et al. (2014) desarrollado en el apartado 3.2. La modificación se produce al añadir un nuevo parámetro (δ) que controla el porcentaje de empeoramiento de los individuos que se incorporan en la población en la fase de la abeja exploradora.

3.5.1 Determinación de la población inicial

La determinación de la población inicial no presenta diferencias respecto a la mejora 1.

3.5.2 Fases del algoritmo

La única diferencia frente a la mejora 1 se realiza en la fase de la abeja exploradora mostrada en la Figura 21, que será donde se añada el parámetro. Al igual que en el caso anterior, para solucionar el problema que supone incorporar en la población individuos que sean en ocasiones peores o mucho peores que los que ya se encontraban dentro, como ocurre en el algoritmo ABC básico, los

vecinos que se analizan en la fase de exploración son los mejores entre un conjunto generado a partir de inserción. Este hecho consigue que los individuos que se incoporan posiblemente no sean mucho peores que los que ya estaban, pero tampoco asegura que sean buenos. Para solucionar este problema, la solución que no había experimentado mejora tras realizar un conjunto de ensayos no es directamente sustituida por el mejor de los vecinos que se genera. Antes de incorporarse a la población, se comprueba que el mejor de los vecinos generados sea mejor o solo un porcentaje peor (δ) que el individuo al que sustituiría. Además, también se debe comprobar que este vecino no se repite con ningún otro individuo que pertenezca a la población en el momento de la incorporación. Si el vecino cumple esas dos nuevas restricciones, se incorpora a la población sustituyendo al individuo que no había mejorado. En caso contrario, este individuo que no ha mejorado no se abandona, es decir, seguiría perteneciendo a la población.

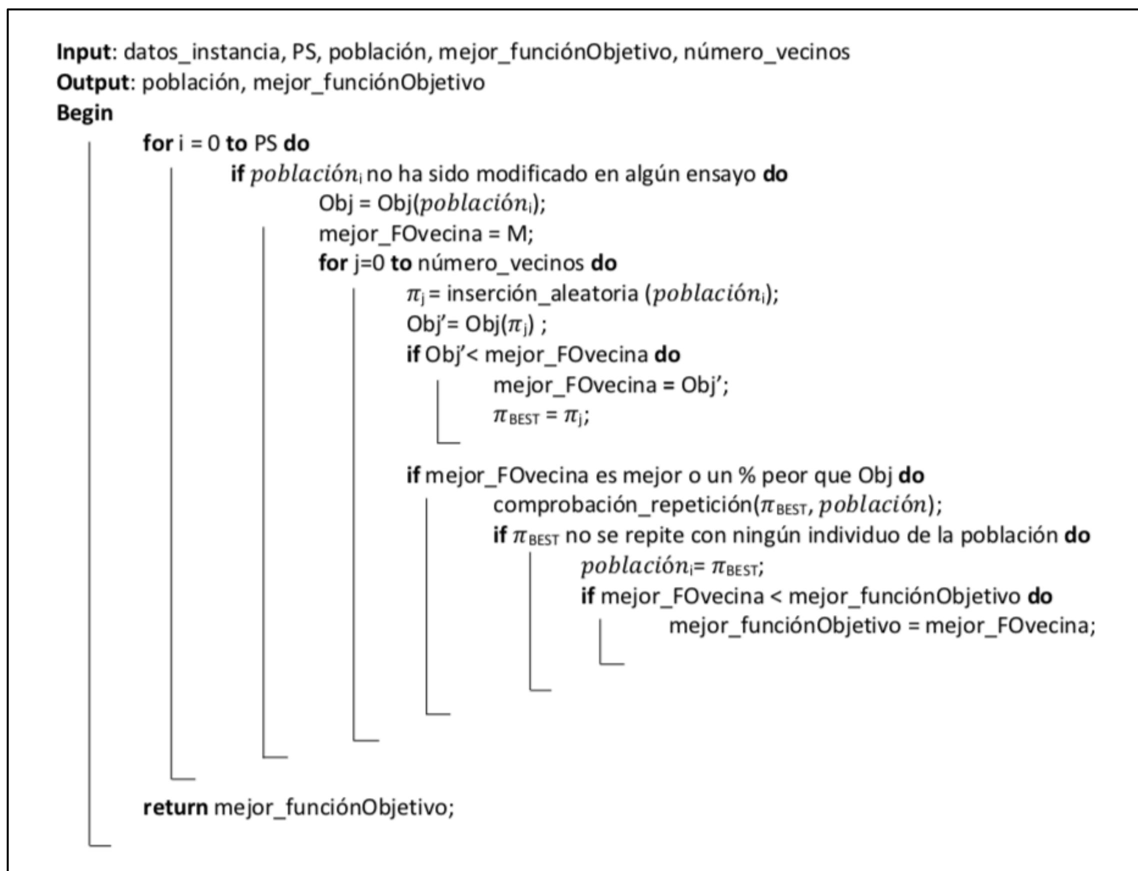


Figura 21. Pseudocódigo fase abeja exploradora mejora 2 (Elaboración propia)

3.6 Mejora 3

La mejora 3 es otra una modificación de la adaptación del algoritmo propuesto por Pan et al. (2014) que se ha desarrollado en el apartado 3.2. En este caso se modifica el operador de vecindad usado.

3.6.1 Determinación de la población inicial

La determinación de la población inicial no presenta diferencias respecto a la mejora 1.

3.6.2 Fases del algoritmo

La diferencia frente a la mejora 1 se observa en la fase de la abeja obrera y la abeja observadora cuyo pseudocódigo aparece en las Figuras 22 y 23. El procedimiento será el mismo, pero se usa

como operador de vecindad la inserción aleatoria, en lugar del intercambio adyacente múltiple. En la inserción se extrae un trabajo de la posición u, elegida de forma aleatoria, para introducirlo en la v que también se elige aleatoriamente.

```
Input: datos_instancia, PS, población, mejor_funciónObjetivo, número_réplicas
Output: población, mejor_funciónObjetivo
Begin
        for i = 0 to número_réplicas do

                for i = 0 to PS do
                        πi = inserción_aleatoria(poblacióni);
                        Obj = Obj(poblacióni);
                        Obj' = Obj(πi );
                        if Obj' < Obj
                                poblacióni = πi
                                if Obj' < mejor_funciónObjetivo do
                                        mejor_funciónObjetivo = Obj';

        return población, mejor_funciónObjetivo
```

Figura 22. Pseudocódigo fase abeja obrera mejora 3 (Elaboración propia)

```
Input: datos_instancia, PS, población, mejor_funciónObjetivo
Output: población, mejor_funciónObjetivo
Begin
        I1 = individuo1_aleatorio(PS);
        I2 = individuo2_aleatorio(PS);
        π = torneo_binario (I1 , I2);
        π' = inserción_aleatoria(π);
        Obj = Obj(π);
        Obj' = Obj(π');
        if Obj'< Obj do
                comprobación_repetición(π', población);
                if π'no se repite con ningun individuo de la población do
                        poblaciónI1/I2= π'
                        If Obj' < mejor_funciónObjetivo do
                                mejor_funciónObjetivo = Obj';

        return población, mejor_funciónObjetivo
```

Figura 23. Pseudocódigo fase abeja observadora mejora 3 (Elaboración propia)

3.7 Mejora 4

El ultimo de los algoritmos presentados modifica de nuevo la adaptación del algoritmo propuesto por Pan et al. (2014) desarrollado en el apartado 3.2. Al igual que en la mejora 3 lo que variará será el operador de vecindad usado.

3.7.1 Determinación de la población inicial

La determinación de la población inicial no presenta diferencias respecto a la mejora 1.

3.7.2 Fases del algoritmo

En comparación con la mejora 1, la única diferencia se muestra en la fase de la abeja obrera y la abeja observadora. En este caso, se usará como operador de vecindad el intercambio de pares en lugar del intercambio adyacente múltiple. En el intercambio de pares, se intercambian los trabajos de la posición u y v, ambas elegidas de forma aleatoria. Los pseudocódigos de ambas fases corresponden con las Figuras 24 y 25.

```
Input: datos_instancia, PS, población, mejor_funciónObjetivo, número_réplicas
Output: población, mejor_funciónObjetivo
Begin
        for i = 0 to número_réplicas do
                for i = 0 to PS do
                        π_i = intercambio_pares(población_i);
                        Obj = Obj(población_i);
                        Obj' = Obj(π_i );
                        if Obj' < Obj
                                población_i = π_i
                                if Obj' < mejor_funciónObjetivo do
                                        mejor_funciónObjetivo = Obj';
        return población, mejor_funciónObjetivo
```

Figura 24. Pseudocódigo fase abeja obrera mejora 4 (Elaboración propia)

```
Input: datos_instancia, PS, población, mejor_funciónObjetivo
Output: población, mejor_funciónObjetivo
Begin
        población_x = individuo_aleatorio(PS);
        poblacion_y = individuo_aleatorio(PS);
        π = torneo_binario (poblacion_x , poblacion_y);
        π' = intercambio_pares(π);
        π'' = peor_individuo_población(población);
        Obj' = Obj(π');
        Obj'' = Obj(π'');
        if Obj'< Obj'' do
                comprobación_repetición(π', población);
                if π'no se repite con ningun individuo de la población do
                        población_x/y= π'
                        If Obj' < mejor_funciónObjetivo do
                                mejor_funciónObjetivo = Obj';
        return población, mejor_funciónObjetivo
```

Figura 25. Pseudocódigo fase abeja observadora mejora 4 (Elaboración propia)

4 EXPERIMENTACIÓN

"Ninguna cantidad de experimentación puede probar definitivamente que tengo razón, pero solo un experimento puede probar que estoy equivocado"
- Albert Einstein -

Un algoritmo de calidad deberá resolver con éxito cualquier situación que pueda presentarse. Por ello, cada una de las cinco versiones del ABC que se han desarrollado se prueban en diversas instancias.

4.1 Generación de las instancias del problema

Como ya se ha explicado en la descripción del problema, el número de turnos en los que se planifican las operaciones, así como los quirófanos disponibles en cada turno son fijos. Sin embargo, el número de cirujanos disponibles o el número de pacientes en lista de espera será distinto según los valores que tomen las variables del problema. Los valores de estas variables son tomados desde el punto de vista de la aplicabilidad al caso real y siguiendo las recomendaciones de la literatura (Molina-Pariente, Hans, et al., 2015).

El número de cirujanos disponibles se determina con la siguiente expresión: $|S| = \alpha \cdot \frac{\sum_{j\in J}\sum_{h\in H} r_{jh}}{l \cdot a \cdot mds}$

- α: factor de control. Varía entre 1.5 y 2.
- r_{jh}: capacidad del quirófano j en el turno h. Su valor es el mismo en todas las instancias, pues se conocen el número de quirófanos disponibles en cada turno y la capacidad de un quirófano en turno de mañana o tarde (Datos reflejados en la descripción del problema del apartado 2).
- l: número de semanas de trabajo en el horizonte de planificación, fijado en 1.
- a: capacidad media de un quirófano. También es un parámetro fijo en todas las instancias. Basta con sumar la capacidad de todos los quirófanos en todos los turnos y dividirla por el número de quirófanos.
- mds: número máximo de turnos semanales en los que puede operar un cirujano. Varía entre 3 y 4.

Así mismo, para determinar el número de cirugías en lista de espera, se irán generando cirugías una a una hasta que la duración de estas supere un porcentaje (β) el tiempo total disponible de los quirófanos en todo el horizonte de planificación. El valor de β variará entre 1 y 1.25.

Para cada combinación de los parámetros variables (α, β, mds), que dan lugar a las distintas situaciones (número de pacientes en lista de espera y cirujanos disponibles), se generan 5

instancias, lo que supone un total de 40 instancias. En cada una de ellas se determina la fecha de lanzamiento, la fecha de vencimiento, la prioridad y la duración de la operación, además del cirujano asociado y los turnos en los que puede operar cada cirujano.

La fecha de lanzamiento de la operación se calcula de forma aleatoria entre el primer y el ultimo turno del horizonte de planificación. Así mismo, la asignación del cirujano a la operación también será aleatoria entre los cirujanos disponibles, pues en la realidad es algo que se determina en la fase de consulta anterior y no está al alcance del documento.

Para calcular la fecha límite o fecha de vencimiento en el que como máximo debería realizarse una cirugía se utiliza la siguiente expresión: (MTBT – dwl) • 2.

- MTBT: tiempo máximo (en días) antes del tratamiento. Depende de la urgencia del paciente. Está definido por los Servicios Nacionales de Atención Médica en función de un conjunto de criterios clínicos (Valente et al., 2009). Toma un valor aleatorio entre {45, 180, 360} como en el Servicio de Salud Español.

- dwl: número de días en lista de espera. Se genera a partir de una distribución uniforme discreta entre [1, MTBT-1] para evitar las fechas negativas.

La prioridad o importancia de cada paciente se obtiene con la siguiente relación: $a \cdot mp^{*} + (1-a) \cdot dwl^{*}$.

- mp^{*}: peso médico normalizado. Se obtiene a partir de una distribución uniforme discreta [1, 5]/5.

- dwl^{*}: número de días en lista de espera normalizado. Para normalizarlo se divide entre el tiempo máximo antes del tratamiento: dwl/MTBT.

- a: importancia de cada indicador. Se establece que ambos indicadores tienen la misma importancia por lo que a=0.5.

Para determinar la duración de cada operación (en horas) se usa una distribución log-normal de dos parámetros: duración esperada (μ), generada aleatoriamente entre los valores {1,2,3,4} (Marcon et al., 2003), y la desviación estándar (σ) que se determina aleatoriamente a partir del intervalo [0.1μ, 0.5μ]. La duración de la operación incluye la preparación de la operación y la limpieza para la siguiente cirugía además del tiempo necesario para realizar la intervención.

Finalmente, para determinar los turnos en los que podrá operar cada cirujano se aplica un procedimiento de dos pasos. En primer lugar, en cada turno se asigna un cirujano de forma aleatoria por cada uno de los quirófanos que haya disponibles. El cirujano podrá asignarse siempre y cuando no haya superado el número máximo de turnos en los que puede operar. Así se evita que ningún quirófano quede inactivo. El siguiente paso consiste en repartir, de forma aleatoria entre los distintos turnos del horizonte, los cirujanos que dispongan de tunos para operar porque no se haya superado el máximo en el paso 1. Un cirujano se asigna a un turno siempre y cuando no se haya asignado ya a ese turno en el paso anterior.

4.2 Determinación de los parámetros del algoritmo

En todas las versiones existe un conjunto de parámetros comunes en la aplicación de los algoritmos para la generación de la secuencia u ordenación de los pacientes. Estos parámetros se han inicializado con los valores que presentan los mejores resultados en la aplicación de la propuesta elaborada por Pan et al., (2014):

- Tiempo computacional: todos los algoritmos serán aplicados en cada instancia durante un intervalo temporal (milisegundos) correspondiente a 10 · número de cirujanos · número de quirófanos presentes.

- Tamaño de la población (PS): La metodología se aplica a una población inicial que se genera de distinta forma según la versión en la que nos encontremos, pero en todos los casos compuesta por 30 individuos. El tamaño ha sido elegido teniendo en cuenta que con un número demasiado pequeño se perderá la credibilidad y en cambio si es demasiado grande los individuos no podrán variarse en diferentes ocasiones dentro del tiempo computacional porque la realización de un ciclo tendrá un tiempo computacional elevado.

- Número de ensayos: Como se explicó en la descripción de la metodología, para la realización de la fase de la abeja exploradora es necesario que la fase de la abeja obrera y observadora se hayan realizado varias veces. Este número de repeticiones o ensayos está fijado en 30.

Además, para el caso de las mejoras correspondientes a la modificación del ABC básico, serán necesarios los siguientes parámetros:

- Réplicas de la fase de la abeja empleada: Para equilibrar la exploración y la explotación presentes en el algoritmo, la fase de la abeja empleada se replica 20 veces.

- Número de vecinos en la fase de exploración: Por cada individuo que no haya mejorado llegados a la fase de exploración se generan 30 vecinos mediante inserción.

Por último, será necesario determinar el porcentaje de empeoramiento (δ) aceptado en la fase de la abeja exploradora de la mejora 2. Como no se conoce la sensibilidad del algoritmo ante los distintos porcentajes de empeoramiento, para calibrar el valor que presenta mejores resultados, se ha probado con δ = 0.25, 0.5, 1, 1.5, 2, 3, 4 y 5% para cada una de las 40 instancias aleatorias generadas siguiendo la sección 4.1.

Para evaluar la efectividad de cada porcentaje aceptado, cuando se trata de problemas donde se valora la tardanza en las soluciones cuyo óptimo tiene valor 0, la medida más común en la literatura es el índice de desviación relativa (RDI) (Vallada et al., 2008). La fórmula para el cálculo es la siguiente: $RDI = (Current_{sol} - Best_{sol}) / (Worst_{sol} - Best_{sol}) \cdot 100$, donde $Current_{sol}$ es la solución para cada instancia usando un posible valor de empeoramiento, $Best_{sol}$ es la mejor solución de cada instancia teniendo en cuenta todos los posibles valores de empeoramiento, y $Worst_{sol}$ es la peor solución de cada instancia teniendo en cuenta todos los posibles valores de empeoramiento. El valor del índice de desviación relativa estará entre 0 y 100, de forma que la mejor solución se da para el caso 0.

En la Tabla 1 se muestran los resultados que se obtienen tras la experimentación. Se observa que, generalmente, los valores que están por debajo del 2% de empeoramiento aceptado no proporcionan buenos resultados, pues presentan un índice de desviación relativa por encima de 0.

Esto podría deberse a que, si se aceptan porcentajes de empeoramiento demasiado pequeños, la exploración se produce en regiones demasiado cercanas a la de la solución óptima en el momento, lo cual impide escapar de los óptimos locales. Como las soluciones tienen valores próximos al 0, una variación menor del 2% es prácticamente insignificante. Por otro lado, los resultados demuestran que aceptar porcentajes de empeoramiento por encima del 4% tampoco proporciona buenos resultados, ya que se está empeorando la solución considerablemente. Por tanto, los valores de empeoramiento aceptables se dan para el 2, 3 y 4 % donde se produce un equilibrio en la aceptación de soluciones peores para explorar nuevas regiones.

α	β	MDS	δ=0.25	δ=0.5	δ=1	δ=1.5	δ=2	δ=3	δ=4	δ=5
1,5			**0**	**0**	**0**	**0**	**0**	**0**	**0**	**5**
	1		**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
		3	0	0	0	0	0	0	0	0
		4	0	0	0	0	0	0	0	0
	1,25		**0**	**0**	**0**	**0**	**0**	**0**	**0**	**10**
		3	0	0	0	0	0	0	0	0
		4	0	0	0	0	0	0	0	20
2			**0**	**5**	**5**	**5**	**0**	**0**	**0**	**5**
	1		**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
		3	0	0	0	0	0	0	0	0
		4	0	0	0	0	0	0	0	0
	1,25		**0**	**10**	**10**	**10**	**0**	**0**	**0**	**10**
		3	0	0	0	0	0	0	0	0
		4	0	20	20	20	0	0	0	20
TOTAL			**0**	**2,5**	**2,5**	**2,5**	**0**	**0**	**0**	**5**

Tabla 1.Valores destacables de RDI de cada porcentaje de empeoramiento

Para determinar qué porcentaje es el que finalmente presenta mejores resultados, se ha vuelto a ejecutar el algoritmo aceptando únicamente empeoramientos del 2, 3 y 4% para otras 40 instancias aleatorias similares generadas siguiendo el procedimiento de la sección 4.1. Aunque se usen los mismos valores de los parámetros variables (α, β, mds), la aletoriedad presente en el procedimiento hace que las instancias que conforman el nuevo banco de prueba sean distintas a las del banco de prueba anterior. Es decir, el número de cirujanos, el número máximo de días en los que el cirujano puede operar, y el porcentaje que la suma de las operaciones supera el tiempo total disponible de los quirófanos si que será igual (por que α, mds y β son los mismos), sin embargo, existen leves diferencias en cuanto al número de operaciones en la lista de espera, ya que cambia la duración de estas. También hay cambios en la fecha de lanzamiento, fecha de vencimiento, prioridad y cirujano asociado de cada operación, así como en los turnos en los que podría operar cada cirujano.

Las soluciones que se obtuvieron de en la segunda calibración se muestran en la Tabla 2. Se puede considerar que el porcentaje de empeoramiento aceptado en la fase de la abeja exploradora que proporciona de media mejores resultados es δ=3, con el que se obtiene un índice de desviación relativa nulo. Los casos δ=2 y δ=4 generan buenos resultados para las instancias en las que el número de cirujanos es pequeño (α = 1.5), con indiferencia del límite de días a la semana que disponen para operar (mds = 3 o 4) o el número de pacientes en lista de espera (β = 1 o 1.25). Sin

embargo, los resultados empeoran cuando el número de cirujanos aumenta ($\alpha = 2$) y el número de pacientes en lista de espera también ($\beta = 1.25$), especialmente si los cirujanos tienen bastantes días a la semana en los que pueden operar (mds = 4).

α	β	MDS	δ=2	δ=3	δ=4
1,5			**0**	**0**	**0**
	1		**0**	**0**	**0**
		3	0	0	0
		4	0	0	0
	1,25		**0**	**0**	**0**
		3	0	0	0
		4	0	0	0
2			**5**	**0**	**5**
	1		**0**	**0**	**0**
		3	0	0	0
		4	0	0	0
	1,25		**10**	**0**	**10**
		3	0	0	0
		4	20	0	20
TOTAL			**2,5**	**0**	**2,5**

Tabla 2. Valores destacables RDI de cada porcentaje de empeoramiento (Segunda calibración)

4.3 Evaluación de las alternativas

Una vez que se han establecido las variables del problema y los parámetros de cada algoritmo, se procede a evaluar las versiones de la metaheurística ABC con cada instancia o situación que pudiese plantearse. Para generar las instancias se seguirá el procedimiento descrito en la sección 4.1, es decir, se generan 40 instancias aleatorias a partir de la combinación de los parámetros variables (α, β, mds) y usando 5 semillas distintas con cada combinación. Como ya se ha explicado en el apartado 4.2, aunque el banco de prueba se genere con los mismos valores de las variables del problema que en otras ocasiones, la aletoriedad del procedimiento consigue que las instancias generadas en este nuevo banco de prueba sean distintas a las que se habían generado anteriormente. Los experimentos se llevaron a cabo en un ordenador con un procesador Intel Core i5 a 2,3 GHz y 8 GB de memoria RAM.

Para analizar la eficiencia de cada uno de ellos se utilizan dos variables de respuesta. Además del RDI utilizado en problemas donde se valora la tardanza, es común en la literatura evaluar las alternativas mediante la desviación porcentual relativa (RPD) (Molina-Pariente, Fernandez-Viagas, et al., 2015). Esta variable de respuesta recoge para cada una de las instancias, la desviación del óptimo teniendo en cuenta todas las alternativas o algoritmos. La fórmula para el cálculo del RPD es la siguiente: RPD = (Currentsol - Bestsol) / Bestsol • 100, donde $Current_{sol}$ es la solución para cada instancia usando un algoritmo determinado y $Best_{sol}$ es la mejor solución cada instancia teniendo en cuenta todos los posibles algoritmos evaluados.

Los resultados que se obtienen tras analizar el RPD de cada algoritmo se indican en la Tabla 3, donde podemos analizar el porcentaje que se desvían respecto al óptimo encontrado las soluciones de cada uno de los algoritmos. La mejora 2 es una de las versiones que de media genera mejores

soluciones con un índice de desviación porcentual nulo. La diferencia respecto a otros algoritmos se presenta en la tercera fase o fase de exploración. En este algoritmo se evita la exploración aleatoria característica de la versión básica, para conseguir que no se analicen regiones vecinas mucho peores y que se perfeccionen las soluciones ya encontradas. Por ello, se explora la mejor de las regiones vecinas generadas tras realizar un conjunto de inserciones aleatorias solo si presenta mejores resultados o los empeora únicamente un 3%. La explotación en este algoritmo se realiza mediante el intercambio adyacente múltiple, donde un trabajo en la posición aleatoria u se intercambia por el de la posición u+1 y el de la posición aleatoria v por el de la v+1. Por otro lado, el segundo algoritmo que también presenta un índice de desviación porcentual nulo es la mejora 4. En este algoritmo, aunque no se impone la condición de que en la fase de exploración las regiones vecinas generadas por inserción tengan que proporcionar soluciones mejores o empeorarlas únicamente un 3%, la explotación mediante el operador de intercambio de pares, donde un trabajo en la posición aleatoria u se intercambia por otro de la posición aleatoria v, consigue que se terminen perfeccionando las soluciones. De esta forma se evita el estancamiento en óptimos locales.

Si nos adentramos en los escenarios concretos que pueden plantearse, es cierto que cuando el número de cirujanos es pequeño (α = 1.5), con indiferencia del límite de días a la semana que disponen para operar (mds = 3 o 4) o el número de pacientes en lista de espera (β = 1 o 1.25), ninguna de las versiones destaca por generar mejores soluciones que el resto. Sin embargo, cuando el número de cirujanos aumenta (α = 2) y el número de pacientes en lista de espera también (β = 1.25), especialmente si los cirujanos tienen bastantes días a la semana en los que pueden operar (mds = 4), tanto la versión básica como la mejora 1 y 3 proporcionan peores resultados. La aleatoriedad en la fase de exploración de la versión básica hace que se exploren regiones mucho peores que no consiguen mejorarse con la explotación mediante el intercambio adyacente múltiple. De hecho, es el algoritmo que peores resultados presenta con una desviación porcentual relativa media respecto al óptimo encontrado de 0.41%. Así mismo, aunque en la mejora 1 se evita la exploración aleatoria, la vecindad que se analiza es la que presenta mejores resultados tras realizar un conjunto de inserciones aleatorias, sin comprobar si es una región que perfecciona o empeora considerablemente la solución ya encontrada. Este hecho no es suficiente cuando el operador que se usa en la explotación es el intercambio adyacente múltiple, porque no consigue perfeccionar finalmente las soluciones si se han empeorado considerablemente en la exploración. Lo mismo ocurre con la mejora 3 cuando se usa el operador de inserción aleatoria.

α	β	MDS	ABC básico	Mejora 1	Mejora 2	Mejora 3	Mejora 4
1,50			**0**	**0**	**0**	**0**	**0**
	1,00		**0**	**0**	**0**	**0**	**0**
		3	0	0	0	0	0
		4	0	0	0	0	0
	1,25		**0**	**0**	**0**	**0**	**0**
		3	0	0	0	0	0
		4	0	0	0	0	0
2,00			**16,67**	**5,56**	**0**	**5,56**	**0**
	1,00		**0**	**0**	**0**	**0**	**0**
		3	0	0	0	0	0
		4	0	0	0	0	0
	1,25		**30**	**10**	**0**	**10**	**0**
		3	20	0	0	0	0
		4	40	20	0	20	0
TOTAL			**7,50**	**2,50**	**0,00**	**2,50**	**0**

Tabla 3. Valores destacables de RPD de cada algoritmo

En la Tabla 4 se muestran los resultados de la evaluación de las alternativas cuando se analiza el índice de desviación relativa. Se vuelve a confirmar, al igual que en el caso anterior, que las mejores versiones en media son las mejoras 2 y 4 con un RDI nulo. Así mismo, los peores resultados se generan con la versión básica. Con este operador normalizado podemos evaluar de 0 a 100 cómo empeoran los algoritmos respecto a la mejor de las versiones que presenta valor 0.

α	β	MDS	ABC básico	Mejora 1	Mejora 2	Mejora 3	Mejora 4
1,5			**0**	**0**	**0**	**0**	**0**
	1		**0**	**0**	**0**	**0**	**0**
		3	0	0	0	0	0
		4	0	0	0	0	0
	1,25		**0**	**0**	**0**	**0**	**0**
		3	0	0	0	0	0
		4	0	0	0	0	0
2			**0,91**	**0,33**	**0**	**0,33**	**0**
	1		**0**	**0**	**0**	**0**	**0**
		3	0	0	0	0	0
		4	0	0	0	0	0
	1,25		**1,63**	**0,59**	**0**	**0,59**	**0**
		3	0,91	0	0	0	0
		4	2,35	1,18	0	1,18	0
TOTAL			**0,41**	**0,15**	**0**	**0,15**	**0**

Tabla 4. Valores destacables de RDI de cada algoritmo

Para determinar cuál es el mejor algoritmo, aunque la mejora 2 y 4 han proporcionado de media los mejores resultados en las instancias que simulan escenarios parecidos a los que se dan en el caso real, se procede a evaluar las alternativas en otro banco de prueba distinto. El nuevo banco de prueba recoge escenarios que simulan una mayor congestión del sistema. En este caso, las instancias que lo componen se generarán de forma similar al apartado 4.1, es decir, el procedimiento será el mismo pero el valor que toman las variables del problema será distinto. A diferencia de α y mds que seguirán tomando los mismos valores (α = 1.5, 2 y mds = 3, 4), β tomará los valores de 1.50, 1.75 y 2. De esta forma se consigue simular la congestión aumentando considerablemente el número de pacientes en lista de espera. En definitiva, se evaluarán las alternativas en 60 nuevas instancias, que con las 40 anteriores suponen un global de 100.

Al igual que en el caso anterior, los algoritmos son evaluados a partir de la desviación porcentual relativa (RPD) y del índice de desviación relativa (RDI). Los resultados de ambas evaluaciones se muestran en la Tabla 5 y 6 respectivamente. Como se puede observar, en esta nueva situación, son la mejora 3 y especialmente la 4, quienes proporcionan en media los mejores resultados con una desviación media respecto al óptimo de 0.09 y 0.08% respectivamente.

De la comparación a partir del índice de desviación relativa en la Tabla 5, donde se puede evaluar cada algoritmo de forma normalizada, podemos determinar que la mejora 2 no proporciona resultados tan buenos como la mejora 4 cuando la congestión es elevada, especialmente si el número de cirujanos también lo es (α = 2). Por tanto, aunque en una situación similar a que presenta el Hospital Virgen del Rocío en su normalidad, representada en la primera experimentación, tanto la mejora 2 como la 4 podían considerarse los mejores algoritmos, cuando aumenta el colapso del sistema es finalmente la mejora 4 quien proporciona los mejores resultados, por lo que podría determinarse como la mejor entre las evaluaciones con un RDI de 0.02, muy próximo a 0.

α	β	MDS	ABC básico	Mejora 1	Mejora 2	Mejora 3	Mejora 4
1,5			**0,72**	**0,09**	**0,20**	**0,09**	**0**
	1,5		**0,83**	**0**	**0**	**0**	**0**
		3	0,83	0	0	0	0
		4	0,83	0	0	0	0
	1,75		**0,59**	**0,26**	**0,59**	**0**	**0**
		3	0	0	0	0	0
		4	1,17	0,53	1,17	0	0
	2		**0,75**	**0**	**0**	**0,28**	**0**
		3	0	0	0	0	0
		4	1,50	0	0	0,56	0
2			**1,34**	**0,61**	**0,54**	**0,08**	**0,16**
	1,5		**0,48**	**0,48**	**0,48**	**0**	**0,48**
		3	0	0	0	0	0
		4	0,95	0,95	0,95	0	0,95
	1,75		**1,69**	**0,66**	**0,67**	**0**	**0**
		3	0	0	0	0	0
		4	3,38	1,31	1,33	0	0
	2		**1,84**	**0,70**	**0,48**	**0,24**	**0**
		3	0	0	0	0	0
		4	3,69	1,41	0,96	0,48	0
TOTAL			**1,03**	**0,35**	**0,37**	**0,09**	**0,08**

Tabla 5. Valores destacables de RPD de cada algoritmo (Segunda experimentación)

α	β	MDS	ABC básico	Mejora 1	Mejora 2	Mejora 3	Mejora 4
1,5			**0,23**	**0,03**	**0,07**	**0,03**	**0**
	1,5		**0,20**	**0**	**0**	**0**	**0**
		3	0,20	0	0	0	0
		4	0,20	0	0	0	0
	1,75		**0,20**	**0,10**	**0,20**	**0**	**0**
		3	0	0	0	0	0
		4	0,40	0,20	0,40	0	0
	2		**0,30**	**0**	**0**	**0,10**	**0**
		3	0	0	0	0	0
		4	0,60	0	0	0,20	0
2			**0,30**	**0,17**	**0,13**	**0,02**	**0,03**
	1,5		**0,10**	**0,10**	**0,10**	**0**	**0,10**
		3	0	0	0	0	0
		4	0,20	0,20	0,20	0	0,20
	1,75		**0,30**	**0,20**	**0,13**	**0**	**0**
		3	0	0	0	0	0
		4	0,60	0,40	0,27	0	0
	2		**0,50**	**0,20**	**0,15**	**0,05**	**0**
		3	0	0	0	0	0
		4	1	0,40	0,30	0,10	0
TOTAL			**0,27**	**0,10**	**0,10**	**0,03**	**0,02**

Tabla 6. Valores destacables de RDI de cada algoritmo (Segunda experimentación)

5 CONCLUSIONES

“Es mejor debatir una cuestión sin llegar a concluirla, que llegar a una conclusión sin debatirla”
- Joseph Joubert -

Para abordar el problema de programación de quirófanos a nivel operativo, en un horizonte temporal de una semana, se han propuesto cinco algoritmos que son versiones adaptadas y/o diseñadas de la metaheurística “Artificial Bee Colony”. El objetivo principal en todos ellos ha sido establecer, para cada paciente de la lista de espera, un quirófano disponible en un turno de operación lo más pronto posible, que se encuentre dentro del horizonte de planificación y cumpla con las restricciones de los recursos (cirujanos y quirófanos). De esta forma se consigue reducir el retraso global de las operaciones. Además, se ha añadido un componente secundario a la Función Objetivo que reduce el número de pacientes que no se operan dentro del periodo que se programa, debido a que el turno de operación establecido sobrepasa el horizonte de planificación. Con este segundo componente, se consigue operar el mayor número de pacientes posibles en caso de que la tardanza no pueda ser reducida aun más.

A la hora de evaluar la eficiencia de las metaheurísticas implementadas, se ha generado un primer banco de pruebas que representa la situación general que se desarrolla en la especialidad de Cirugía Plástica y Quemaduras Mayores del Hospital Virgen del Rocío. El primer banco de prueba se formó a partir de 40 instancias fruto de la combinación de los parámetros variables del problema ($\alpha = 1.5, 2$, β = 1,1.25 y mds = 3,4), y del uso de 5 semillas distintas por cada posible combinación. Por otro lado, se han evaluado las alternativas en un segundo banco de pruebas, formado por 60 instancias, que simula una mayor congestión. Para ello se han modificado los valores de las variables del problema. ($\alpha = 1.5, 2$, β = 1.5, 1.75, 2 y mds = 3,4).

Los experimentos computacionales demuestran que, analizando desviación porcentual relativa y el índice de desviación relativo, las mejores versiones de media para el primer banco de prueba son la mejora 2 y la mejora 4, dos modificaciones del algoritmo propuesto por Pan et al. (2014). Sin embargo, en el segundo banco de prueba se evidencia que, cuando la congestión aumenta y el número de pacientes en lista de espera se incrementa, la mejora 2 no proporciona tan buenos resultados como la mejora 4. Este hecho podría deberse a que, como en la mejora 2 se controla el porcentaje de empeoramiento para no aceptar soluciones mucho peores, cuando aumenta la congestión, aumenta la multiplicidad y diversidad de los pacientes, y si se acepta únicamente un 3% de empeoramiento, se estaría limitando el proceso de búsqueda que ahora sería mucho más amplio, lo cual dificulta la posibilidad de encontrar la solución óptima global.

En definitiva, se constata que la mejor programación de los quirófanos en media se consigue cuando se usa el algoritmo en el que la explotación se produce mediante el intercambio de pares y la exploración mediante inserción aleatoria, sin la necesidad de restringir el porcentaje de empeoramiento aceptado en esta útlima fase. En cualquier caso, los experimentos computacionales demuestran también que los algoritmos en los que la población se inicializa con heurísticas proporcionan siempre mejores soluciones que en los que se inicializa de forma aleatoria.

Como búsqueda constante de mejora, aunque de media se haya determinado que la mejora 4 es el mejor algoritmo, se podrá estudiar en líneas futuras si lo siguen siendo también estadísticamente. Por otro lado, sería también sugerente observar el comportamiento de la metaheurística utilizando nuevos operadores de vecindad que se desarrollen. Finalmente, para que el documento pueda servir de análisis previo a la prueba de los algoritmos en el entorno real del hospital, podría estudiarse la influencia de la fase de consulta anterior en la programación de los quirófanos.

6 BIBLIOGRAFÍA

Cardoen, B., Demeulemeester, E., & Beliën, J. (2010). Operating room planning and scheduling: A literature review. *European Journal of Operational Research*, *201*(3), 921–932. https://doi.org/10.1016/j.ejor.2009.04.011

Entender ITIL 2011 Normas y mejores prácticas para avanzar hacia ISO 20000 - Los niveles decisionales. (n.d.). Retrieved June 19, 2020, from https://www.ediciones-eni.com/open/mediabook.aspx?idR=1d149467a559fd2acbc4aab12e413dd5

Fei, H., Meskens, N., & Chu, C. (2010). A planning and scheduling problem for an operating theatre using an open scheduling strategy. *Computers and Industrial Engineering*, *58*(2), 221–230. https://doi.org/10.1016/j.cie.2009.02.012

Framinan, J. M., Leisten, R., & Ruiz García, R. (2014). Manufacturing Scheduling Systems. In *Manufacturing Scheduling Systems*. https://doi.org/10.1007/978-1-4471-6272-8

Ioannou, C. C. (2017). Swarm intelligence in fish? The difficulty in demonstrating distributed and self-organised collective intelligence in (some) animal groups. In *Behavioural Processes* (Vol. 141, pp. 141–151). Elsevier B.V. https://doi.org/10.1016/j.beproc.2016.10.005

Karaboga, D., & Basturk, B. (2008). On the performance of artificial bee colony (ABC) algorithm. *Applied Soft Computing Journal*, *8*(1), 687–697. https://doi.org/10.1016/j.asoc.2007.05.007

Karaboga, Dervis, & Basturk, B. (2007). A powerful and efficient algorithm for numerical function optimization: Artificial bee colony (ABC) algorithm. *Journal of Global Optimization*, *39*(3), 459–471. https://doi.org/10.1007/s10898-007-9149-x

Macario, A., Vitez, T. S., Dunn, B., & McDonald, T. (1995). Where are the costs in perioperative care?: Analysis of hospital costs and charges for inpatient surgical care. *Anesthesiology*, *83*(6), 1138–1144. https://doi.org/10.1097/00000542-199512000-00002

Marcon, E., Kharraja, S., & Simonnet, G. (2003). The operating theatre planning by the follow-up of the risk of no realization. *International Journal of Production Economics*, *85*(1), 83–90. https://doi.org/10.1016/S0925-5273(03)00088-4

Metaheurísticas | Peralta-Abarca | Inventio, la génesis de la cultura universitaria en Morelos. (n.d.). Retrieved June 20, 2020, from http://inventio.uaem.mx/index.php/inventio/article/view/576/1226

Molina-Pariente, J. M., Fernandez-Viagas, V., & Framinan, J. M. (2015). Integrated operating room planning and scheduling problem with assistant surgeon dependent surgery durations. *Computers and Industrial Engineering*, *82*, 8–20. https://doi.org/10.1016/j.cie.2015.01.006

Molina-Pariente, J. M., Hans, E. W., Framinan, J. M., & Gomez-Cia, T. (2015). New heuristics for planning operating rooms. *Computers & Industrial Engineering*, *90*, 429–443. https://doi.org/10.1016/J.CIE.2015.10.002

Nawaz, M., Enscore, E. E., & Ham, I. (1983). A heuristic algorithm for the m-machine, n-job flow-shop sequencing problem. *Omega*, *11*(1), 91–95. https://doi.org/10.1016/0305-0483(83)90088-9

Pan, Q. K., Wang, L., Li, J. Q., & Duan, J. H. (2014). A novel discrete artificial bee colony algorithm for the hybrid flowshop scheduling problem with makespan minimisation. *Omega (United Kingdom)*, *45*, 42–56. https://doi.org/10.1016/j.omega.2013.12.004

Pan, Q. K., Wang, L., Mao, K., Zhao, J. H., & Zhang, M. (2013). An effective artificial bee colony algorithm for a real-world hybrid flowshop problem in steelmaking process. *IEEE Transactions on Automation Science and Engineering*, *10*(2), 307–322. https://doi.org/10.1109/TASE.2012.2204874

Schiavinotto, T., & Stützle, T. (2007). A review of metrics on permutations for search landscape analysis. *Computers and Operations Research*, *34*(10), 3143–3153.

https://doi.org/10.1016/j.cor.2005.11.022

Vallada, E., Ruiz, R., & Minella, G. (2008). Minimising total tardiness in the m-machine flowshop problem: A review and evaluation of heuristics and metaheuristics. *Computers and Operations Research*, *35*(4), 1350–1373. https://doi.org/10.1016/j.cor.2006.08.016

7 ANEXOS

7.1 Código función principal

```
Using System;
using System.Diagnostics;
using System.IO;

namespace EvaluacionAlternativas
{
   class MainClass
   {
      public static void Main(string[] args)
      {
         // PARÁMETROS FIJOS DEL PROBLEMA

         int dias = 5; // El horizonte temporal para la programación de las operaciones es de 5 días = 10 turnos (mañana y tarde)

         int numero_quirofanos = 3; // Hay 3 quirófanos, pero no están siempre disponibles. Depende del día de la semana y si es turno de mañana o de tarde

         int[,] disponibilidadquirofano_turno = new int[numero_quirofanos, dias * 2]; // 1 = quirófano disponible en el turno, 0 = quirófano no disponible en el turno. Como se programan una semana de lunes a viernes tanto por la mañana como por la tarde, los quirófanos que están disponibles en cada turno son:

         for (int j = 0; j < dias * 2; j++)
         {
               // Para lunes y martes por la mañana: 3

               if (j == 0 || j == 2)
               {
                     for (int i = 0; i < numero_quirofanos; i++)
                     {
                           disponibilidadquirofano_turno[i, j] = 1;
                     }
               }

               // Para lunes y martes por la tarde y miércoles, jueves y viernes por la mañana: 2

               else if (j == 1 || j == 3 || j == 4 || j == 6 || j == 8)
               {
                     disponibilidadquirofano_turno[0, j] = 1;
                     disponibilidadquirofano_turno[1, j] = 1;
                     disponibilidadquirofano_turno[2, j] = 0;
               }

               // Para miércoles jueves y viernes por la tarde: 1

               else
               {
                     disponibilidadquirofano_turno[0, j] = 1;
                     disponibilidadquirofano_turno[1, j] = 0;
                     disponibilidadquirofano_turno[2, j] = 0;
               }
         }
```

```
double[] horasmaximaquirofano_turno = new double[] { 6.5, 4, 6.5, 4, 6.5, 4, 6.5, 4, 6.5, 4 }; // El número de horas máxima que un quirófano está operativo solo depende de si es turno de mañana o tarde (Mañanas de 8:30 a 15:00 y tardes de 15:00 a 20:00)

// PARÁMETROS FIJOS DEL ALGORITMO

int n = 30; // La población está formada por 30 individuos. Cada uno de ellos será una secuencia (orden) de la lista de los pacientes a operar.

int ensayos = 30; // Número de iteraciones consecutivas anteriores a la fase de exploración

int replicas = 20; // Réplicas de la etapa de la abeja obrera

int numero_vecinos = 30; // Número de vecinos que se generan para un individuo abandonado en la fase de exploración

// PARÁMETROS VARIABLES DEL PROBLEMA

double[] alpha = new double[] { 1.5, 2 }; // Factor para determinar el número de cirujanos

int[] mds = new int[] { 3, 4 }; // Número máximo de turnos por semana en los que el cirujano está disponible para realizar cirugías. Se usa para determinar el número de cirujanos

double[] beta = new double[] { 1, 1.25 }; // Porcentaje que la suma de las duraciones de las operaciones excede el tiempo total disponible en los quirófanos. Se usa para determinar el número de cirugías en lista de espera

// EXPORTANCIÓN A EXCEL

String ruta = AppDomain.CurrentDomain.BaseDirectory;
ruta = ruta.Replace("\\", "/").ToString();
DirectoryInfo DIRESC = new DirectoryInfo(ruta + "/temp");
if (!DIRESC.Exists)
{
        DIRESC.Create();
}
String ficBatBorrar = ruta + "temp" + "/Experimentacion.csv";
FileInfo FicheroEliminar = new FileInfo(ficBatBorrar);
if (FicheroEliminar.Exists)
{
        File.Delete(ficBatBorrar);
}
String ficCSV = ruta + "temp" + "/Experimentacion.csv";
System.IO.StreamWriter FicheroCSV1 = new System.IO.StreamWriter(ficCSV, true);

// GENERACIÓN DE INSTANCIAS

for (int a = 0; a < alpha.Length; a++)
{
        for (int b = 0; b < beta.Length; b++)
        {
                for (int c = 0; c < mds.Length; c++)
                {
                        for (int d = 0; d < 5; d++)
                        {
                                // DATOS DE LA INSTANCIA

                                int m; // Cantidad de pacientes en lista de espera
                                int numero_cirujanos;
                                double[] duracion_operacion;
                                int[] dia_disponible; // Fecha de lanzamiento de las operaciones
                                int[] fecha_limite;
```

```
                                    double[] prioridad;
                                    int[] cirujano_asociado;
                                    double[,] horascirujano_turno; // Indica la disponibilidad de un cirujano en un turno

                                    generacion_datos(dias, alpha[a], beta[b], mds[c], d, numero_quirofanos, disponibilidadquirofano_turno, out duracion_operacion, out dia_disponible, out fecha_limite, out prioridad, out cirujano_asociado, out horascirujano_turno, out m, out numero_cirujanos);

                                    ABC_basico(n, ensayos, dias, alpha[a], beta[b], mds[c], d, numero_quirofanos, disponibilidadquirofano_turno, duracion_operacion, dia_disponible, fecha_limite, out prioridad, cirujano_asociado, horascirujano_turno, m, numero_cirujanos, horasmaximaquirofano_turno, FicheroCSV1);

                                    mejora1(n, ensayos, dias, alpha[a], beta[b], mds[c], d, numero_quirofanos, disponibilidadquirofano_turno, duracion_operacion, dia_disponible, fecha_limite, out prioridad, cirujano_asociado, horascirujano_turno, m, numero_cirujanos, horasmaximaquirofano_turno, FicheroCSV1);

                                    mejora2(n, ensayos, dias, alpha[a], beta[b], mds[c], d, numero_quirofanos, disponibilidadquirofano_turno, duracion_operacion, dia_disponible, fecha_limite, out prioridad, cirujano_asociado, horascirujano_turno, m, numero_cirujanos, horasmaximaquirofano_turno, FicheroCSV1);

                                    mejora3(n, ensayos, dias, alpha[a], beta[b], mds[c], d, numero_quirofanos, disponibilidadquirofano_turno, duracion_operacion, dia_disponible, fecha_limite, out prioridad, cirujano_asociado, horascirujano_turno, m, numero_cirujanos, horasmaximaquirofano_turno, FicheroCSV1);

                                    mejora4(n, ensayos, dias, alpha[a], beta[b], mds[c], d, numero_quirofanos, disponibilidadquirofano_turno, duracion_operacion, dia_disponible, fecha_limite, out prioridad, cirujano_asociado, horascirujano_turno, m, numero_cirujanos, horasmaximaquirofano_turno, FicheroCSV1);
                                }
                            }
                        }
                    }

                    FicheroCSV1.Close();
                }
            }
        }
```

7.2 Código funciones auxiliaries

```
public static void generacion_datos(int dias, double alpha, double beta, int mds, int instancia, int numero_quirofanos, int[,] disponibilidadquirofano_turno, out double[] duracion_operacion, out int[] dia_disponible, out int[] fecha_limite, out double[] prioridad, out int[] cirujano_asociado, out double[,] horascirujano_turno, out int m, out int numero_cirujanos)
        {
            Random aleatorio = new Random(instancia);
```

```
// NÚMERO CIRUJANOS

int l = 1; // Número de semanas de trabajo en el horizonte de planificación

double SumRjh = (3 * 6.5) + (2 * 4) + (3 * 6.5) + (2 * 4) + (2 * 6.5) + (1 * 4) + (2 * 6.5) + (1 * 4) + (2 * 6.5)
+ (1 * 4); // Sumatorio de la capacidad de los quirófanos en todo el horizonte de planificación

double a = SumRjh / 19; // Capacidad media de un quirófano

double cirujanos = alpha * (SumRjh / (l * a * mds));

numero_cirujanos = (int)Math.Round(cirujanos);

// NÚMERO DE PACIENTES EN LISTA DE ESPERA

m = 1;
duracion_operacion = new double[m];
double aux = 0;

while (aux < beta * SumRjh)
{
        int mean = aleatorio.Next(1, 5); // La  media esperada tomará un valor aleatorio entre 1, 2, 3 o 4
        horas

        double SD = mean * ((aleatorio.Next(0, 1) * 0.4) + 0.1); // El coeficiente de variación se genera
        aleatoriamente a partir del intervalo [0.1µ. . .0.5µ].

        double cv = SD / mean;
        double variance = Math.Pow(mean * cv, 2.0);
        double mu = Math.Log(mean / Math.Sqrt(1 + variance / (mean * mean)));
        double sigma = Math.Sqrt(Math.Log(1 + variance / (mean * mean)));

        MathNet.Numerics.Distributions.NormalnormalDist=
        newMathNet.Numerics.Distributions.Normal(mu, sigma);
        double randomGaussianValue = normalDist.Sample();
        duracion_operacion[m - 1] = Math.Exp(randomGaussianValue);
        duracion_operacion[m - 1] = Math.Round(duracion_operacion[m - 1], 2); // Redondeo

        aux = aux + duracion_operacion[m - 1];
        m++;
        Array.Resize(ref duracion_operacion, m);
}

// CIRUJANOS DISPONIBLES EN CADA TURNO

horascirujano_turno = new double[numero_cirujanos, dias * 2];

int[] aux2 = new int[numero_cirujanos];
for (int i = 0; i < dias * 2; i++)
{
        for (int j = 0; j < numero_quirofanos; j++)
        {
                // En cada uno de los quirófanos disponibles de cada turno

                if (disponibilidadquirofano_turno[j, i] == 1)
                {
                        int aux3 = -1;
                        while (aux3 == -1)
                        {
                                // se asigna aleatoriamente un cirujano

                                int cirujano_elegido = aleatorio.Next(0, numero_cirujanos);
```

```
                        if (aux2[cirujano_elegido] < mds - 1) // Tiene que cumplirse que pueda asignarse a un turno una vez más para poder usar ese cirujano
                        {
                            horascirujano_turno[cirujano_elegido, i] = 6.5;
                            aux2[cirujano_elegido]++;
                            aux3 = 0;
                        }
                    }
                }
            }
        }

// Si el cirujano puede operar en algún otro turno

for (int i = 0; i < numero_cirujanos; i++)
{
    while (aux2[i] < mds - 1)
    {
        int aux4 = -1;

        // Se escoje un turno aleatorio de entre los que no tenía asignado

        while (aux4 == -1)
        {
            int aux5 = aleatorio.Next(0, dias * 2);
            if (horascirujano_turno[i, aux5] == 0)
            {
                horascirujano_turno[i, aux5] = 6.5;
                aux2[i]++;
                aux4 = 0;
            }
        }
    }
}

// DÍA DISPONIBLE-FECHA LÍMITE-PRIORIDAD-CIRUJANO ASOCIADO

dia_disponible = new int[m];
fecha_limite = new int[m];
prioridad = new double[m];
cirujano_asociado = new int[m];

for (int i = 0; i < m; i++)
{
    // Asignación cirujano de forma aleatoria

    cirujano_asociado[i] = aleatorio.Next(0, numero_cirujanos);

    // Imposición del release date de forma aleatoria

    dia_disponible[i] = aleatorio.Next(0, dias); // No tiene sentido considerar operaciones que no estén disponibles en algún instante dentro del horizonte de planificación

    // Cálculo fecha límite

    int mtbt;// Tiempo máximo antes del tratamiento (dias). Se genera aleatoriamente a partir del conjunto {45, 180, 360} como en el Servicio de Salud Español
    int[] vectoraux = new int []{ 45, 180, 360 };
    mtbt = vectoraux[aleatorio.Next(0, 3)];

    double dwl = aleatorio.Next(1, mtbt-1); // El número de días en la lista de espera (dwl) se extrae de una distribución uniforme discreta [1, MTBT-1] para evitar las últimas fechas negativas
```

```
                fecha_limite[i] = (int)(mtbt - dwl);

                // Cálculo prioridad

                double mp = aleatorio.Next(1, 6); // Mp se genera a partir de una distribución uniforme discreta [1,
                5], siendo 5 la máxima prioridad
                mp = mp / 5;
                dwl = dwl / mtbt;
                prioridad[i] = 0.5 * mp + ((1 - 0.5) * dwl);
                prioridad[i] = Math.Round(prioridad[i], 2); // Redondeo
            }
        }

public static void ABC_basico(int n, int ensayos, int dias, double alpha, double beta, int mds, int instancia, int
numero_quirofanos, int[,] disponibilidadquirofano_turno, double[] duracion_operacion,  int[] dia_disponible, int[]
fecha_limite, double[] prioridad, int[] cirujano_asociado, double[,] horascirujano_turno, int m,  int numero_cirujanos,
double[,] horasmaxima_turno, StreamWriter FicheroCSV1)
        {
            // VARIABLES NECESARIAS PARA LA APLICACIÓN DEL ALGORITMO

            // Matriz que recoge todos los individuos que componen la población

            int[,] poblacion = new int[n, m];

            // Vector que guarda la secuencia que se va a incorporar en la población y su F.O.

            int[] secuencia = new int[m];
            double FO1 = 0;

            // Vector que guarda la secuencia vecina que se genera en el algoritmo y su F.O.

            int[] vecina = new int[m];
            double FO2 = 0;

            // Matriz necesaria para generar una secuencia en función de un orden

            int[,] individuo = new int[2, m];
            int[,] individuosalida = new int[2, m];

            // Vector que recoge la mejor secuencia hasta el momento y su posición en la población que se usará en la
            fase de la abeja observadora

            int[] mejor_secuencia = new int[m];
            double mejorFO = 999; // Inicializada a un valor alto para que la mejor secuencia sea el primer individuo que
            se incorpora a la población (la primera secuencia que se comprueba) y sirva esta de comparativa con el resto
            int mejor_posicion = -1;

            // GENERACIÓN DE LA POBLACIÓN

            // Para cada individuo de la población,

            for (int i = 0; i < n; i++)
            {
                // se asocian unos tiempos de proceso aleatorios a cada paciente

                for (int j = 0; j < m; j++)
                {
                    individuo[0, j] = j;
                    Random aleatorio = new Random();
                    individuo[1, j] = aleatorio.Next(1, 50);
                }
```

```
            // Se obtiene una secuencia ordenada de pacientes en función de estos tiempos aleatorios que será
            una secuencia aleatoria

            individuosalida = int_Quicksort_decre(individuo, 0, m - 1);
            rellenar_vector(individuosalida, secuencia, 0);

            // Se añade la secuencia a la población

            rellenar_fila_matriz(poblacion, secuencia, i);

            // Para guardar la mejor secuencia generada

            FO1=Sum_tiempo_acceso_ponderado(dia_disponible, fecha_limite, duracion_operacion,
            prioridad, cirujano_asociado, numero_quirofanos, disponibilidadquirofano_turno,
            horasmaximaquirofano_turno, numero_cirujanos, horascirujano_turno, dias, secuencia);

            if (FO1 < mejorFO)
            {
                Array.Copy(secuencia,mejor_secuencia, secuencia.Length);
                mejorFO = FO1;
                mejor_posicion = i;
            }
        }

        // APLICACIÓN DE LAS 3 FASES DEL ALGORITMO (CICLO)

        // Rescate del tiempo actual. Se necesita para repetir las tres fases (ciclo) hasta llegar al tiempo límite

        var timer = Stopwatch.StartNew();
        double tiempoActualMilisegundos = timer.ElapsedMilliseconds;
        double tiempoActual = tiempoActualMilisegundos / 1000;

        int tiempoLimite = 10 * numero_quirofanos * numero_cirujanos;
        while (tiempoActual < tiempoLimite)
        {
            // Vector que recoge si un individuo ha mejorado o no en un ciclo. Es necesario en la fase de la abeja
            exploradora

            bool[] mejora = new bool[n];

            // Bucle que recoge el número de iteraciones o ensayos anteriores a la fase de exploración

            for (int y = 0; y < ensayos; y++)
            {
                // FASE DE LA ABEJA OBRERA

                // Para cada individuo de la población,

                for (int i = 0; i < n; i++)
                {
                    rellenar_vector(poblacion, secuencia, i);
                    rellenar_vector(poblacion, vecina, i);

                    // se genera un vecino aplicando el intercambio múltiple de trabajos adyacentes

                    intercambio_multiple(vecina);

                    // Se acepta el vecino solo si es mejor

                    FO1 = Sum_tiempo_acceso_ponderado(dia_disponible, fecha_limite,
                    duracion_operacion, prioridad, cirujano_asociado, numero_quirofanos,
```

```
                disponibilidadquirofano_turno, horasmaximaquirofano_turno, numero_cirujanos,
                horascirujano_turno, dias, secuencia);

                FO2 = Sum_tiempo_acceso_ponderado(dia_disponible, fecha_limite,
                duracion_operacion, prioridad, cirujano_asociado, numero_quirofanos,
                disponibilidadquirofano_turno, horasmaximaquirofano_turno, numero_cirujanos,
                horascirujano_turno, dias, vecina);

                if (FO2 < FO1)
                {
                    mejora[i] = true;
                    rellenar_fila_matriz(poblacion, vecina, i);

                    // En el caso de que sea mejor y se incorpore a la población, se comprueba
                    si es mejor que la mejor_secuencia que estaba recogida y si es así, se
                    actualiza

                    if (FO2 < mejorFO)
                    {
                        Array.Copy(vecina, mejor_secuencia, secuencia.Length);
                        mejorFO = FO2;
                        mejor_posicion = i;
                    }
                }
            }

            // FASE DE LA ABEJA OBSERVADORA

            // Al mejor individuo de la población,

            Array.Copy(mejor_secuencia, vecina, mejor_secuencia.Length);

            // se le aplica el mismo procedimiento que en la fase anterior. Se genera un vecino mediante
            el intercambio adyacente múltiple

            intercambio_multiple(vecina);

            // Se acepta el vecino si es mejor

            FO2 = Sum_tiempo_acceso_ponderado(dia_disponible, fecha_limite, duracion_operacion,
            prioridad, cirujano_asociado, numero_quirofanos, disponibilidadquirofano_turno,
            horasmaximaquirofano_turno, numero_cirujanos, horascirujano_turno, dias, vecina);

            if (FO2 < mejorFO)
            {
                mejora[mejor_posicion] = true;
                rellenar_fila_matriz(poblacion,vecina, mejor_posicion);

                // Si se ha aceptado el vecino es porque es mejor que la mejor_secuencia que había
                registrada, por lo que se actualiza

                mejorFO = FO2;
                Array.Copy(vecina,mejor_secuencia, vecina.Length);
            }
        }

        // FASE ABEJA EXPLORADORA

        // Para cada individuo de la población que no ha sido mejorado en los ensayos anteriores,

        for (int i = 0; i < n; i++)
        {
```

```
                        if (mejora[i] == false)
                        {
                                // se abandona y se genera un nuevo individuo aleatorio en su posición

                                for (int j = 0; j < m; j++)
                                {
                                        Random aleatorio2 = new Random();
                                        individuo[1, j] = aleatorio2.Next(1, 50);
                                }

                                individuosalida = int_Quicksort_decre (individuo, 0, m - 1);
                                rellenar_vector(individuosalida, vecina, 0);

                                // y se incorpora a la población

                                rellenar_fila_matriz(poblacion, vecina, i);

                                // Se comprueba si es mejor que la mejor_secuencia que estaba recogida y si es
                                así, se actualiza

                                FO2 = Sum_tiempo_acceso_ponderado(dia_disponible, fecha_limite,
                                duracion_operacion, prioridad, cirujano_asociado, numero_quirofanos,
                                disponibilidadquirofano_turno, horasmaximaquirofano_turno, numero_cirujanos,
                                horascirujano_turno, dias, vecina);

                                if (FO2 < mejorFO)
                                {
                                        Array.Copy(vecina, mejor_secuencia, secuencia.Length);
                                        mejorFO = FO2;
                                        mejor_posicion = i;
                                }
                        }
                }

                // Actualización del tiempo

                tiempoActualMilisegundos = timer.ElapsedMilliseconds;
                tiempoActual = tiempoActualMilisegundos / 1000;
        }

        // IMPRESIÓN EN EXCELL

        FicheroCSV1.Write(alpha + ";" + beta + ";" + mds + ";" + numero_cirujanos + ";" + numero_quirofanos +
        ";" + instancia + ";" + mejorFO + "\n");
}

public static void mejora1(int n, int ensayos, int replicas, int numero_vecinos int dias, double alpha, double beta, int
mds, int instancia, int numero_quirofanos, int[,] disponibilidadquirofano_turno, double[] duracion_operacion,  int[]
dia_disponible, int[] fecha_limite, double[] prioridad, int[] cirujano_asociado, double[,] horascirujano_turno, int m,
int numero_cirujanos, double[,] horasmaxima_turno, StreamWriter FicheroCSV1)
    {
        //VARIABLES NECESARIAS PARA LA APLICACIÓN DEL ALGORITMO

        // Matriz que recoge todos los individuos que componen la población

        int[,] poblacion = new int[n, m];

        // Vector que guarda la secuencia que se va a incorporar en la población y su FO

        int[] secuencia = new int[m];
        double FO1 = 0;
```

```
// Vector que guarda la secuencia vecina que se genera en el algoritmo y su FO

int[] vecina = new int[m];
double FO2 = 0;

// Variable que recogerá la mejor solución encontrada

double mejorFO = 999;//Inicializada a un valor alto para que la mejor secuencia sea el primer individuo que se incorpora a la población (la primera secuencia que se comprueba) y sirva esta de comparativa con el resto.

// Matriz necesaria para generar una secuencia en función de un orden de datos enteros

int[,] individuo = new int[2, m];
int[,] individuosalida = new int[2, m];

// Matriz necesaria para generar una secuencia en función de un orden de datos decimales

double[,] individuo2 = new double[2, m]
double[,] individuosalida2 = new double[2, m];

        //GENERACIÓN DE LA POBLACIÓN

// PRIMER INDIVIDUO: Heurística NEH

// Se ordenan los trabajos según la regla LPT para dar lugar a la secuencia J

for (int j = 0; j < m; j++)
{
        Individuo2[0, j] = j;
        Individuo2[1, j] = duracion_operacion[j];
}

Individuosalida2 = Quicksort_decre(individuo, 0, m - 1);

for (int j = 0; j < m; j++)
{
        individuosalida[0, j] = Convert.ToInt32(individuosalida2[0, j]);
        secuencia[j] = individuosalida[0, j];
}


// En la primera iteración, se inserta en mejor_permutacion los trabajos 1 y 2 de la secuencia J con la ordenación que mejor FO presente. En caso de empate se seleccionará la última que se ha construido.

int[] mejor_permutacion = new int[2];

// Vector que recoge las distintas ordenaciones

int[] vector_aux = new int[2];

// Se comienza probando la primera ordenación (1-2)

vector_aux[0] = secuencia[0];
vector_aux[1] = secuencia[1];

FO1 = Sum_tiempo_acceso_ponderado(dia_disponible, fecha_limite, duracion_operacion, prioridad, cirujano_asociado, numero_quirofanos, disponibilidadquirofano_turno, horasmaximaquirofano_turno, numero_cirujanos, horascirujano_turno, dias, vector_aux);

// Se considera que 1 - 2 es la mejor permutación

Array.Copy(vector_aux, mejor_permutacion, 2);
```

```
// Se prueba la segunda ordenación (2-1)

vector_aux[0] = secuencia[1];
vector_aux[1] = secuencia[0];

FO2 = Sum_tiempo_acceso_ponderado(dia_disponible, fecha_limite, duracion_operacion, prioridad,
cirujano_asociado, numero_quirofanos, disponibilidadquirofano_turno, horasmaximaquirofano_turno,
numero_cirujanos, horascirujano_turno, dias, vector_aux);

// Se comprueba si finalmente 1-2 es la mejor ordenación y si no lo es, se actualiza como mejor_permutación
la 2-1

if (FO2 <= FO1)
{
        Array.Copy(vector_aux, mejor_permutacion, 2);
}

// Vector que recoge la mejor permutación de una iteración para usarla en la siguiente

int[] vector_aux2 = new int[2];
Array.Copy(mejor_permutacion, vector_aux2, 2);

// Bucle para insertar el resto de los trabajos de la secuencia J (es decir, realiza las siguientes iteraciones)

for (int i = 2; i < secuencia.Length; i++)
{
        FO1 = 999; // Inicializado a un valor alto para que la mejor FO sea la de la primera permutación y
        sirva como comparativa con las siguientes

        // Cada vez que se introduce un nuevo trabajo de J a se aumenta en 1 el tamaño de todos los vectores
        auxiliares

        Array.Resize(ref vector_aux, vector_aux.Length + 1);
        Array.Resize(ref vector_aux2, vector_aux2.Length + 1);
        Array.Resize(ref mejor_permutacion, mejor_permutacion.Length + 1);

        // El trabajo de J a se introduce en la posición que de lugar a la mejor FO

        for (int j = 0; j <= i; j++)
        {
                vector_aux[j] = secuencia[i];

                for (int k = j + 1; k <= i; k++)
                {

                        vector_aux[k] = vector_aux2[k - 1];
                }

                for (int k = 0; k < j; k++)
                {
                        vector_aux[k] = vector_aux2[k];
                }

                FO2 = Sum_tiempo_acceso_ponderado(dia_disponible, fecha_limite, duracion_operacion,
                prioridad, cirujano_asociado, numero_quirofanos, disponibilidadquirofano_turno,
                horasmaximaquirofano_turno, numero_cirujanos, horascirujano_turno, dias, vector_aux);

                if (FO2 <= FO1)
                {
                        FO1 = FO2;
                        Array.Copy(vector_aux, mejor_permutacion, vector_aux.Length);
```

```
            }
        }

        Array.Copy(mejor_permutacion, vector_aux2, mejor_permutacion.Length);
}

// Una vez que se obtiene la mejor_permutación final (mejor secuencia que se ha obtenido del NEH), se copia en secuencia para incorporarla en la población

mejorFO = FO1;
Array.Copy(mejor_permutacion, secuencia, m);
rellenar_fila_matriz(poblacion, secuencia, 0);

//SEGUNDO INDIVIDUO: Regla de despacho EDD (El trabajo con menor fecha de entrega se procesa primero)

for (int j = 0; j < m; j++)
{
        individuo[0, j] = j;
        individuo[1, j] = fecha_limite[j];
}

individuosalida = int_Quicksort (individuo, 0, m - 1);
rellenar_vector(individuosalida, secuencia, 0);
rellenar_fila_matriz(poblacion, secuencia, 1);

FO1 = Sum_tiempo_acceso_ponderado(dia_disponible, fecha_limite, duracion_operacion, prioridad, cirujano_asociado, numero_quirofanos, disponibilidadquirofano_turno, horasmaximaquirofano_turno, numero_cirujanos, horascirujano_turno, dias, secuencia);

if (FO1 < mejorFO)
{
        mejorFO = FO1;
}

// TERCER INDIVIDUO: Regla de despacho LPT (El trabajo con mayor tiempo de proceso se procesa primero)

for (int j = 0; j < m; j++)
{
        Individuo2[0, j] = j;
        Individuo2[1, j] = duracion_operacion[j];
}

Individuosalida2 = Quicksort_decre(individuo, 0, m - 1);

for (int j = 0; j < m; j++)
{
        individuosalida[0, j] = Convert.ToInt32(individuosalida2[0, j]);
        secuencia[j] = individuosalida[0, j];
}

rellenar_fila_matriz(poblacion, secuencia, 2);

FO1 = Sum_tiempo_acceso_ponderado(dia_disponible, fecha_limite, duracion_operacion, prioridad, cirujano_asociado, numero_quirofanos, disponibilidadquirofano_turno, horasmaximaquirofano_turno, numero_cirujanos, horascirujano_turno, dias, secuencia);

if (FO1 < mejorFO)
{
        mejorFO = FO1;
}
```

```
// CUARTO INDIVIDUO: Se va a generar ordenando los pacientes de mayor a menor prioridad

for (int j = 0; j < m; j++)
{
        Individuo2[0, j] = j;
        Individuo2[1, j] = prioridad [j];
}

Individuosalida2 = Quicksort_decre(individuo, 0, m - 1);

for (int j = 0; j < m; j++)
{
        individuosalida[0, j] = Convert.ToInt32(individuosalida2[0, j]);
        secuencia[j] = individuosalida[0, j];
}
rellenar_fila_matriz(poblacion, secuencia, 3);

FO1 = Sum_tiempo_acceso_ponderado(dia_disponible, fecha_limite, duracion_operacion, prioridad,
cirujano_asociado, numero_quirofanos, disponibilidadquirofano_turno, horasmaximaquirofano_turno,
numero_cirujanos, horascirujano_turno, dias, secuencia);

if (FO1 < mejorFO)
{
        mejorFO = FO1;
}

// INDIVIDUOS RESTANTES: hasta completar la población se generan de manera aleatoria.

int contador = 4;
while (contador < n)
{
        // Se asocian unos tiempos de proceso aleatorios a cada paciente

        for (int j = 0; j < m; j++)
        {
                individuo[0, j] = j;
                Random aleatorio = new Random();
                individuo[1, j] = aleatorio.Next(1, 50);
        }

        // Se obtiene una secuencia ordenada de pacientes en función de estos tiempos aleatorios que será
        una secuencia aleatoria

        individuosalida = int_Quicksort_decre(individuo, 0, m - 1);
        rellenar_vector(individuosalida, secuencia, 0);

        // Se introduce la secuencia en la población siempre que sea distinta a las que ya existían

        bool apto = no_repeticion(poblacion, secuencia);

        if (apto == true)
        {
                rellenar_fila_matriz(poblacion, secuencia, i);
                contador++;
                FO1 = Sum_tiempo_acceso_ponderado(dia_disponible, fecha_limite, duracion_operacion,
                prioridad, cirujano_asociado, numero_quirofanos, disponibilidadquirofano_turno,
                horasmaximaquirofano_turno, numero_cirujanos, horascirujano_turno, dias, secuencia);

                if (FO1 < mejorFO)
                {
                        mejorFO = FO1;
```

```
                    }

            }
}

// RESCATE DEL TIEMPO ACTUAL

var timer = Stopwatch.StartNew();
double tiempoActualMilisegundos = timer.ElapsedMilliseconds;
double tiempoActual = tiempoActualMilisegundos / 1000;


// APLICACIÓN DE LAS 3 FASES DEL ALGORITMO (CICLO) HASTA SUPERAR EL TIEMPO
LÍMITE

int tiempoLimite = 10 * numero_quirofanos * numero_cirujanos;
while (tiempoActual < tiempoLimite)
{
        // Vector que recogerá si un individuo ha mejorado o no en un ciclo

        bool[] mejora = new bool[n];

        for (int y = 0; y < ensayos; y++)
        {
                // FASE DE LA ABEJA OBRERA

                // En este caso, la fase de la abeja obrera se replicará varias veces

                for (int x = 0; x < replicas; x++)
                {
                        // Para cada individuo de la población,

                        for (int i = 0; i < n; i++)
                        {
                                rellenar_vector(poblacion, secuencia, i);
                                rellenar_vector(poblacion, vecina, i);

                                // Se genera un vecino aplicando el intercambio adyacente multiple

                                intercambio_multiple(vecina);

                                // Se acepta el vecino solo si es mejor

                                FO1 = Sum_tiempo_acceso_ponderado(dia_disponible, fecha_limite,
                                duracion_operacion, prioridad, cirujano_asociado, numero_quirofanos,
                                disponibilidadquirofano_turno,horasmaximaquirofano_turno,
                                numero_cirujanos, horascirujano_turno, dias, secuencia);

                                FO2 = Sum_tiempo_acceso_ponderado(dia_disponible, fecha_limite,
                                duracion_operacion, prioridad, cirujano_asociado, numero_quirofanos,
                                disponibilidadquirofano_turno,horasmaximaquirofano_turno,
                                numero_cirujanos, horascirujano_turno, dias, vecina);

                                if (FO2 < FO1)
                                {
                                        mejora[i] = true;
                                        rellenar_fila_matriz(poblacion, vecina, i);

                                        // Si el vecino que se incopora es mejor que la mejor_secuencia,
                                        se actualiza

                                        if (FO2 < mejorFO)
```

```
                    {
                              mejorFO = FO2;
                    }
          }
     }
}

// FASE DE LA ABEJA OBSERVADORA

// Se escogen aleatoriamente dos individuos de la población

Random aleatorio3 = new Random();
int aux1 = aleatorio3.Next(0, n);
int aux2 = aleatorio3.Next(0, n);

// Se usa "vecina" y "secuencia" para no declarar dos nuevos vectores que recojan los dos individuos

rellenar_vector(poblacion, secuencia, aux1);
rellenar_vector(poblacion, vecina, aux2);

// Se selecciona el mejor de los dos

FO1 = Sum_tiempo_acceso_ponderado (dia_disponible, fecha_limite, duracion_operacion, prioridad, cirujano_asociado, numero_quirofanos, disponibilidadquirofano_turno, horasmaximaquirofano_turno, numero_cirujanos, horascirujano_turno, dias, secuencia);

FO2 = Sum_tiempo_acceso_ponderado (dia_disponible, fecha_limite, duracion_operacion, prioridad, cirujano_asociado, numero_quirofanos, disponibilidadquirofano_turno, horasmaximaquirofano_turno, numero_cirujanos, horascirujano_turno, dias, vecina);

if (FO2 <= FO1)
{
          Array.Copy(vecina, secuencia, m);
}

else
{
          Array.Copy(secuencia, vecina, m);
}

// Se genera un vecino del mejor individuo seleccionado mediante intercambio adyacente múltiple

intercambio_multiple(vecina);

FO2 = Sum_tiempo_acceso_ponderado (dia_disponible, fecha_limite, duracion_operacion, prioridad, cirujano_asociado, numero_quirofanos, disponibilidadquirofano_turno, horasmaximaquirofano_turno, numero_cirujanos, horascirujano_turno, dias, vecina);

// Se comienza suponiendo que el vecino va a ser peor que todos los individuos de la población. Es decir, no existe ningun individuo de la población (ninguna fila) para la cual la FO es peor que la FO vecina, y en principio no se aceptaría

int peorfila = -1;

for (int i = 0; i < n; i++)
{
```

```
				rellenar_vector(poblacion, secuencia, i);

					FO1 = Sum_tiempo_acceso_ponderado (dia_disponible, fecha_limite,
					duracion_operacion, prioridad, cirujano_asociado, numero_quirofanos,
					disponibilidadquirofano_turno,horasmaximaquirofano_turno,
					numero_cirujanos, horascirujano_turno, dias, secuencia);

					if (FO1 > FO2)
					{
						FO2 = FO1;
						peorfila = i;
					}
				}

				// Si se demuestra que si que existe una fila en la cual la FO es peor que la vecina,

				if (peorfila >= 0)
				{
					// y además se comprueba que la vecina no se repite con ninguna otra ya
					existente en la población, se acepta

					bool apto = no_repeticion(poblacion, vecina);

					if (apto == true)
					{
						rellenar_fila_matriz(poblacion, vecina, peorfila);
						mejora[peorfila] = true;

						FO2 = Sum_tiempo_acceso_ponderado (dia_disponible,
						fecha_limite, duracion_operacion, prioridad, cirujano_asociado,
						numero_quirofanos,disponibilidadquirofano_turno,
						horasmaximaquirofano_turno,numero_cirujanos,
						horascirujano_turno, dias, vecina);

						// Si el vecino que se incopora es mejor que la mejor_secuencia,
						se actualiza
						if (FO2 < mejorFO)
						{
							mejorFO = FO2;
						}
					}
				}
			}

			// FASE ABEJA EXPLORADORA

			// Para cada individuo de la población que no se haya mejorado,

			for (int i = 0; i < n; i++)
			{
				if (mejora[i] == false)
				{
					rellenar_vector(poblacion, secuencia, i);

					FO1= Sum_tiempo_acceso_ponderado (dia_disponible, fecha_limite,
					duracion_operacion, prioridad, cirujano_asociado, numero_quirofanos,
					disponibilidadquirofano_turno,horasmaximaquirofano_turno,
					numero_cirujanos, horascirujano_turno, dias, secuencia);

					// se generan 3 vecinos mediante inserción. Debe hacerse una copia para
					que todos se generen a partir de la secuencia que no ha mejoradO
```

```
int[] copia = new int[m];

FO2= 999; // Para comenzar suponiendo que la mejor es la primera que se genera, se inicializa la variable con un valor muy alto.

for (int j = 0; j < numero_vecinos; j++)

{
        Array.Copy(secuencia, copia, secuencia.Length);

        insercion_aleatoria(copia);

        FO1 = Sum_tiempo_acceso_ponderado(dia_disponible, fecha_limite,duracion_operacion,prioridad, cirujano_asociado,numero_quirofanos, disponibilidadquirofano_turno, horasmaximaquirofano_turno, numero_cirujanos, horascirujano_turno, dias, copia);

        // Por cada vecino que se genera, se comprueba si es mejor que los que se habían generado previamente para quedarse con el mejor de todos

        if (FO1 < FO2)
        {
                FO2 = FO1;
                Array.Copy(copia, vecina, m);
        }

}

// Si el mejor vecino que se incopora es mejor que la mejor_secuencia, se actualiza

if (FO2 < mejorFO)
{
        mejorFO = FO2;
}

rellenar_fila_matriz(poblacion, vecina, i);
        }
    }

    // ACTUALIZACIÓN DEL TIEMPO

    tiempoActualMilisegundos = timer.ElapsedMilliseconds;
    tiempoActual = tiempoActualMilisegundos / 1000;
}

// IMPRESIÓN EN EXCELL

FicheroCSV1.Write(alpha + ";" + beta + ";" + mds + ";" + numero_cirujanos + ";" + numero_quirofanos + ";" + instancia + ";" + mejorFO + "\n");

}


public static void mejora2(int n, int ensayos, int replicas, int numero_vecinos int dias, double alpha, double beta, int mds, int instancia, int numero_quirofanos, int[,] disponibilidadquirofano_turno, double[] duracion_operacion, int[] dia_disponible, int[] fecha_limite, double[] prioridad, int[] cirujano_asociado, double[,] horascirujano_turno, int m, int numero_cirujanos, double[,] horasmaxima_turno, StreamWriter FicheroCSV1)
{
```

```
//VARIABLES NECESARIAS PARA LA APLICACIÓN DEL ALGORITMO

// Matriz que recoge todos los individuos que componen la población

int[,] poblacion = new int[n, m];

// Vector que guarda la secuencia que se va a incorporar en la población y su FO

int[] secuencia = new int[m];
double FO1 = 0;

// Vector que guarda la secuencia vecina que se genera en el algoritmo y su FO

int[] vecina = new int[m];
double FO2 = 0;

// Variable que recogerá la mejor solución encontrada

double mejorFO = 999;//Inicializada a un valor alto para que la mejor secuencia sea el primer individuo que se incorpora a la población (la primera secuencia que se comprueba) y sirva esta de comparativa con el resto.

// Matriz necesaria para generar una secuencia en función de un orden de datos enteros

int[,] individuo = new int[2, m];
int[,] individuosalida = new int[2, m];

// Matriz necesaria para generar una secuencia en función de un orden de datos decimales

double[,] individuo2 = new double[2, m]
double[,] individuosalida2 = new double[2, m];

//GENERACIÓN DE LA POBLACIÓN

// PRIMER INDIVIDUO: Heurística NEH

// Se ordenan los trabajos según la regla LPT para dar lugar a la secuencia J

for (int j = 0; j < m; j++)
{
        Individuo2[0, j] = j;
        Individuo2[1, j] = duracion_operacion[j];
}

Individuosalida2 = Quicksort_decre(individuo, 0, m - 1);

for (int j = 0; j < m; j++)
{
        individuosalida[0, j] = Convert.ToInt32(individuosalida2[0, j]);
        secuencia[j] = individuosalida[0, j];
}


// En la primera iteración, se inserta en mejor_permutacion los trabajos 1 y 2 de la secuencia J con la ordenación que mejor FO presente. En caso de empate se seleccionará la última que se ha construido.

int[] mejor_permutacion = new int[2];

// Vector que recoge las distintas ordenaciones

int[] vector_aux = new int[2];

// Se comienza probando la primera ordenación (1-2)
```

```
vector_aux[0] = secuencia[0];
vector_aux[1] = secuencia[1];

FO1 = Sum_tiempo_acceso_ponderado(dia_disponible, fecha_limite, duracion_operacion, prioridad, cirujano_asociado, numero_quirofanos, disponibilidadquirofano_turno, horasmaximaquirofano_turno, numero_cirujanos, horascirujano_turno, dias, vector_aux);

// Se considera que 1 - 2 es la mejor permutación

Array.Copy(vector_aux, mejor_permutacion, 2);

// Se prueba la segunda ordenación (2-1)

vector_aux[0] = secuencia[1];
vector_aux[1] = secuencia[0];

FO2 = Sum_tiempo_acceso_ponderado(dia_disponible, fecha_limite, duracion_operacion, prioridad, cirujano_asociado, numero_quirofanos, disponibilidadquirofano_turno, horasmaximaquirofano_turno, numero_cirujanos, horascirujano_turno, dias, vector_aux);

// Se comprueba si finalmente 1-2 es la mejor ordenación y si no lo es, se actualiza como mejor_permutación la 2-1

if (FO2 <= FO1)
{
	Array.Copy(vector_aux, mejor_permutacion, 2);
}

// Vector que recoge la mejor permutación de una iteración para usarla en la siguiente

int[] vector_aux2 = new int[2];
Array.Copy(mejor_permutacion, vector_aux2, 2);

// Bucle para insertar el resto de los trabajos de la secuencia J (es decir, realiza las siguientes iteraciones)

for (int i = 2; i < secuencia.Length; i++)
{
	FO1 = 999; // Inicializado a un valor alto para que la mejor FO sea la de la primera permutación y sirva como comparativa con las siguientes

	// Cada vez que se introduce un nuevo trabajo de J a se aumenta en 1 el tamaño de todos los vectores auxiliares

	Array.Resize(ref vector_aux, vector_aux.Length + 1);
	Array.Resize(ref vector_aux2, vector_aux2.Length + 1);
	Array.Resize(ref mejor_permutacion, mejor_permutacion.Length + 1);

	// El trabajo de J a se introduce en la posición que de lugar a la mejor FO

	for (int j = 0; j <= i; j++)
	{
		vector_aux[j] = secuencia[i];

		for (int k = j + 1; k <= i; k++)
		{

			vector_aux[k] = vector_aux2[k - 1];
		}

		for (int k = 0; k < j; k++)
		{
```

```
                        vector_aux[k] = vector_aux2[k];
                }

                FO2 = Sum_tiempo_acceso_ponderado (dia_disponible, fecha_limite,
                duracion_operacion, prioridad, cirujano_asociado, numero_quirofanos,
                disponibilidadquirofano_turno, horasmaximaquirofano_turno, numero_cirujanos,
                horascirujano_turno, dias, vector_aux);

                if (FO2 <= FO1)
                {
                        FO1 = FO2;
                        Array.Copy(vector_aux, mejor_permutacion, vector_aux.Length);
                }
        }

        Array.Copy(mejor_permutacion, vector_aux2, mejor_permutacion.Length);
}

// Una vez que se obtiene la mejor_permutación final (mejor secuencia que se ha obtenido del NEH), se copia
en secuencia para incorporarla en la población

mejorFO = FO1;
Array.Copy(mejor_permutacion, secuencia, m);
rellenar_fila_matriz(poblacion, secuencia, 0);

//SEGUNDO INDIVIDUO: Regla de despacho EDD (El trabajo con menor fecha de entrega se procesa
primero)

for (int j = 0; j < m; j++)
{
        individuo[0, j] = j;
        individuo[1, j] = fecha_limite[j];
}

individuosalida = int_Quicksort (individuo, 0, m - 1);
rellenar_vector(individuosalida, secuencia, 0);
rellenar_fila_matriz(poblacion, secuencia, 1);

FO1 = Sum_tiempo_acceso_ponderado (dia_disponible, fecha_limite, duracion_operacion, prioridad,
cirujano_asociado, numero_quirofanos, disponibilidadquirofano_turno, horasmaximaquirofano_turno,
numero_cirujanos, horascirujano_turno, dias, secuencia);

if (FO1 < mejorFO)
{
        mejorFO = FO1;
}

// TERCER INDIVIDUO: Regla de despacho LPT (El trabajo con mayor tiempo de proceso se procesa
primero)

for (int j = 0; j < m; j++)
{
        Individuo2[0, j] = j;
        Individuo2[1, j] = duracion_operacion[j];
}

Individuosalida2 = Quicksort_decre(individuo, 0, m - 1);

for (int j = 0; j < m; j++)
{
        individuosalida[0, j] = Convert.ToInt32(individuosalida2[0, j]);
        secuencia[j] = individuosalida[0, j];
```

```
}

rellenar_fila_matriz(poblacion, secuencia, 2);

FO1 = Sum_tiempo_acceso_ponderado(dia_disponible, fecha_limite, duracion_operacion, prioridad, cirujano_asociado, numero_quirofanos, disponibilidadquirofano_turno, horasmaximaquirofano_turno, numero_cirujanos, horascirujano_turno, dias, secuencia);

if (FO1 < mejorFO)
{
        mejorFO = FO1;
}

// CUARTO INDIVIDUO: Se va a generar ordenando los pacientes de mayor a menor prioridad

for (int j = 0; j < m; j++)
{
        Individuo2[0, j] = j;
        Individuo2[1, j] = prioridad [j];
}

Individuosalida2 = Quicksort_decre(individuo, 0, m - 1);

for (int j = 0; j < m; j++)
{
        individuosalida[0, j] = Convert.ToInt32(individuosalida2[0, j]);
        secuencia[j] = individuosalida[0, j];
}
rellenar_fila_matriz(poblacion, secuencia, 3);

FO1 = Sum_tiempo_acceso_ponderado(dia_disponible, fecha_limite, duracion_operacion, prioridad, cirujano_asociado, numero_quirofanos, disponibilidadquirofano_turno, horasmaximaquirofano_turno, numero_cirujanos, horascirujano_turno, dias, secuencia);

if (FO1 < mejorFO)
{
        mejorFO = FO1;
}

// INDIVIDUOS RESTANTES: hasta completar la población se generan de manera aleatoria.

int contador = 4;
while (contador < n)
{
        // Se asocian unos tiempos de proceso aleatorios a cada paciente

        for (int j = 0; j < m; j++)
        {
                individuo[0, j] = j;
                Random aleatorio = new Random();
                individuo[1, j] = aleatorio.Next(1, 50);
        }

        // Se obtiene una secuencia ordenada de pacientes en función de estos tiempos aleatorios que será una secuencia aleatoria

        individuosalida = int_Quicksort_decre(individuo, 0, m - 1);
        rellenar_vector(individuosalida, secuencia, 0);

        // Se introduce la secuencia en la población siempre que sea distinta a las que ya existían

        bool apto = no_repeticion(poblacion, secuencia);
```

```
if (apto == true)
{
        rellenar_fila_matriz(poblacion, secuencia, i);
        contador++;
        FO1 = Sum_tiempo_acceso_ponderado (dia_disponible, fecha_limite,
        duracion_operacion, prioridad, cirujano_asociado, numero_quirofanos,
        disponibilidadquirofano_turno, horasmaximaquirofano_turno, numero_cirujanos,
        horascirujano_turno, dias, secuencia);

        if (FO1 < mejorFO)
        {
                mejorFO = FO1;
        }

        }
}

// RESCATE DEL TIEMPO ACTUAL

var timer = Stopwatch.StartNew();
double tiempoActualMilisegundos = timer.ElapsedMilliseconds;
double tiempoActual = tiempoActualMilisegundos / 1000;


// APLICACIÓN DE LAS 3 FASES DEL ALGORITMO (CICLO) HASTA SUPERAR EL
TIEMPO LÍMITE

int tiempoLimite = 10 * numero_quirofanos * numero_cirujanos;
while (tiempoActual < tiempoLimite)
{
        // Vector que recogerá si un individuo ha mejorado o no en un ciclo

        bool[] mejora = new bool[n];

        for (int y = 0; y < ensayos; y++)
        {
                // FASE DE LA ABEJA OBRERA

                // En este caso, la fase de la abeja obrera se replicará varias veces

                for (int x = 0; x < replicas; x++)
                {
                        // Para cada individuo de la población,

                        for (int i = 0; i < n; i++)
                        {
                                rellenar_vector(poblacion, secuencia, i);
                                rellenar_vector(poblacion, vecina, i);

                                // Se genera un vecino aplicando el intercambio adyacente
                                multiple

                                intercambio_multiple(vecina);

                                // Se acepta el vecino solo si es mejor

                                FO1 = Sum_tiempo_acceso_ponderado(dia_disponible,
                                fecha_limite, duracion_operacion, prioridad, cirujano_asociado,
                                numero_quirofanos,disponibilidadquirofano_turno,
                                horasmaximaquirofano_turno,numero_cirujanos,
                                horascirujano_turno, dias, secuencia);
```

```
FO2 = Sum_tiempo_acceso_ponderado(dia_disponible, fecha_limite, duracion_operacion, prioridad, cirujano_asociado, numero_quirofanos,disponibilidadquirofano_turno, horasmaximaquirofano_turno,numero_cirujanos, horascirujano_turno, dias, vecina);

if (FO2 < FO1)
{
    mejora[i] = true;
    rellenar_fila_matriz(poblacion, vecina, i);

    // Si el vecino que se incopora es mejor que la mejor_secuencia, se actualiza

    if (FO2 < mejorFO)
    {
        mejorFO = FO2;
    }
}
}
}

// FASE DE LA ABEJA OBSERVADORA

// Se escogen aleatoriamente dos individuos de la población

Random aleatorio3 = new Random();
int aux1 = aleatorio3.Next(0, n);
int aux2 = aleatorio3.Next(0, n);

// Se usa "vecina" y "secuencia" para no declarar dos nuevos vectores que recojan los dos individuos

rellenar_vector(poblacion, secuencia, aux1);
rellenar_vector(poblacion, vecina, aux2);

// Se selecciona el mejor de los dos

FO1 = Sum_tiempo_acceso_ponderado (dia_disponible, fecha_limite, duracion_operacion, prioridad, cirujano_asociado, numero_quirofanos, disponibilidadquirofano_turno, horasmaximaquirofano_turno, numero_cirujanos, horascirujano_turno, dias, secuencia);

FO2 = Sum_tiempo_acceso_ponderado (dia_disponible, fecha_limite, duracion_operacion, prioridad, cirujano_asociado, numero_quirofanos, disponibilidadquirofano_turno, horasmaximaquirofano_turno, numero_cirujanos, horascirujano_turno, dias, vecina);

if (FO2 <= FO1)
{
    Array.Copy(vecina, secuencia, m);
}

else
{
    Array.Copy(secuencia, vecina, m);
}

// Se genera un vecino del mejor individuo seleccionado mediante intercambio adyacente múltiple
```

```
			intercambio_multiple(vecina);

			FO2 = Sum_tiempo_acceso_ponderado (dia_disponible, fecha_limite, duracion_operacion, prioridad, cirujano_asociado, numero_quirofanos, disponibilidadquirofano_turno, horasmaximaquirofano_turno, numero_cirujanos, horascirujano_turno, dias, vecina);

			// Se comienza suponiendo que el vecino va a ser peor que todos los individuos de la población. Es decir, no existe ningun individuo de la población (ninguna fila) para la cual la FO es peor que la FO vecina, y en principio no se aceptaría

			int peorfila = -1;

			for (int i = 0; i < n; i++)
			{

				rellenar_vector(poblacion, secuencia, i);

				FO1 = Sum_tiempo_acceso_ponderado (dia_disponible, fecha_limite, duracion_operacion, prioridad, cirujano_asociado, numero_quirofanos, disponibilidadquirofano_turno,horasmaximaquirofano_turno, numero_cirujanos, horascirujano_turno, dias, secuencia);

				if (FO1 > FO2)
				{
					FO2 = FO1;
					peorfila = i;
				}
			}

			// Si se demuestra que si que existe una fila en la cual la FO es peor que la vecina,

			if (peorfila >= 0)
			{
				// y además se comprueba que la vecina no se repite con ninguna otra ya existente en la población, se acepta

				bool apto = no_repeticion(poblacion, vecina);

				if (apto == true)
				{
					rellenar_fila_matriz(poblacion, vecina, peorfila);
					mejora[peorfila] = true;

					FO2 = Sum_tiempo_acceso_ponderado (dia_disponible, fecha_limite, duracion_operacion, prioridad, cirujano_asociado, numero_quirofanos,disponibilidadquirofano_turno, horasmaximaquirofano_turno,numero_cirujanos, horascirujano_turno, dias, vecina);

					// Si el vecino que se incopora es mejor que la mejor_secuencia, se actualiza
					if (FO2 < mejorFO)
					{
						mejorFO = FO2;
					}
				}
			}
		}

		// FASE ABEJA EXPLORADORA
```

```
// Para cada individuo de la población que no se haya mejorado,

for (int i = 0; i < n; i++)
{
	if (mejora[i] == false)
	{
		rellenar_vector(poblacion, secuencia, i);

		FO1= Sum_tiempo_acceso_ponderado(dia_disponible, fecha_limite,
		duracion_operacion, prioridad, cirujano_asociado, numero_quirofanos,
		disponibilidadquirofano_turno,horasmaximaquirofano_turno,
		numero_cirujanos, horascirujano_turno, dias, secuencia);

		// se generan 3 vecinos mediante inserción. Debe hacerse una copia para
		que todos se generen a partir de la secuencia que no ha mejorado

		int[] copia = new int[m];

		double mejorFOvecina= 999; // Para comenzar suponiendo que la mejor
		es la primera que se genera, se inicializa la variable con un valor muy
		alto.

		for (int j = 0; j < numero_vecinos; j++)

		{
			Array.Copy(secuencia, copia, secuencia.Length);

			Insercion_aleatoria(copia);

			FO2 = Sum_tiempo_acceso_ponderado(dia_disponible,
			fecha_limite, duracion_operacion, prioridad, cirujano_asociado,
			numero_quirofanos,disponibilidadquirofano_turno,
			horasmaximaquirofano_turno,numero_cirujanos,
			horascirujano_turno, dias, copia);

			// Por cada vecino que se genera, se comprueba si es mejor que
			los que se habían generado previamente para quedarse con el
			mejor de todos que será el que se introduzca en la población

			if (FO2 < mejorFOvecina)
			{
				mejorFOvecina = FO2;
				Array.Copy(copia, vecina, m);
			}
		}

		// Si el mejor vecino es mejor que la secuencia a partir de la que se obtiene
		o un 2% peor
		if (mejorFOvecina <= (1 + (3 / 100)) * FO1)
		{

			bool apto = no_repeticion(poblacion, vecina);

			// y además se comprueba que no se repite
			if (apto == true)
			{
				// se incorpora a la población

				rellenar_fila_matriz(poblacion, vecina, i);

				// Si el vecino que se incopora es mejor que la
				mejor_secuencia, se actualiza
```

```
                                                  if (mejorFOvecina < mejorFO)
                                                  {
                                                        mejorFO = mejorFOvecina;
                                                  }
                                            }
                                      }
                                }
                          }

                          // ACTUALIZACIÓN DEL TIEMPO

                          tiempoActualMilisegundos = timer.ElapsedMilliseconds;
                          tiempoActual = tiempoActualMilisegundos / 1000;
                    }

                    // IMPRESIÓN EN EXCELL

                    FicheroCSV1.Write(alpha + ";" + beta + ";" + mds + ";" + numero_cirujanos + ";" +
                    numero_quirofanos + ";" + instancia + ";" + mejorFO + "\n");

      }

public static void mejora3(int n, int ensayos, int replicas, int numero_vecinos int dias, double alpha, double beta, int
mds, int instancia, int numero_quirofanos, int[,] disponibilidadquirofano_turno, double[] duracion_operacion, int[]
dia_disponible, int[] fecha_limite, double[] prioridad, int[] cirujano_asociado, double[,] horascirujano_turno, int m,
int numero_cirujanos, double[,] horasmaxima_turno, StreamWriter FicheroCSV1)
      {
            //VARIABLES NECESARIAS PARA LA APLICACIÓN DEL ALGORITMO

            // Matriz que recoge todos los individuos que componen la población

            int[,] poblacion = new int[n, m];

            // Vector que guarda la secuencia que se va a incorporar en la población y su FO

            int[] secuencia = new int[m];
            double FO1 = 0;

            // Vector que guarda la secuencia vecina que se genera en el algoritmo y su FO

            int[] vecina = new int[m];
            double FO2 = 0;

            // Variable que recogerá la mejor solución encontrada

            double mejorFO = 999;//Inicializada a un valor alto para que la mejor secuencia sea el primer individuo que
            se incorpora a la población (la primera secuencia que se comprueba) y sirva esta de comparativa con el resto.

            // Matriz necesaria para generar una secuencia en función de un orden de datos enteros

            int[,] individuo = new int[2, m];
            int[,] individuosalida = new int[2, m];

            // Matriz necesaria para generar una secuencia en función de un orden de datos decimales

            double[,] individuo2 = new double[2, m]
            double[,] individuosalida2 = new double[2, m];

            //GENERACIÓN DE LA POBLACIÓN

            // PRIMER INDIVIDUO: Heurística NEH
```

```
// Se ordenan los trabajos según la regla LPT para dar lugar a la secuencia J

for (int j = 0; j < m; j++)
{
        Individuo2[0, j] = j;
        Individuo2[1, j] = duracion_operacion[j];
}

Individuosalida2 = Quicksort_decre(individuo, 0, m - 1);

for (int j = 0; j < m; j++)
{
        individuosalida[0, j] = Convert.ToInt32(individuosalida2[0, j]);
        secuencia[j] = individuosalida[0, j];
}


// En la primera iteración, se inserta en mejor_permutacion los trabajos 1 y 2 de la secuencia J con la
ordenación que mejor FO presente. En caso de empate se seleccionará la última que se ha construido.

int[] mejor_permutacion = new int[2];

// Vector que recoge las distintas ordenaciones

int[] vector_aux = new int[2];

// Se comienza probando la primera ordenación (1-2)

vector_aux[0] = secuencia[0];
vector_aux[1] = secuencia[1];

FO1 = Sum_tiempo_acceso_ponderado(dia_disponible, fecha_limite, duracion_operacion, prioridad,
cirujano_asociado, numero_quirofanos, disponibilidadquirofano_turno, horasmaximaquirofano_turno,
numero_cirujanos, horascirujano_turno, dias, vector_aux);

// Se considera que 1 - 2 es la mejor permutación

Array.Copy(vector_aux, mejor_permutacion, 2);

// Se prueba la segunda ordenación (2-1)

vector_aux[0] = secuencia[1];
vector_aux[1] = secuencia[0];

FO2 = Sum_tiempo_acceso_ponderado(dia_disponible, fecha_limite, duracion_operacion, prioridad,
cirujano_asociado, numero_quirofanos, disponibilidadquirofano_turno, horasmaximaquirofano_turno,
numero_cirujanos, horascirujano_turno, dias, vector_aux);

// Se comprueba si finalmente 1-2 es la mejor ordenación y si no lo es, se actualiza como mejor_permutación
la 2-1

if (FO2 <= FO1)
{
        Array.Copy(vector_aux, mejor_permutacion, 2);
}

// Vector que recoge la mejor permutación de una iteración para usarla en la siguiente

int[] vector_aux2 = new int[2];
Array.Copy(mejor_permutacion, vector_aux2, 2);
```

```
// Bucle para insertar el resto de los trabajos de la secuencia J (es decir, realiza las siguientes iteraciones)

for (int i = 2; i < secuencia.Length; i++)
{
        FO1 = 999; // Inicializado a un valor alto para que la mejor FO sea la de la primera permutación y
        sirva como comparativa con las siguientes

        // Cada vez que se introduce un nuevo trabajo de J a se aumenta en 1 el tamaño de todos los vectores
        auxiliares

        Array.Resize(ref vector_aux, vector_aux.Length + 1);
        Array.Resize(ref vector_aux2, vector_aux2.Length + 1);
        Array.Resize(ref mejor_permutacion, mejor_permutacion.Length + 1);

        // El trabajo de J a se introduce en la posición que de lugar a la mejor FO

        for (int j = 0; j <= i; j++)
        {
                vector_aux[j] = secuencia[i];

                for (int k = j + 1; k <= i; k++)
                {

                        vector_aux[k] = vector_aux2[k - 1];
                }

                for (int k = 0; k < j; k++)
                {
                        vector_aux[k] = vector_aux2[k];
                }

                FO2 = Sum_tiempo_acceso_ponderado (dia_disponible, fecha_limite,
                duracion_operacion, prioridad, cirujano_asociado, numero_quirofanos,
                disponibilidadquirofano_turno, horasmaximaquirofano_turno, numero_cirujanos,
                horascirujano_turno, dias, vector_aux);

                if (FO2 <= FO1)
                {
                        FO1 = FO2;
                        Array.Copy(vector_aux, mejor_permutacion, vector_aux.Length);
                }
        }

        Array.Copy(mejor_permutacion, vector_aux2, mejor_permutacion.Length);
}

// Una vez que se obtiene la mejor_permutación final (mejor secuencia que se ha obtenido del NEH), se copia
en secuencia para incorporarla en la población

mejorFO = FO1;
Array.Copy(mejor_permutacion, secuencia, m);
rellenar_fila_matriz(poblacion, secuencia, 0);

//SEGUNDO INDIVIDUO: Regla de despacho EDD (El trabajo con menor fecha de entrega se procesa
primero)

for (int j = 0; j < m; j++)
{
        individuo[0, j] = j;
        individuo[1, j] = fecha_limite[j];
}
```

```
individuosalida = int_Quicksort (individuo, 0, m - 1);
rellenar_vector(individuosalida, secuencia, 0);
rellenar_fila_matriz(poblacion, secuencia, 1);

FO1 = Sum_tiempo_acceso_ponderado (dia_disponible, fecha_limite, duracion_operacion, prioridad,
cirujano_asociado, numero_quirofanos, disponibilidadquirofano_turno, horasmaximaquirofano_turno,
numero_cirujanos, horascirujano_turno, dias, secuencia);

if (FO1 < mejorFO)
{
	mejorFO = FO1;
}

// TERCER INDIVIDUO: Regla de despacho LPT (El trabajo con mayor tiempo de proceso se procesa
primero)

for (int j = 0; j < m; j++)
{
	Individuo2[0, j] = j;
	Individuo2[1, j] = duracion_operacion[j];
}

Individuosalida2 = Quicksort_decre(individuo, 0, m - 1);

for (int j = 0; j < m; j++)
{
	individuosalida[0, j] = Convert.ToInt32(individuosalida2[0, j]);
	secuencia[j] = individuosalida[0, j];
}

rellenar_fila_matriz(poblacion, secuencia, 2);

FO1 = Sum_tiempo_acceso_ponderado(dia_disponible, fecha_limite, duracion_operacion, prioridad,
cirujano_asociado, numero_quirofanos, disponibilidadquirofano_turno, horasmaximaquirofano_turno,
numero_cirujanos, horascirujano_turno, dias, secuencia);

if (FO1 < mejorFO)
{
	mejorFO = FO1;
}

// CUARTO INDIVIDUO: Se va a generar ordenando los pacientes de mayor a menor prioridad

for (int j = 0; j < m; j++)
{
	Individuo2[0, j] = j;
	Individuo2[1, j] = prioridad [j];
}

Individuosalida2 = Quicksort_decre(individuo, 0, m - 1);

for (int j = 0; j < m; j++)
{
	individuosalida[0, j] = Convert.ToInt32(individuosalida2[0, j]);
	secuencia[j] = individuosalida[0, j];
}
rellenar_fila_matriz(poblacion, secuencia, 3);

FO1 = Sum_tiempo_acceso_ponderado(dia_disponible, fecha_limite, duracion_operacion, prioridad,
cirujano_asociado, numero_quirofanos, disponibilidadquirofano_turno, horasmaximaquirofano_turno,
numero_cirujanos, horascirujano_turno, dias, secuencia);
```

```
if (FO1 < mejorFO)
{
        mejorFO = FO1;
}

// INDIVIDUOS RESTANTES: hasta completar la población se generan de manera aleatoria.

int contador = 4;
while (contador < n)
{
        // Se asocian unos tiempos de proceso aleatorios a cada paciente

        for (int j = 0; j < m; j++)
        {
                individuo[0, j] = j;
                Random aleatorio = new Random();
                individuo[1, j] = aleatorio.Next(1, 50);
        }

        // Se obtiene una secuencia ordenada de pacientes en función de estos tiempos aleatorios que será
        una secuencia aleatoria

        individuosalida = int_Quicksort_decre(individuo, 0, m - 1);
        rellenar_vector(individuosalida, secuencia, 0);

        // Se introduce la secuencia en la población siempre que sea distinta a las que ya existían

        bool apto = no_repeticion(poblacion, secuencia);

        if (apto == true)
        {
                rellenar_fila_matriz(poblacion, secuencia, i);
                contador++;
                FO1 = Sum_tiempo_acceso_ponderado (dia_disponible, fecha_limite,
                duracion_operacion, prioridad, cirujano_asociado, numero_quirofanos,
                disponibilidadquirofano_turno, horasmaximaquirofano_turno, numero_cirujanos,
                horascirujano_turno, dias, secuencia);

                if (FO1 < mejorFO)
                {
                        mejorFO = FO1;
                }

        }
}

// RESCATE DEL TIEMPO ACTUAL

var timer = Stopwatch.StartNew();
double tiempoActualMilisegundos = timer.ElapsedMilliseconds;
double tiempoActual = tiempoActualMilisegundos / 1000;


// APLICACIÓN DE LAS 3 FASES DEL ALGORITMO (CICLO) HASTA SUPERAR EL TIEMPO
LÍMITE

int tiempoLimite = 10 * numero_quirofanos * numero_cirujanos;
while (tiempoActual < tiempoLimite)
{
        // Vector que recogerá si un individuo ha mejorado o no en un ciclo

        bool[] mejora = new bool[n];
```

```
for (int y = 0; y < ensayos; y++)
{

        // FASE DE LA ABEJA OBRERA

        // En este caso, la fase de la abeja obrera se replicará varias veces

        for (int x = 0; x < replicas; x++)
        {
                // Para cada individuo de la población,

                for (int i = 0; i < n; i++)
                {
                        rellenar_vector(poblacion, secuencia, i);
                        rellenar_vector(poblacion, vecina, i);

                        // Se genera un vecino aplicando el intercambio adyacente múltiple

                        Insercion_aleatoria (vecina);

                        // Se acepta el vecino solo si es mejor

                        FO1 = Sum_tiempo_acceso_ponderado(dia_disponible, fecha_limite,
                        duracion_operacion, prioridad, cirujano_asociado, numero_quirofanos,
                        disponibilidadquirofano_turno,horasmaximaquirofano_turno,
                        numero_cirujanos, horascirujano_turno, dias, secuencia);

                        FO2 = Sum_tiempo_acceso_ponderado(dia_disponible, fecha_limite,
                        duracion_operacion, prioridad, cirujano_asociado, numero_quirofanos,
                        disponibilidadquirofano_turno,horasmaximaquirofano_turno,
                        numero_cirujanos, horascirujano_turno, dias, vecina);

                        if (FO2 < FO1)
                        {
                                mejora[i] = true;
                                rellenar_fila_matriz(poblacion, vecina, i);

                                // Si el vecino que se incopora es mejor que la mejor_secuencia,
                                se actualiza

                                if (FO2 < mejorFO)
                                {
                                        mejorFO = FO2;
                                }
                        }
                }
        }


        // FASE DE LA ABEJA OBSERVADORA

        // Se escogen aleatoriamente dos individuos de la población

        Random aleatorio3 = new Random();
        int aux1 = aleatorio3.Next(0, n);
        int aux2 = aleatorio3.Next(0, n);

        // Se usa "vecina" y "secuencia" para no declarar dos nuevos vectores que recojan los dos
        individuos
```

```
rellenar_vector(poblacion, secuencia, aux1);
rellenar_vector(poblacion, vecina, aux2);

// Se selecciona el mejor de los dos

FO1 = Sum_tiempo_acceso_ponderado(dia_disponible, fecha_limite, duracion_operacion, prioridad, cirujano_asociado, numero_quirofanos, disponibilidadquirofano_turno, horasmaximaquirofano_turno, numero_cirujanos, horascirujano_turno, dias, secuencia);

FO2 = Sum_tiempo_acceso_ponderado(dia_disponible, fecha_limite, duracion_operacion, prioridad, cirujano_asociado, numero_quirofanos, disponibilidadquirofano_turno, horasmaximaquirofano_turno, numero_cirujanos, horascirujano_turno, dias, vecina);

if (FO2 <= FO1)
{
        Array.Copy(vecina, secuencia, m);
}

else
{
        Array.Copy(secuencia, vecina, m);
}

// Se genera un vecino del mejor individuo seleccionado mediante intercambio adyacente múltiple

Inserción_aleatoria (vecina);

FO2 = Sum_tiempo_acceso_ponderado(dia_disponible, fecha_limite, duracion_operacion, prioridad, cirujano_asociado, numero_quirofanos, disponibilidadquirofano_turno, horasmaximaquirofano_turno, numero_cirujanos, horascirujano_turno, dias, vecina);

// Se comienza suponiendo que el vecino va a ser peor que todos los individuos de la población. Es decir, no existe ningun individuo de la población (ninguna fila) para la cual la FO es peor que la FO vecina, y en principio no se aceptaría.

int peorfila = -1;

for (int i = 0; i < n; i++)
{
        rellenar_vector(poblacion, secuencia, i);

        FO1 = Sum_tiempo_acceso_ponderado(dia_disponible, fecha_limite, duracion_operacion, prioridad, cirujano_asociado, numero_quirofanos, disponibilidadquirofano_turno,horasmaximaquirofano_turno, numero_cirujanos, horascirujano_turno, dias, secuencia);

        if (FO1 > FO2)
        {
                FO2 = FO1;
                peorfila = i;
        }
}

// Si se demuestra que sí que existe una fila en la cual la FO es peor que la vecina,

if (peorfila >= 0)
{
```

```
                    // y además se comprueba que la vecina no se repite con ninguna otra ya existente en
                    la población, se acepta

                    bool apto = no_repeticion(poblacion, vecina);

                    if (apto == true)
                    {
                          rellenar_fila_matriz(poblacion, vecina, peorfila);
                          mejora[peorfila] = true;

                          FO2 = Sum_tiempo_acceso_ponderado(dia_disponible, fecha_limite,
                          duracion_operacion, prioridad, cirujano_asociado, numero_quirofanos,
                          disponibilidadquirofano_turno, horasmaximaquirofano_turno,
                          numero_cirujanos, horascirujano_turno, dias, vecina);

                          // Si el vecino que se incopora es mejor que la mejor_secuencia, se actualiza

                          if (FO2 < mejorFO)
                          {
                                mejorFO = FO2;
                          }
                    }
              }
        }

// FASE ABEJA EXPLORADORA

// Para cada individuo de la población que no se haya mejorado,

for (int i = 0; i < n; i++)
{
        if (mejora[i] == false)
        {
              rellenar_vector(poblacion, secuencia, i);

              FO1= Sum_tiempo_acceso_ponderado(dia_disponible, fecha_limite,
              duracion_operacion, prioridad, cirujano_asociado, numero_quirofanos,
              disponibilidadquirofano_turno, horasmaximaquirofano_turno,
              numero_cirujanos, horascirujano_turno, dias, secuencia);

              // se generan 3 vecinos mediante inserción. Debe hacerse una copia para que todos
              se generen a partir de la secuencia que no ha mejoradO

              int[] copia = new int[m];

              FO2= 999; // Para comenzar suponiendo que la mejor es la primera que se genera,
              se inicializa la variable con un valor muy alto.

              for (int j = 0; j < numero_vecinos; j++)

              {
                    Array.Copy(secuencia, copia, secuencia.Length);

                    insercion_aleatoria(copia);

                    FO1 = Sum_tiempo_acceso_ponderado(dia_disponible, fecha_limite,
                    duracion_operacion, prioridad, cirujano_asociado, numero_quirofanos,
                    disponibilidadquirofano_turno, horasmaximaquirofano_turno,
                    numero_cirujanos, horascirujano_turno, dias, copia);

                    // Por cada vecino que se genera, se comprueba si es mejor que los que
                    se habían generado previamente para quedarse con el mejor de todos
```

```
                        if (FO1 < FO2)
                        {
                            FO2 = FO1;
                            Array.Copy(copia, vecina, m);
                        }

                    }

                    // Si el mejor vecino que se incopora es mejor que la mejor_secuencia, se actualiza

                    if (FO2 < mejorFO)
                    {
                        mejorFO = FO2;
                    }

                    rellenar_fila_matriz(poblacion, vecina, i);
                }
            }

            // ACTUALIZACIÓN DEL TIEMPO

            tiempoActualMilisegundos = timer.ElapsedMilliseconds;
            tiempoActual = tiempoActualMilisegundos / 1000;
        }

        // IMPRESIÓN EN EXCELL

        FicheroCSV1.Write(alpha + ";" + beta + ";" + mds + ";" + numero_cirujanos + ";" + numero_quirofanos +
        ";" + instancia + ";" + mejorFO + "\n");
    }

public static void mejora4(int n, int ensayos, int replicas, int numero_vecinos int dias, double alpha, double beta, int
mds, int instancia, int numero_quirofanos, int[,] disponibilidadquirofano_turno, double[] duracion_operacion,  int[]
dia_disponible, int[] fecha_limite, double[] prioridad, int[] cirujano_asociado, double[,] horascirujano_turno, int m,
int numero_cirujanos, double[,] horasmaxima_turno, StreamWriter FicheroCSV1)
    {
        //VARIABLES NECESARIAS PARA LA APLICACIÓN DEL ALGORITMO

        // Matriz que recoge todos los individuos que componen la población

        int[,] poblacion = new int[n, m];

        // Vector que guarda la secuencia que se va a incorporar en la población y su FO

        int[] secuencia = new int[m];
        double FO1 = 0;

        // Vector que guarda la secuencia vecina que se genera en el algoritmo y su FO

        int[] vecina = new int[m];
        double FO2 = 0;

        // Variable que recogerá la mejor solución encontrada

        double mejorFO = 999;//Inicializada a un valor alto para que la mejor secuencia sea el primer individuo que
        se incorpora a la población (la primera secuencia que se comprueba) y sirva esta de comparativa con el resto.

        // Matriz necesaria para generar una secuencia en función de un orden de datos enteros

        int[,] individuo = new int[2, m];
        int[,] individuosalida = new int[2, m];
```

```
// Matriz necesaria para generar una secuencia en función de un orden de datos decimales

double[,] individuo2 = new double[2, m]
double[,] individuosalida2 = new double[2, m];

//GENERACIÓN DE LA POBLACIÓN

// PRIMER INDIVIDUO: Heurística NEH

// Se ordenan los trabajos según la regla LPT para dar lugar a la secuencia J

for (int j = 0; j < m; j++)
{
        Individuo2[0, j] = j;
        Individuo2[1, j] = duracion_operacion[j];
}

Individuosalida2 = Quicksort_decre(individuo, 0, m - 1);

for (int j = 0; j < m; j++)
{
        individuosalida[0, j] = Convert.ToInt32(individuosalida2[0, j]);
        secuencia[j] = individuosalida[0, j];
}


// En la primera iteración, se inserta en mejor_permutacion los trabajos 1 y 2 de la secuencia J con la ordenación que mejor FO presente. En caso de empate se seleccionará la última que se ha construido.

int[] mejor_permutacion = new int[2];

// Vector que recoge las distintas ordenaciones

int[] vector_aux = new int[2];

// Se comienza probando la primera ordenación (1-2)

vector_aux[0] = secuencia[0];
vector_aux[1] = secuencia[1];

FO1 = Sum_tiempo_acceso_ponderado(dia_disponible, fecha_limite, duracion_operacion, prioridad, cirujano_asociado, numero_quirofanos, disponibilidadquirofano_turno, horasmaximaquirofano_turno, numero_cirujanos, horascirujano_turno, dias, vector_aux);

// Se considera que 1 - 2 es la mejor permutación

Array.Copy(vector_aux, mejor_permutacion, 2);

// Se prueba la segunda ordenación (2-1)

vector_aux[0] = secuencia[1];
vector_aux[1] = secuencia[0];

FO2 = Sum_tiempo_acceso_ponderado(dia_disponible, fecha_limite, duracion_operacion, prioridad, cirujano_asociado, numero_quirofanos, disponibilidadquirofano_turno, horasmaximaquirofano_turno, numero_cirujanos, horascirujano_turno, dias, vector_aux);

// Se comprueba si finalmente 1-2 es la mejor ordenación y si no lo es, se actualiza como mejor_permutación la 2-1

if (FO2 <= FO1)
```

```
{
            Array.Copy(vector_aux, mejor_permutacion, 2);
}

// Vector que recoge la mejor permutación de una iteración para usarla en la siguiente

int[] vector_aux2 = new int[2];
Array.Copy(mejor_permutacion, vector_aux2, 2);

// Bucle para insertar el resto de los trabajos de la secuencia J (es decir, realiza las siguientes iteraciones)

for (int i = 2; i < secuencia.Length; i++)
{
            FO1 = 999; // Inicializado a un valor alto para que la mejor FO sea la de la primera permutación y
            sirva como comparativa con las siguientes

            // Cada vez que se introduce un nuevo trabajo de J a se aumenta en 1 el tamaño de todos los vectores
            auxiliares

            Array.Resize(ref vector_aux, vector_aux.Length + 1);
            Array.Resize(ref vector_aux2, vector_aux2.Length + 1);
            Array.Resize(ref mejor_permutacion, mejor_permutacion.Length + 1);

            // El trabajo de J a se introduce en la posición que de lugar a la mejor FO

            for (int j = 0; j <= i; j++)
            {
                        vector_aux[j] = secuencia[i];

                        for (int k = j + 1; k <= i; k++)
                        {

                                    vector_aux[k] = vector_aux2[k - 1];
                        }

                        for (int k = 0; k < j; k++)
                        {
                                    vector_aux[k] = vector_aux2[k];
                        }

                        FO2 = Sum_tiempo_acceso_ponderado (dia_disponible, fecha_limite,
                        duracion_operacion, prioridad, cirujano_asociado, numero_quirofanos,
                        disponibilidadquirofano_turno, horasmaximaquirofano_turno, numero_cirujanos,
                        horascirujano_turno, dias, vector_aux);

                        if (FO2 <= FO1)
                        {
                                    FO1 = FO2;
                                    Array.Copy(vector_aux, mejor_permutacion, vector_aux.Length);
                        }
            }

            Array.Copy(mejor_permutacion, vector_aux2, mejor_permutacion.Length);
}

// Una vez que se obtiene la mejor_permutación final (mejor secuencia que se ha obtenido del NEH), se copia
en secuencia para incorporarla en la población

mejorFO = FO1;
Array.Copy(mejor_permutacion, secuencia, m);
rellenar_fila_matriz(poblacion, secuencia, 0);
```

```
//SEGUNDO INDIVIDUO: Regla de despacho EDD (El trabajo con menor fecha de entrega se procesa primero)

for (int j = 0; j < m; j++)
{
        individuo[0, j] = j;
        individuo[1, j] = fecha_limite[j];
}

individuosalida = int_Quicksort (individuo, 0, m - 1);
rellenar_vector(individuosalida, secuencia, 0);
rellenar_fila_matriz(poblacion, secuencia, 1);

FO1 = Sum_tiempo_acceso_ponderado (dia_disponible, fecha_limite, duracion_operacion, prioridad, cirujano_asociado, numero_quirofanos, disponibilidadquirofano_turno, horasmaximaquirofano_turno, numero_cirujanos, horascirujano_turno, dias, secuencia);

if (FO1 < mejorFO)
{
        mejorFO = FO1;
}

// TERCER INDIVIDUO: Regla de despacho LPT (El trabajo con mayor tiempo de proceso se procesa primero)

for (int j = 0; j < m; j++)
{
        Individuo2[0, j] = j;
        Individuo2[1, j] = duracion_operacion[j];
}

Individuosalida2 = Quicksort_decre(individuo, 0, m - 1);

for (int j = 0; j < m; j++)
{
        individuosalida[0, j] = Convert.ToInt32(individuosalida2[0, j]);
        secuencia[j] = individuosalida[0, j];
}

rellenar_fila_matriz(poblacion, secuencia, 2);

FO1 = Sum_tiempo_acceso_ponderado(dia_disponible, fecha_limite, duracion_operacion, prioridad, cirujano_asociado, numero_quirofanos, disponibilidadquirofano_turno, horasmaximaquirofano_turno, numero_cirujanos, horascirujano_turno, dias, secuencia);

if (FO1 < mejorFO)
{
        mejorFO = FO1;
}

// CUARTO INDIVIDUO: Se va a generar ordenando los pacientes de mayor a menor prioridad

for (int j = 0; j < m; j++)
{
        Individuo2[0, j] = j;
        Individuo2[1, j] = prioridad [j];
}

Individuosalida2 = Quicksort_decre(individuo, 0, m - 1);

for (int j = 0; j < m; j++)
{
```

```
        individuosalida[0, j] = Convert.ToInt32(individuosalida2[0, j]);
        secuencia[j] = individuosalida[0, j];
}
rellenar_fila_matriz(poblacion, secuencia, 3);

FO1 = Sum_tiempo_acceso_ponderado(dia_disponible, fecha_limite, duracion_operacion, prioridad,
cirujano_asociado, numero_quirofanos, disponibilidadquirofano_turno, horasmaximaquirofano_turno,
numero_cirujanos, horascirujano_turno, dias, secuencia);

if (FO1 < mejorFO)
{
        mejorFO = FO1;
}

// INDIVIDUOS RESTANTES: hasta completar la población se generan de manera aleatoria.

int contador = 4;
while (contador < n)
{
        // Se asocian unos tiempos de proceso aleatorios a cada paciente

        for (int j = 0; j < m; j++)
        {
                individuo[0, j] = j;
                Random aleatorio = new Random();
                individuo[1, j] = aleatorio.Next(1, 50);
        }

        // Se obtiene una secuencia ordenada de pacientes en función de estos tiempos aleatorios que será
        una secuencia aleatoria

        individuosalida = int_Quicksort_decre(individuo, 0, m - 1);
        rellenar_vector(individuosalida, secuencia, 0);

        // Se introduce la secuencia en la población siempre que sea distinta a las que ya existían

        bool apto = no_repeticion(poblacion, secuencia);

        if (apto == true)
        {
                rellenar_fila_matriz(poblacion, secuencia, i);
                contador++;
                FO1 = Sum_tiempo_acceso_ponderado(dia_disponible, fecha_limite, duracion_operacion,
                prioridad, cirujano_asociado, numero_quirofanos, disponibilidadquirofano_turno,
                horasmaximaquirofano_turno, numero_cirujanos, horascirujano_turno, dias, secuencia);

                if (FO1 < mejorFO)
                {
                        mejorFO = FO1;
                }

        }
}

// RESCATE DEL TIEMPO ACTUAL

var timer = Stopwatch.StartNew();
double tiempoActualMilisegundos = timer.ElapsedMilliseconds;
double tiempoActual = tiempoActualMilisegundos / 1000;
```

```
// APLICACIÓN DE LAS 3 FASES DEL ALGORITMO (CICLO) HASTA SUPERAR EL TIEMPO LÍMITE

int tiempoLimite = 10 * numero_quirofanos * numero_cirujanos;
while (tiempoActual < tiempoLimite)
{
        // Vector que recogerá si un individuo ha mejorado o no en un ciclo

        bool[] mejora = new bool[n];

        for (int y = 0; y < ensayos; y++)
        {

                        // FASE DE LA ABEJA OBRERA

                // En este caso, la fase de la abeja obrera se replicará varias veces

                for (int x = 0; x < replicas; x++)
                {
                        // Para cada individuo de la población,

                        for (int i = 0; i < n; i++)
                        {
                                rellenar_vector(poblacion, secuencia, i);
                                rellenar_vector(poblacion, vecina, i);

                                // Se genera un vecino aplicando el intercambio adyacente múltiple

                                Intercambio_pares (vecina);

                                // Se acepta el vecino solo si es mejor

                                FO1 = Sum_tiempo_acceso_ponderado(dia_disponible, fecha_limite,
                                duracion_operacion, prioridad, cirujano_asociado, numero_quirofanos,
                                disponibilidadquirofano_turno,horasmaximaquirofano_turno,
                                numero_cirujanos, horascirujano_turno, dias, secuencia);

                                FO2 = Sum_tiempo_acceso_ponderado(dia_disponible, fecha_limite,
                                duracion_operacion, prioridad, cirujano_asociado, numero_quirofanos,
                                disponibilidadquirofano_turno,horasmaximaquirofano_turno,
                                numero_cirujanos, horascirujano_turno, dias, vecina);

                                if (FO2 < FO1)
                                {
                                        mejora[i] = true;
                                        rellenar_fila_matriz(poblacion, vecina, i);

                                        // Si el vecino que se incopora es mejor que la mejor_secuencia,
                                        se actualiza

                                        if (FO2 < mejorFO)
                                        {
                                                mejorFO = FO2;
                                        }
                                }
                        }
                }


                // FASE DE LA ABEJA OBSERVADORA
```

```
// Se escogen aleatoriamente dos individuos de la población

Random aleatorio3 = new Random();
int aux1 = aleatorio3.Next(0, n);
int aux2 = aleatorio3.Next(0, n);

// Se usa "vecina" y "secuencia" para no declarar dos nuevos vectores que recojan los dos individuos

rellenar_vector(poblacion, secuencia, aux1);
rellenar_vector(poblacion, vecina, aux2);

// Se selecciona el mejor de los dos

FO1 = Sum_tiempo_acceso_ponderado(dia_disponible, fecha_limite, duracion_operacion, prioridad, cirujano_asociado, numero_quirofanos, disponibilidadquirofano_turno, horasmaximaquirofano_turno, numero_cirujanos, horascirujano_turno, dias, secuencia);

FO2 = Sum_tiempo_acceso_ponderado(dia_disponible, fecha_limite, duracion_operacion, prioridad, cirujano_asociado, numero_quirofanos, disponibilidadquirofano_turno, horasmaximaquirofano_turno, numero_cirujanos, horascirujano_turno, dias, vecina);

if (FO2 <= FO1)
{
        Array.Copy(vecina, secuencia, m);
}

else
{
        Array.Copy(secuencia, vecina, m);
}

// Se genera un vecino del mejor individuo seleccionado mediante intercambio adyacente múltiple

Intercambio_pares (vecina);

FO2 = Sum_tiempo_acceso_ponderado(dia_disponible, fecha_limite, duracion_operacion, prioridad, cirujano_asociado, numero_quirofanos, disponibilidadquirofano_turno, horasmaximaquirofano_turno, numero_cirujanos, horascirujano_turno, dias, vecina);

// Se comienza suponiendo que el vecino va a ser peor que todos los individuos de la población. Es decir, no existe ningun individuo de la población (ninguna fila) para la cual la FO es peor que la FO vecina, y en principio no se aceptaría.

int peorfila = -1;

for (int i = 0; i < n; i++)
{
        rellenar_vector(poblacion, secuencia, i);

        FO1 = Sum_tiempo_acceso_ponderado(dia_disponible, fecha_limite, duracion_operacion, prioridad, cirujano_asociado, numero_quirofanos, disponibilidadquirofano_turno,horasmaximaquirofano_turno, numero_cirujanos, horascirujano_turno, dias, secuencia);

        if (FO1 > FO2)
        {
                FO2 = FO1;
                peorfila = i;
        }
}
```

```
// Si se demuestra que sí que existe una fila en la cual la FO es peor que la vecina,

if (peorfila >= 0)
{
        // y además se comprueba que la vecina no se repite con ninguna otra ya existente en
        la población, se acepta

        bool apto = no_repeticion(poblacion, vecina);

        if (apto == true)
        {
                rellenar_fila_matriz(poblacion, vecina, peorfila);
                mejora[peorfila] = true;

                FO2 = Sum_tiempo_acceso_ponderado(dia_disponible, fecha_limite,
                duracion_operacion, prioridad, cirujano_asociado, numero_quirofanos,
                disponibilidadquirofano_turno,horasmaximaquirofano_turno,
                numero_cirujanos, horascirujano_turno, dias, vecina);

                // Si el vecino que se incopora es mejor que la mejor_secuencia, se actualiza

                if (FO2 < mejorFO)
                {
                        mejorFO = FO2;
                }
        }
}
}

// FASE ABEJA EXPLORADORA

// Para cada individuo de la población que no se haya mejorado,

for (int i = 0; i < n; i++)
{
        if (mejora[i] == false)
        {
                rellenar_vector(poblacion, secuencia, i);

                FO1= Sum_tiempo_acceso_ponderado(dia_disponible, fecha_limite,
                duracion_operacion, prioridad, cirujano_asociado, numero_quirofanos,
                disponibilidadquirofano_turno, horasmaximaquirofano_turno, numero_cirujanos,
                horascirujano_turno, dias, secuencia);

                // se generan 3 vecinos mediante inserción. Debe hacerse una copia para que todos
                se generen a partir de la secuencia que no ha mejoradO

                int[] copia = new int[m];

                FO2= 999; // Para comenzar suponiendo que la mejor es la primera que se genera,
                se inicializa la variable con un valor muy alto.

                for (int j = 0; j < numero_vecinos; j++)

                {
                        Array.Copy(secuencia, copia, secuencia.Length);

                        insercion_aleatoria(copia);

                        FO1 = Sum_tiempo_acceso_ponderado(dia_disponible, fecha_limite,
                        duracion_operacion, prioridad, cirujano_asociado, numero_quirofanos,
```

```
                                        disponibilidadquirofano_turno,horasmaximaquirofano_turno,
                                        numero_cirujanos, horascirujano_turno, dias, copia);

                                        // Por cada vecino que se genera, se comprueba si es mejor que los que
                                        se habían generado previamente para quedarse con el mejor de todos

                                        if (FO1 < FO2)
                                        {
                                                FO2 = FO1;
                                                Array.Copy(copia, vecina, m);
                                        }

                                }

                                // Si el mejor vecino que se incopora es mejor que la mejor_secuencia, se actualiza

                                if (FO2 < mejorFO)
                                {
                                        mejorFO = FO2;
                                }

                                rellenar_fila_matriz(poblacion, vecina, i);
                        }
                }

                // ACTUALIZACIÓN DEL TIEMPO

                tiempoActualMilisegundos = timer.ElapsedMilliseconds;
                tiempoActual = tiempoActualMilisegundos / 1000;
        }

        // IMPRESIÓN EN EXCELL

        FicheroCSV1.Write(alpha + ";" + beta + ";" + mds + ";" + numero_cirujanos + ";" + numero_quirofanos +
        ";" + instancia + ";" + mejorFO + "\n");
    }

public static void rellenar_fila_matriz(int[,] poblacion, int[] secuencia, int fila_que_relleno)
    {
        for (int j = 0; j < secuencia.Length; j++)
        {
                poblacion[fila_que_relleno, j] = secuencia[j];
        }
    }

public static void rellenar_vector(int[,] poblacion, int[] secuencia, int fila_que_copio)
    {
        for (int j = 0; j < secuencia.Length; j++)
        {
                secuencia[j] = poblacion[fila_que_copio, j];
        }
    }

public static int[,] int_Quicksort_decre(int[,] elements, int left, int right)
    {
        int i = left, j = right;
        int pivot = elements[1, (left + right) / 2];

        while (i <= j)
        {
                while (elements[1, i].CompareTo(pivot) > 0)
                {
```

```
                i++;
            }

            while (elements[1, j].CompareTo(pivot) < 0)
            {
                j--;
            }

            if (i <= j)
            {
                // Swap

                int tmp0 = elements[0, i];
                int tmp1 = elements[1, i];
                elements[0, i] = elements[0, j];
                elements[1, i] = elements[1, j];
                elements[0, j] = tmp0;
                elements[1, j] = tmp1;

                i++;
                j--;
            }
        }

        // Recursive calls

        if (left < j)
        {
          int_Quicksort_decre(elements, left, j);
        }

        if (i < right)
        {
          int_Quicksort_decre(elements, i, right);
        }

        return elements;
    }

public static int[,] int_Quicksort(int[,] elements, int left, int right)
    {
        int i = left, j = right;
        int pivot = elements[1, (left + right) / 2];

        while (i <= j)
        {
            while (elements[1, i].CompareTo(pivot) < 0)
            {
                i++;
            }

            while (elements[1, j].CompareTo(pivot) > 0)
            {
                j--;
            }

            if (i <= j)
            {
                //Swap

                int tmp0 = elements[0, i];
                int tmp1 = elements[1, i];
```

```
                        elements[0, i] = elements[0, j];
                        elements[1, i] = elements[1, j];
                        elements[0, j] = tmp0;
                        elements[1, j] = tmp1;

                        i++;
                        j--;
                    }
            }

            // Recursive calls

            if (left < j)
            {
                    int_Quicksort(elements, left, j);
            }

            if (i < right)
            {
                    int_Quicksort(elements, i, right);
            }
            return elements;
        }

    public static double[,] Quicksort_decre(double[,] elements, int left, int right)
        {
            int i = left, j = right;
            double pivot = elements[1, (left + right) / 2];

            while (i <= j)
            {
                    while (elements[1, i].CompareTo(pivot) > 0)
                    {
                        i++;
                    }

                    while (elements[1, j].CompareTo(pivot) < 0)
                    {
                        j--;
                    }

                    if (i <= j)
                    {
                        // Swap
                        double tmp0 = elements[0, i];
                        double tmp1 = elements[1, i];
                        elements[0, i] = elements[0, j];
                        elements[1, i] = elements[1, j];
                        elements[0, j] = tmp0;
                        elements[1, j] = tmp1;

                        i++;
                        j--;
                    }
            }

            // Recursive calls
            if (left < j)
            {
                    Quicksort_decre(elements, left, j);
            }
```

```
        if (i < right)
        {
            Quicksort_decre(elements, i, right);
        }

        return elements;
    }

public static bool no_repeticion(int[,] poblacion, int[] secuencia)
    {
        bool secuencia_apta = true; // Si la secuencia no se repite con ninguna fila será true y si se repite será false
        bool[] fila_norepetida = new bool[poblacion.GetLength(0)]; // Si la fila no se repite con la secuencia será true
        y si se repite será false

        // Se compara la secuencia con cada fila de la matriz

        for (int i = 0; i< poblacion.GetLength(0); i++)
        {
            for (int i = 0; i < poblacion.GetLength(0); i++)
            {
                // En el momento en el haya un elemento que no se repita, la secuencia ya no se repite con
                esa fila
                if (poblacion[i, j] != secuencia[j])
                {
                    fila_norepetida[i] = true;
                }
            }

        }

        // Si ninguna fila se repite con la secuencia, es porque la secuencia es apta para incorporarse a la población
        for (int i = 0; i< poblacion.GetLength(0); i++)
        {
            if (fila_norepetida[i] == false)
            {
                secuencia_apta = false;
            }
        }

        return secuencia_apta;
    }

public static void intercambio_multiple(int[] vecina)
    {
        for (int i = 0; i < 2; i++)
        {
            Random aleatorio = new Random();
            int posicion = aleatorio.Next(0, vecina.Length);
            int auxi = vecina[posicion];

            // En el caso de que la posición elegida sea la última de la secuencia

            if (posicion == vecina.Length - 1)
            {
                vecina[posicion] = vecina[0];
                vecina[0] = auxi;
            }

            else
            {
                vecina[posicion] = vecina[posicion + 1];
                vecina[posicion + 1] = auxi;
```

```
        }
    }

public static void insercion_aleatoria(int[] secuencia)
    {
        Random aleatorio = new Random();
        int posicionextraccion = aleatorio.Next(0, secuencia.Length);
        int posicioninsercion = aleatorio.Next(0, secuencia.Length);

        // Para evitar que no se genere un nuevo vecino, se prohíbe la inserción del trabajo en la posición de extracción

        while (posicionextraccion == posicioninsercion)
        {
            posicioninsercion = aleatorio.Next(0, secuencia.Length);
        }


        // Se utiliza una copia para guardar las posiciones originales

        int[] copia = new int[secuencia.Length];
        Array.Copy(secuencia, copia, secuencia.Length);

        secuencia[posicioninsercion] = secuencia[posicionextraccion];

        // Si la inserción es de un elemento en una posición hacia delante

        if (posicionextraccion > posicioninsercion)
        {
            for (int i = posicioninsercion + 1; i <= posicionextraccion; i++)
            {
                secuencia[i] = copia[i - 1];
            }
        }

        // Si la inserción es de un elemento en una posición hacia atrás

        else
        {
            for (int i = posicionextraccion; i < posicioninsercion; i++)
            {
                secuencia[i] = copia[i + 1];
            }
        }

    }

 public static void intercambio_pares(int[] vecina)
    {
            Random aleatorio = new Random();
            int posicion1 = aleatorio.Next(0, vecina.Length);
            int posicion2 = aleatorio.Next(0, vecina.Length);

            while (posicion1 == posicion2)
            {
                posicion2 = aleatorio.Next(0, vecina.Length);
            }

            int auxi = vecina[posicion1];
            vecina[posicion1] = vecina[posicion2];
            vecina[posicion2] = auxi;
    }
```

```
public static int Sum_tiempo_acceso_ponderado(int[] dia_disponible, int[] fecha_limite, double[] duracion_operacion, double[] prioridad, int[] cirujano_asociado, int numero_quirofanos, int[,] disponibilidadquirofano_turno, double[] horasmaximaquirofano_turno, int numero_cirujanos, double[,] horascirujano_turno, int dias, int[] secuencia)
    {
        // Variables necesarias para calcular el tardiness

        double[] tardiness = new double[secuencia.Length];
        double[] completiontimetarea = new double[secuencia.Length];
        double Sumtj_ponderado = 0;

        // Recoge el inicio de cada tarea para evitar el solape

        double[,,] iniciotarea_quirofano_turno = new double [secuencia.Length, numero_quirofanos, dias * 2];

        // La carga de cada cirujano en cada turno es necesaria saberla para no superar las horas que puede trabajar un cirujano en turno

        double[,] cargadelcirujano_elturno = new double[numero_cirujanos, dias * 2];

        // Además se debe recoger la carga del cirujano en cada quirófano cada turno porque se necesita a la hora de adjudicar pacientes a quirófanos (Regla de asignación)

        double[,,] cargadelcirujano_enelquirofano_elturno = new double[numero_cirujanos, numero_quirofanos, dias * 2];

        // Recoge la carga de cada quirófano para no superar las horas operativas de un quirófano en turno

        double[,] cargadelquirofano_elturno = new double[numero_quirofanos, dias * 2];

        // Se programan los pacientes: El objetivo es encontrar qué turno y en qué quirófano se opera

        for (int i = 0; i < secuencia.Length; i++)
        {
            // Esta variable recoge el instante de programación en el que se está comprobando si se puede operar al paciente (se inicia en el primer turno)

            int turno = 0;

            // Se comienza sin haberle asignado ningún turno de operación al paciente

            int turno_operacion = -1;

            // Se realiza un proceso iterativo de búsqueda de un turno de operación que cumpla RESTRICCIÓN 1: El cirujano asignado debe disponer de horas de trabajo para realizar la operación en el turno en el que se está comprobando, además, el turno debe ser mayor que el release date

            while (turno_operacion == -1)
            {
                // Siempre que el turno sea anterior al release date o no se cumpla que existen horas disponibles del cirujano asignado, se postpone la operación al turno siguiente

                if (cargadelcirujano_elturno[cirujano_asociado[secuencia[i]], turno] + duracion_operacion[secuencia[i]] > horascirujano_turno[cirujano_asociado[secuencia[i]], turno] || turno < dia_disponible[secuencia[i]]*2)
                {
                    turno++;
                }

                // Si por el contrario se cumple la restricción, se asocia la operación a ese turno
```

```
			else
			{
				turno_operacion = turno;
			}

			// En el caso de que la operación se haya postpuesto para un turno que quede fuera del horizonte de planificación, se sale del bucle asignando el paciente a ese turno, aunque quede fuera

			if (turno == dias * 2)
			{
				turno_operacion = turno;
			}
		}

		// Si se encuentra un turno de operación que cumpla la RESTRICCIÓN 1 dentro del horizonte de planificación, se procede a la busqueda de quirófano

		if (turno_operacion < dias * 2)
		{
			// Variable que recoge la carga máxima del cirujano (Regla asignación). El valor -1 es para que en el bucle se asigne como quirofano_asignado al primero disponible y a partir de este se vaya comparando con el resto de los que están disponibles y se seleccione el máximo

			double maxima_carga_del_cirujano = -1;

			// De momento no se ha asignado el paciente a ningún quirófano

			int quirofano_asignado = -1;

			// Recoge el instante en que puede comenzar la operación de un paciente en un quirófano

			double comienzo = 0;

			// Se realiza un proceso iterativo de búsqueda de quirófano que cumpla la RESTRICCIÓN 2: El quirófano en el que se asigna el paciente debe disponer de horas operativas para poder realizar la operación. Además, se deben evitar los solapes de las operaciones que comparten el mismo cirujano y ya están asignadas en otro quirófano

			do
			{
				for (int j = 0; j < numero_quirofanos; j++)
				{
					// Para cada quirófano que estan disponibles en ese turno, se introduce el paciente en aquel en el que la carga del cirujano asociado sea máxima (Regla de asignación)

					if (disponibilidadquirofano_turno[j, turno_operacion] == 1 && cargadelquirofano_elturno[j, turno_operacion] + duracion_operacion [secuencia[i]] <= horasmaximaquirofano_turno[turno_operacion] && cargadelcirujano_enelquirofano_elturno[cirujano_asociado[secuencia[i] ], j, turno_operacion] > maxima_carga_del_cirujano)
					{
						// En principio la actividad comenzaría justo después de la última que se haya planificado en ese quirófano

						comienzo = cargadelquirofano_elturno[j, turno_operacion];

						while (comienzo + duracion_operacion[secuencia[i]] <= horasmaximaquirofano_turno[turno_operacion])
						{
```

```
                        // Pero se comprueba que situándola ahí no solapa con
                        otra operación que comparte cirujano en otro quirófano
                        del mismo turno

                        for (int k = 0; k < numero_quirofanos && j != k; k++)
                        {

                            for (int l = 0; l < i &&
                            cirujano_asociado[secuencia[i]] ==
                            cirujano_asociado[secuencia[l]]; l++)
                            {
                                // Si solapa, se retrasa el comienzo
                                hasta que no solape

                                if(comienzo<
                                iniciotarea_quirofano_turno[l, k,
                                turno_operacion] +
                                duracion_operacion[secuencia[l]])
                                {
                                    comienzo =
                                    iniciotarea_quirofano_turno
                                    [l, k, turno_operacion] +
                                    duracion_operacion[secuen
                                    cia[l]];
                                }
                            }
                        }

                        // Si se ha llegado hasta aquí es porque se cumplen todas
                        las restricciones incluida el solape, por lo que el
                        paciente se asigna a ese quirófano

                        quirofano_asignado = j;

                        maxima_carga_del_cirujano=
                        cargadelcirujano_enelquirofano_elturno[cirujano_asoc
                        iado[secuencia[i]],quirofano_asignado,
                        turno_operacion] + duracion_operacion[secuencia[i]];

                        break; // Salida del while
                    }
                }
            }

            // Si no se ha podido asignar el paciente a ningún quirófano en ese turno, se busca
            otro turno para comprobar posteriormente si existe un quirófano disponible en él

            if (quirofano_asignado == -1)
            {
                int aux3 = -1;

                // Se busca un nuevo turno que cumpla la RESTRICCIÓN 1

                while (aux3 == -1)
                {
                    // Si hay más turnos para buscar dentro del horizonte de
                    planificación,

                    if (turno_operacion < (dias * 2) - 1)
                    {
                        // se comprueba que se cumplen las restricciones en el
                        siguiente turno
```

```
                                        turno_operacion++;

                                        // Si el cirujano puede operar en ese turno, ya se ha
                                        encontrado nuevo turno por lo que se sale de ese bucle,
                                        pero no de la búsqueda de quirófano.

                                        if
                                        (cargadelcirujano_elturno[cirujano_asociado[secuenci
                                        a[i]], turno_operacion] + duracion_operacion
                                        [secuencia[i]] <= horascirujano_turno
                                        [cirujano_asociado[secuencia[i]], turno_operacion])
                                        {
                                                aux3 = 0;
                                        }
                                }

                                // Si no hay más turnos en horizonte se sale del bucle de
                                búsqueda de nuevo turno y por tanto de la búsqueda también de
                                quirófano porque no se puede operar

                                else
                                {
                                        aux3 = 0;
                                        quirofano_asignado = 4;
                                }
                        }
                }
        }

        while (quirofano_asignado == -1);

        // Si se ha podido encontrar un quirófano dentro del horizonte de planificación, se actualizan
        las variables

        if (quirofano_asignado != 4)
        {

                iniciotarea_quirofano_turno[i, quirofano_asignado, turno_operacion] =
                comienzo;

                cargadelcirujano_enelquirofano_elturno[cirujano_asociado[secuencia[i]],
                quirofano_asignado, turno_operacion] =
                cargadelcirujano_enelquirofano_elturno[cirujano_asociado[secuencia[i]],
                quirofano_asignado, turno_operacion] + duracion_operacion[secuencia[i]];

                cargadelcirujano_elturno[cirujano_asociado[secuencia[i]], turno_operacion] =
                cargadelcirujano_elturno[cirujano_asociado[secuencia[i]], turno_operacion] +
                duracion_operacion[secuencia[i]];

                cargadelquirofano_elturno[quirofano_asignado, turno_operacion] = comienzo +
                duracion_operacion[secuencia[i]];

                completiontimetarea[i] = turno_operacion;

                tardiness[i] = completiontimetarea[i] - fecha_limite[i];

                if (tardiness[i] < 0)
                {
                        tardiness[i] = 0;
                }
```

```
				Sumtj_ponderado = Sumtj_ponderado + (tardiness[i] * prioridad[i]);
			}

			// Si en cambio el quirófano se ha encontrado fuera del horizonte de planificación, se
			penaliza la función objetivo porque el paciente no se opera

			else
			{
				Sumtj_ponderado = Sumtj_ponderado + 0.001;
			}
		}

		// Si directamente el turno que cumple con la RESTRICCIÓN 1 está fuera del horizonte de
		planificación, se penaliza la función objetivo porque el paciente no se opera

		else
		{
			Sumtj_ponderado = Sumtj_ponderado + 0.001;
		}
	}

	return Sumtj_ponderado;
}
```