Proyecto Fin de Grado
Ingeniería de Organización Industrial

Programación de la producción en entornos sanitarios: resolución mediante algoritmo iterativo con restricción de camas

Autor: José María González Nieto

Tutor: Víctor Fernández-Viagas Escudero y Carla Talens Fayos

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

Sevilla, 2021

OPTIMIZACIÓN INDUSTRIAL

Proyecto Fin de Grado

Ingeniería de Organización Industrial

Programación de la producción en entornos sanitarios: resolución mediante algoritmo iterativo con restricción de camas

Autor:

José María González Nieto

Tutores:

Víctor Fernández-Viagas Escudero

Carla Talens Fayos

Dpto. de Organización Industrial y Gestión de Empresas I

Escuela Técnica Superior de Ingeniería

Universidad de Sevilla

Sevilla, 2021

Trabajo Fin de Grado: Programación de la producción en entornos sanitarios: resolución mediante algoritmo iterativo con restricción de camas

Autor: José María González Nieto
Tutores: Víctor Fernández-Viagas Escudero y Carla Talens Fayos

El tribunal nombrado para juzgar el trabajo arriba indicado, compuesto por los siguientes profesores:

Presidente:

Vocal/es:

Secretario:

acuerdan otorgarle la calificación de:

El Secretario del Tribunal

Fecha:

Agradecimientos

Me gustaría empezar por agradecer a mi tutor, Víctor Fernández-Viagas Escudero, todo el tiempo que me ha dedicado durante los últimos meses, su atención y su pronta respuesta a mis dudas. No podría haber pedido un mejor tutor para este trabajo que presento. También agradecer la colaboración de Carla Talens Fayos que ha atendido a las tutorías para ofrecer su visión del trabajo y ayudarme en su realización.

Por supuesto, quiero agradecer a mi familia, en especial a mis padres y mi hermano que son los que más me han apoyado siempre. A mi padre por hacerme competitivo desde pronta edad. A mi madre por mantenerme durante toda mi formación de pequeño trabajando duro y animándome a no abandonar. Y a ellos dos y mis cuatro abuelos por animarme continuamente a convertirme en mi mejor versión. A mi hermano le agradezco su inestimable confianza en mí, que le hace no dudar en apoyarme en cada decisión que tomo.

Por último, agradecer a los compañeros que me han acompañado a lo largo de estos 4 años de grado, las clases y las horas de estudio no habrían sido lo mismo sin la ayuda mutua, el apoyo y la amistad forjada en este tiempo. Esas amistades que me llevo de mi estancia en esta facultad y en Sevilla en general espero que me acompañen siempre.

José María González Nieto

Sevilla, 2021

Resumen

El problema de decisión que se ha tratado busca determinar cuál es la mejor forma de integrar las camas en la programación de las operaciones de los pacientes en lista de espera quirúrgica de la especialidad de Cirugía Plástica y Quemaduras Mayores del Hospital Universitario “Virgen del Rocío”. Tradicionalmente, este problema no es abordado porque las camas necesarias para el posoperatorio no suelen suponer un cuello de botella para el problema de programación, pero este año, con la pandemia, se ha podido comprobar que en muchos hospitales la falta de camas ha supuesto un problema en el calendario de operaciones que se tenían programadas.

Con el ánimo de comprobar hasta qué punto afecta esta situación a los pacientes que se pueden operar y las tardanzas de los mismos, se han estudiado tres formas diferentes de integrar las camas en el problema, una de las cuáles sería la de uso más común, la propagación, pero con la comprobación de validez que requiere la situación especial que se estudia.

Por encima de esta integración, se ha empleado un algoritmo iterativo con recocido simulado para buscar la mejor solución para unos datos generados en el propio problema que buscan simular la situación en la especialidad del hospital. Este algoritmo iterativo tiene tres variantes distintas.

Las diferentes propuestas que se plantean buscan establecer una programación de las operaciones que cumpla con las restricciones del problema. Fundamentalmente lo que se trata es de encontrar la secuencia de pacientes que reduzca la tardanza de estos y que además consiga introducir en el horizonte temporal todas las operaciones posibles. La tardanza de los pacientes viene determinada por la fecha que se ha establecido como máxima para que el paciente se opere, la fecha de vencimiento o límite. Ambos factores, tardanza y si el paciente es operado o no en el horizonte temporal estudiado, serán ponderados según la prioridad del paciente.

Índice

Índice de figuras

Índice de tablas

1. Introducción

"Las circunstancias no hacen al hombre,
solo lo revelan"

*- **Epicteto***

1.1 Programación de la producción

La programación de la producción puede ser definida como la asignación de recursos disponibles durante un tiempo y satisfaciendo unos criterios. El problema de programación involucra una serie de tareas que se deben de ejecutar, y los criterios pueden implicar una compensación entre la finalización pronta o tardía de la tarea, y entre mantener inventario para la tarea o frecuentes cambios de producción (Graves, 1981). No se puede subestimar la importancia de adoptar buenas estrategias de programación en los entornos de producción. Es necesario ser capaces de responder rápidamente a la demanda de mercado y ser eficientes con los recursos en los competitivos mercados de hoy. Estas necesidades aumentan la complejidad de los problemas en los entornos de producción (MacCarthy y Liu, 1993).

Dentro de la organización de la empresa, se pueden distinguir tres niveles en la toma de decisiones:

- Estratégico: las decisiones a este nivel son a largo plazo y determinan las metas y la dirección de toda la empresa. Se toman en niveles superiores de la empresa puesto que afectan a toda la organización y son conceptuales, es decir, no son específicas, ya que deben poder ser aplicadas a niveles más bajos.
- Táctico: aquí se encontrarían las decisiones tomadas por los mandos medios de la empresa para el desarrollo de tácticas que permitan cumplir con las metas estratégicas. Son decisiones más orientadas a las acciones y a la asignación de recursos, son más específicas y concretas.
- Operativo: las decisiones operativas son las que cuentan con un mayor nivel de detalle pues son las decisiones que se implementan a diario. Buscan ser las decisiones que cumplan con las metas del nivel superior de forma más eficaz y eficiente. En estas decisiones, la infraestructura está dada y no es variable en este nivel. (Framinan et al., 2014)

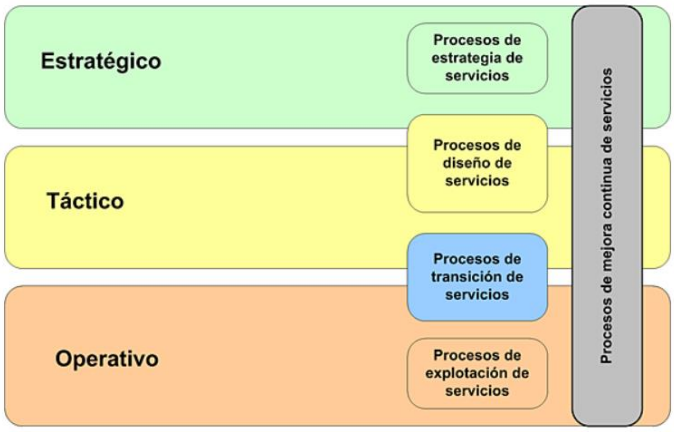

Figura 1. Niveles de la toma de decisiones (Entender ITIL 2011 Normas y mejores prácticas para avanzar hacia ISO 20000 – Los niveles decisionales, n.d.)

La programación de la producción está integrada dentro de las decisiones de nivel operativo y

es un paso fundamental para la posterior ejecución de operaciones en el entorno productivo. En esta programación son conocidos los recursos productivos con los que se va a contar, las tareas que se tienen que realizar y los recursos que estas tareas necesitan. En principio, también se conocerá el tiempo que los recursos deben emplear en las tareas, aunque si el enfoque es estocástico estos tiempos tendrán un componente de aleatoriedad. Por otro lado, los modelos con el acercamiento determinista ignoran las incertidumbres considerando normalmente la media de los rendimientos de los procesos. Este método suele resultar en inventario extra, entre otros factores. Los modelos con el acercamiento estocástico están basados en la generación de distintos escenarios, por lo que se puede considerar un enfoque más proactivo al lidiar con la incertidumbre. Su principal inconveniente es la complejidad en su modelado (Vahidian, 2012). Este trabajo fin de grado se centra en el enfoque determinista. Se ha elegido para poder encontrar soluciones a los problemas reales en tiempos razonables.

El resultado de la programación de la producción es el programa de producción, el cual se puede expresar como un cronograma detallado. En este cronograma, habitualmente representado como un diagrama de Gantt, se puede observar la solución al problema de programación, es decir, cuándo se empiezan a procesar los trabajos y en qué máquinas se hace. Con el fin de obtener una buena solución en el programa de producción, se desarrolla un algoritmo que genere posibles soluciones y seleccione las que van obteniendo mejores resultados. El modelado y los resultados dependen de los trabajos, las máquinas y los criterios. Todos estos factores tienen unas características propias: los trabajos requieren de una serie de operaciones y de unos recursos, además tienen un tiempo asociado para las operaciones, suelen tener fechas a partir de las cuales se puede empezar a operar con ellos y fechas en las que los trabajos debieran acabar de ser procesados; las máquinas son necesarias para el tratamiento de los trabajos, estas pueden tener horarios asociados y capacidades máximas de trabajos para ser ejecutados en un mismo momento, por nombrar un par de posibles características; por otro lado, los criterios se refieren a medidas de rendimiento.

Las características de máquinas y trabajos son las que marcan las restricciones que tiene el problema. Son las que responden a la pregunta de por qué no todas las soluciones son válidas.

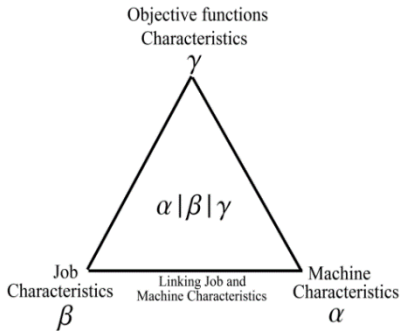

Figura 2. Clasificación de modelos de programación de la producción (Framinan et al., 2014)

Este trabajo se centra así en el desarrollo de diferentes algoritmos de programación de los quirófanos con integración de camas postoperatorias, con el objetivo de encontrar la mejor solución para obtener una secuencia válida que los quirófanos puedan emplear para la operación de los pacientes. Estos algoritmos implementan diversas posibilidades de integración de las camas postoperatorias.

La creación de diferentes modelos busca arrojar luz sobre un nuevo problema causado por la pandemia, que las camas se conviertan en un cuello de botella en el problema de programación de quirófanos. En este punto ya no se podría aislar la etapa del operatorio y luego propagar la solución a las camas, se necesita una integración de las camas en la toma de decisiones. En estos modelos se han considerado las etapas del operatorio y el posoperatorio. Para el operatorio se han obtenido todos los datos, que se consideran deterministas, de la anterior etapa del

preoperatorio, mientras que el posoperatorio se basa en una estimación de los tiempos medios que los pacientes suelen pasar en las camas de recuperación cuando salen de las operaciones. Así, el problema considera como trabajos a los pacientes, y como máquinas a los quirófanos y a los cirujanos asignados a los pacientes para la operación. Ambos son tratados como máquinas paralelas. Por supuesto, tanto quirófano como cirujano son fundamentales para que la operación se lleve a cabo y tienen que estar disponibles en el mismo instante de tiempo.

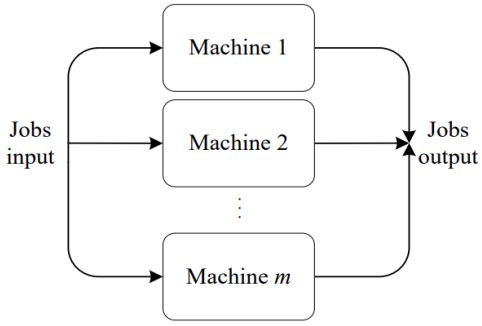

Figura 3. Esquema de flujo de trabajos en las máquinas paralelas
(Framinan et al., 2014)

1.2 Programación de la producción en entornos sanitarios

En el contexto económico actual, los quirófanos están considerados como una actividad crítica en la gestión del cuidado de la salud. Los quirófanos son una importante fuente de gastos para los hospitales, principalmente por su gasto en recursos materiales y humanos. Si se quiere buscar reducir los costes de los quirófanos, se hace necesaria una planificación consciente de estos y de los recursos que necesitan. Además, es importante incluir en la planificación del calendario la comodidad del equipo sanitario, de los recursos humanos de la organización (Roland et al., 2010).

Las largas listas de espera es uno de los mayores problemas que afronta el sistema público de sanidad de España. Estas listas constituyen un problema serio porque comprometen a los hospitales que las sufren, ya que revelan su incapacidad para satisfacer la demanda en periodos de tiempo que puedan ser considerados como adecuados. Sin embargo, en el caso de España, al estar hablando de un sistema público, el problema ya no es solo a nivel sanitario, lo es a nivel político, y se hace más importante conforme más población se preocupa por el tema (González-Bustos & García, 1999).

La gestión de los quirófanos no es una tarea fácil, al haber choque de intereses entre diferentes interesados y escasez de los recursos necesarios al ser estos de elevado coste. Además, hay que tener en cuenta que la demanda no va a dejar de crecer debido al envejecimiento de la población en los países más desarrollados. Todos estos factores que se han mencionado exigen eficiencia y para ello se requiere el desarrollo de una buena planificación y programación (Cardoen et al., 2010). Es decir, se hace imprescindible definir: qué materiales y recursos van a ser necesarios, cómo se van a llevar a cabo las operaciones y cuándo se van a llevar a cabo.

Las decisiones llevadas a cabo en la planificación y la programación forman parte del nivel operativo. Por supuesto, depende de las decisiones que se hayan tomado en las esferas más altas a nivel estratégico y a nivel táctico. Con estas decisiones a nivel estratégico se podría concluir que se necesita aumentar la capacidad operativa del hospital, y a nivel táctico tratar de ampliar los quirófanos, pero a nivel operativo hay que trabajar con lo que existe actualmente.
Este trabajo se centra en la especialidad de Cirugía Plástica y Quemados Mayores del Hospital Universitario Virgen del Rocío de Sevilla. El problema de decisión es el de la asignación de unos determinados pacientes en lista de espera a un quirófano en un día y una hora determinadas. Y el principal objetivo será el de minimizar los pacientes que se operan más tarde de su fecha límite.

1.3 Algoritmos

Los problemas, desde la perspectiva de la complejidad computacional, pueden ser clasificados en dos tipos: problemas polinomiales (P) y problemas no polinomiales (NP). El problema presentado en el actual trabajo es NP-hard al ser una generalización del taller de flujo híbrido con dos etapas. Por tanto, lo más conveniente es el uso de métodos aproximados con los que se puedan obtener buenos resultados tanto en calidad como en tiempo. Para este tipo de problemas, los algoritmos exactos no suelen tener un buen rendimiento en la literatura, por lo que se han desarrollado muchos algoritmos aproximados con la habilidad de proporcionar soluciones de calidad a los problemas combinatorios en breves tiempos computacionales. Las técnicas heurísticas y metaheurísticas están incluidas en estos algoritmos aproximados (de Antonio Suárez, 2014). Las heurísticas pueden ser definidas como algoritmos dependientes del problema, es decir, se define una heurística para un problema determinado. Mientras que las metaheurísticas son genéricas, con pocos cambios se pueden emplear en toda una variedad de problemas.

El algoritmo que se utiliza en el presente trabajo es el algoritmo codicioso iterativo (*Iterated Greedy*) es un algoritmo metaheurístico de búsqueda que itera usando la ejecución repetida de dos fases principales, la destrucción parcial de un candidato y la reconstrucción del mismo. Está basado en un principio simple, pero a pesar de su simplicidad ha servido para la creación de algoritmos de alto rendimiento. En combinación con otras metaheurísticas de optimización como la búsqueda local, ha tenido resultado muy interesantes en su aplicación en multitud de problemas (Ruiz & Stüzle, 2007).

En la literatura no hay un acuerdo completo acerca de si el algoritmo codicioso iterativo entra en la definición de heurística o metaheurística. En este trabajo se ha decidido adoptar la visión de la metaheurística por entender que las heurísticas son algoritmos que paran en un tiempo fijo definido por las variables del problema. Las metaheurísticas serían algoritmos que podrían ser alargados hasta el infinito si no se impusiera un tiempo de finalización.

2. Descripción del problema

"Las obras se tienen medio terminadas
cuando se han comenzado bien"

*- **Séneca***

El problema que se aborda en este Trabajo Fin de Grado se localiza en la especialidad de Cirugía Plástica y Quemaduras Mayores del Hospital Universitario Virgen del Rocío. Poniendo en contexto, la especialidad ejecuta alrededor de 3000 cirugías por año, incluyendo emergencias. Más específicamente, la especialidad cuenta con 4 quirófanos, uno de los cuales no va a estar disponible para las operaciones listadas en espera, ya que será reservado para las emergencias. Por lo que, al final, existen 3 quirófanos disponibles para estas cirugías (Molina-Pariente et al., 2015). Estos 3 quirófanos no están siempre disponibles, siguen unos horarios. Por las mañanas todos los quirófanos pueden ser usados de 8:30 a 14:30, es decir, durante 6 horas. Por las tardes solo 1 quirófano de los anteriores va a poder emplearse, y su horario es de 15:30 a 19:30, está abierto durante 4 horas. La cantidad de quirófanos disponibles se representa por Q.

En la especialidad se tiene acceso a dos tipos de camas (representadas por C) para la post operación del paciente. Por un lado, las camas PACU, que son unidades destinadas al ingreso de pacientes que han sido sometidos a anestesia general, regional o alguna sedación considerable. Su objetivo es la recuperación de las funciones orgánicas y los reflejos vitales del paciente, que pueden ser anulados tras la anestesia. En estas camas se monitorea el pulso del paciente, su presión sanguínea y sus niveles de oxígeno. Estas camas las necesitan más de la mitad de los pacientes, pues son sometidos a dosis de anestesia suficientes como para ser monitoreados durante un tiempo, que generalmente suele ser corto, entre 1 y 3 horas.

Por otro lado, las camas UCI, son camas localizadas en la Unidad de Cuidados Intensivos. En estas unidades se proporciona medicina intensiva a los pacientes que tienen una condición grave que pone en riesgo su vida. La medicina intensiva se basa en dar soporte vital o soporte a sistemas orgánicos, para lo que se supervisa y monitoriza continuamente al paciente para seguir su evolución (Medicina intensiva, 2021). Las camas UCI son mucho menos usadas que las PACU, pero los pacientes que las necesitan suelen permanecer en estas entre 12 y 36 horas.

En condiciones normales de funcionamiento, en las que el hospital no se encuentra saturado, la especialidad cuenta aproximadamente con un 200% de camas UCI respecto a la cantidad de quirófanos, y con un 167% de camas PACU, también respecto a la cantidad de quirófanos. Durante algunos momentos de la pandemia, este número se ha visto reducido y es objeto de análisis en este estudio.

Previamente a la programación de los quirófanos se ha llevado a cabo una consulta con cada paciente de la lista de espera. El encargado de esta consulta es el que decide qué cirujano será el que tiene que operar al paciente, teniendo en cuenta los problemas que el paciente presenta y las habilidades de los cirujanos disponibles. Teniendo disponibles datos históricos de operaciones similares a las que los pacientes se van a someter, este encargado puede hacer una estimación del tiempo que se va a tardar en la cirugía (este tiempo se representa como duracion_operacion), de si el paciente va a necesitar recuperarse en alguna cama UCI o PACU (si esto ocurre, la representación es NC=1, para la necesidad de PACU o 2 para la necesidad de UCI), y de cuánto tiempo tiene que permanecer en la cama (dur_post_operacion). Según la gravedad del paciente se va a establecer una prioridad para el mismo. Además, conforme a la prioridad que se le ha asignado se le va a dar una fecha límite para la cirugía, que se tratará de no superar. Una vez el paciente está listo, se establece una fecha de lanzamiento, a partir de la cual puede ser operado.

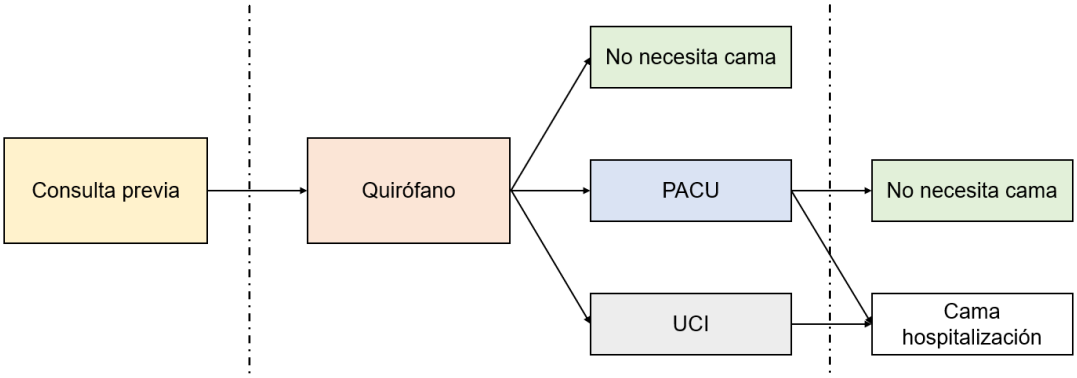

Figura 4. Flujo de pacientes (Elaboración propia)

El presente trabajo recoge la programación del calendario de operaciones y el quirófano en el que las mismas van a suceder durante los 5 días laborales de la semana (representados por D). En la Figura 4 se presenta un cuadro para entender el punto en el que se enmarca el problema. El problema integrado que se aborda en este trabajo consiste en la programación de las operaciones a pacientes en quirófanos, y su posterior etapa de postoperatorio, si lo necesitasen, en camas PACU o UCI (ver elementos en Figura 4 acotados por las líneas discontinuas). Como se puede observar, este problema recibe entonces información de la consulta previa. Con esa información, la estructura de la unidad, y las características específicas de los pacientes se realiza la asignación de pacientes a los quirófanos, teniendo en consideración que no pueden quedar pacientes con necesidad de camas PACU o UCI sin cama o sin poder permanecer en el quirófano después de la operación. Cuando los pacientes terminan de ser operados, o bien se les da el alta, que es el caso en el que los pacientes no necesitan una cama, o bien son atendidos en una cama PACU o UCI. Todos los pacientes que salen de la cama UCI pasan por una cama de hospitalización luego, mientras que solo algunos de los pacientes de las camas PACU necesitan luego la atención en la cama de hospitalización. Sin embargo, lo que ocurra después del uso de las camas UCI o PACU queda fuera del objetivo de este trabajo.

2.1 Restricciones del problema

En cuanto a las restricciones del problema de estudio y comenzando por los cirujanos, éstos no pueden operar siempre, por lo que tienen asignados unos turnos en la semana. Así, cuando se quiera alojar la operación del paciente en un quirófano hay que revisar previamente que el cirujano asociado al mismo tenga el turno disponible, y no tenga otra operación ya programada a la misma hora. Como es comprensible, un solo cirujano solo puede operar a un paciente al mismo tiempo, y en un quirófano solo puede haber un paciente al mismo tiempo. También el paciente no puede ser incluido en calendario hasta que la fecha en la que podría estar disponible (denotado por fecha de lanzamiento) no se alcance. Como máximo los pacientes entran en el calendario el último día considerado en el horizonte temporal. Si no se ha podido incluir al paciente en ese momento, se queda en la lista de espera hasta que se programe la siguiente semana de cirugías.

Las camas UCI y PACU, necesarias para muchos pacientes después de las operaciones para recuperarse, también van a suponer una restricción para la programación de las cirugías. En este trabajo se aborda el problema de la restricción de las camas de tres formas diferentes:

- Las camas se integran en la decisión: no se puede operar a ningún paciente si no se está seguro de que va a existir una cama libre en el momento en el que el paciente acabe de ser operado.

- Las camas no se integran en la decisión: se decide el calendario y los horarios sin contar con la disponibilidad de las camas. A posteriori, el resultado se propaga a las camas. Si dos pacientes tienen que usar una cama al mismo tiempo, la secuencia será inválida.

- En una de las versiones de las camas separadas, la condición es que todos los pacientes, al salir de la operación, deben tener una cama disponible obligatoriamente.

- En la otra versión, al salir de la operación los pacientes que necesiten una cama pueden quedarse en el quirófano si las camas están todas ocupadas. Esto es posible siempre que ningún paciente necesite el quirófano durante ese tiempo. Los quirófanos están totalmente equipados, por lo que no hay problemas en dar a los pacientes los tratamientos que requieren en los mismos.

2.2 Objetivo

El objetivo de la programación es crear una lista de pacientes para la entrada de estos en quirófanos. Se trata de que esta lista de pacientes esté ordenada de forma que se minimice la suma de la tardanza de los pacientes ponderada, $FO = \sum_{i=1}^{P} \left(\frac{T_i}{D} - Op_i\right) * Pr_i$

En el primer término del sumatorio, el primer sumando divide la tardanza del paciente *i* (T_i) entre el horizonte temporal del problema. Aunque el anterior es el objetivo principal, podría ocurrir que muy poca cantidad de pacientes (*P*) fuesen incluidos en los quirófanos y se obtuviese un buen resultado. Para evitar ese problema, se integra en la función objetivo una variable dependiente de si el paciente pudiera ser operado en el horizonte temporal del trabajo (Op_i). Esta variable sería igualmente ponderada según la prioridad del paciente (Pr_i). Por supuesto, se trata de obtener la mejor secuencia para estos objetivos y cumpliendo con todas las restricciones del problema.

La tardanza de un paciente es el tiempo que pasa entre la fecha de vencimiento, fecha antes de la cual debería haber sido operado, y el momento en el que verdaderamente es operado. El concepto de tardanza ponderada, en este trabajo, se refiere a dividir la tardanza del paciente entre el horizonte temporal que se está estudiando, que en este caso es de 5 días.

Como se ha explicado en el punto anterior, en el código se han incluido diferentes formas de trabajar respecto a la restricción de las camas, estas modificaciones se encuentran tanto en la función que decodifica las secuencias para crear la matriz con información acerca de las operaciones y en el algoritmo iterativo. Entonces uno de los objetivos del trabajo es encontrar cuál de las diferentes versiones soluciona mejor el problema de programación que se aborda, si existiese alguna mejor que otra.

En todos los algoritmos de decodificación del programa se emplea una solución matricial que guarda, las entradas de los pacientes en quirófanos por orden. En la matriz se guarda el paciente, el quirófano al que entra, el turno en el que lo hace, la hora a la que se empieza a operar, y a la hora que acaba. En lo que concierne a la función objetivo, esta matriz se utiliza para el cálculo inicial de la tardanza, ya que contiene la información de los turnos, y eso hace fácilmente extraíble el día de operación. En el dibujo descriptivo de esta matriz que se muestra en la Figura 5, las filas representan a las entradas de pacientes en los quirófanos.

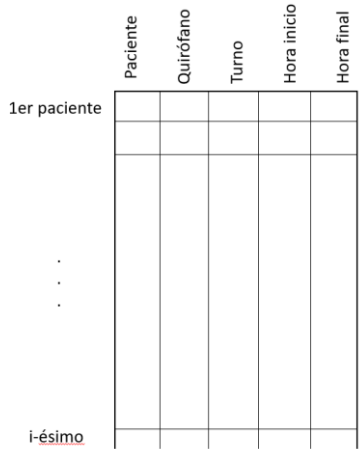

Figura 5. Matriz de entrada de pacientes
(Elaboración propia)

3. Metodología

"Ninguna pérdida debe sernos más sensible que la del tiempo, puesto que es irreparable"

*- **Zenón de Citio***

Como se ha dicho anteriormente el objetivo del problema es el de ofrecer una ordenación de la lista de pacientes en espera que minimice la tardanza ponderada y maximice la cantidad de pacientes operados en el horizonte temporal.

Para simular la realidad del hospital, se han generado toda una serie de datos acerca de los pacientes y los cirujanos (ver sección 4). Se crea una secuencia inicial de pacientes empleando la regla de despacho LPT. Esta función alimenta de datos a un algoritmo iterativo codicioso, que es el encargado de crear secuencias, buscar su óptimo local, y quedarse con la secuencia que mejor resultado ha ofrecido en la función objetivo (ver detalle en Sección 2.2). Dentro del algoritmo, cabe resaltar la función de decodificación (ver Sección 3.1), que es la función que recibe una secuencia y la transforma en un calendario del que el resultado es la matriz de la que se ha hablado previamente, y la función objetivo.

Se crean diferentes algoritmos iterativos codiciosos empleando todas las combinaciones posibles entre las funciones de decodificación desarrolladas (ver Sección 3.1) y las formas de tratar los resultados de la función objetivo obtenidos con la secuencia que sale de la función de decodificación. Se crean estos diversos algoritmos para adaptar el problema a las posibles decisiones que la especialidad del hospital podría tomar en caso de que una situación de escasez de camas acontezca y no sea solucionable de forma inmediata.

Cada uno de los pasos que se han seguido y las variaciones entre las diferentes versiones van a ser expuestas a continuación.

3.1 Función de decodificación de las secuencias

Se han desarrollado dos funciones de decodificación diferentes, una para la versión de camas integradas, y otra para las dos versiones de camas separadas. Ahora se van a explicar en mayor profundidad. Cada una de ellas responde a una forma diferente de generar un problema completo a partir de la secuencia de trabajos.

En todas ellas hay que tener en cuenta la secuencia temporal que siguen los distintos recursos del problema. En los quirófanos, el tiempo se expresa en turnos y horas, por lo que cada paciente es asignado a un quirófano en un turno y a una hora concreta. Las horas se reinician cada vez que se cambia de turno, los turnos de mañana son de 6 horas y los de tarde de 4 horas. Las camas siguen una configuración temporal distinta. El tiempo durante el cual un paciente necesita usar una cama PACU o UCI viene dado en horas totales, no en turnos y horas como los quirófanos, por lo que, conociendo el momento en el que el paciente se operaría en turno y hora, hay que traducir las horas de las camas para poder estimar en qué momento la cama estará libre de nuevo para albergar a otro paciente de la lista. En la Figura 6 se pone de manifiesto este hecho, el paciente comienza a operarse en el turno 0 (correspondiente al primer turno de mañana) a la hora 0:00, y termina de operarse en el turno 0 a la hora 03:00. Para la cama en la que entra el paciente, este paciente está entrando a las 11:30, permanece en la cama 12 horas y sale a las 23:30. La cama vuelve a estar disponible para la entrada de otros pacientes en el turno 2 (turno de mañana del día siguiente) a la hora 0:00.

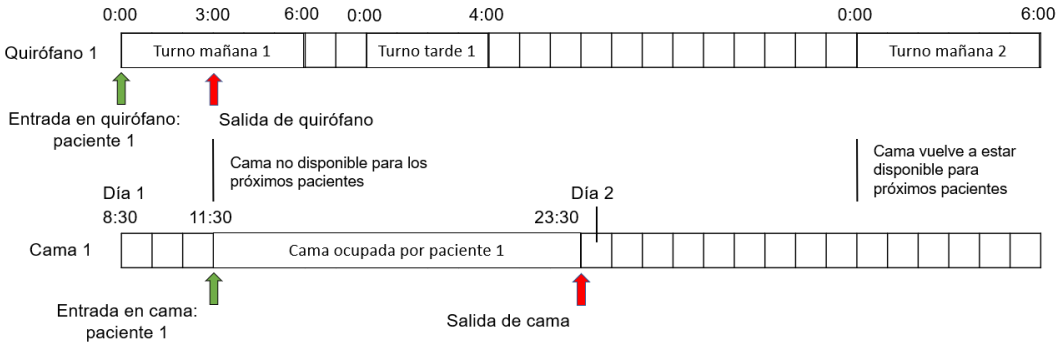

Figura 6. Representación de la divergencia de secuencias temporales (Elaboración propia)

Una de las restricciones del problema es que la operación no puede ser llevada a cabo si se acabara después de la hora máxima establecida para el turno, es decir, aunque la operación se pueda iniciar dentro de las restricciones temporales, si no acaba también dentro de las mismas esa operación no se añade al calendario en ese turno.

El procedimiento de decodificación engloba a todos los pacientes listados, con lo que se estudian los pacientes por el orden en el que se encuentran en la lista, buscando en qué momento pueden entrar estos pacientes en los diferentes quirófanos. Cuando son conocidos los momentos en los que podría ser operado, tanto el turno (T), como la hora dentro del turno (H), los tiempos que han resultado son comparados y la función elige el tiempo que permite la entrada al paciente lo antes posible (TiempoT[quirofano] y TiempoH[quirofano]). En caso de empate, elige el quirófano que más tarde se haya estudiado. Durante el estudio de los posibles momentos de entrada del paciente en cada uno de los quirófanos, los tiempos de entrada en cada uno de los quirófanos se expresan con TiempoTAux[quirofano] y TiempoHAux[quirofano] de forma provisional hasta que se comparen los tiempos de los distintos quirófanos y se decida por el mejor.

En el momento en el que un paciente no pueda entrar en ninguno de los quirófanos disponibles ($Op_i = 0$), la función acaba y el último paciente que se estaba estudiando es el primero en quedar fuera del calendario. Todos los pacientes que siguen a este también quedan fuera.

Respecto a las camas, estas solo se tienen en cuenta en la función de decodificación en su versión de camas integradas, los tiempos de las camas se representan por TiempoTCama y TiempoHCama. Su dinámica se explica en la Sección 3.1.1.

La salida de la función es la matriz (*M*), ya explicada en el punto 2.2. Esta matriz es vital, por razones obvias, para el equipo del hospital, y también lo es tanto para las funciones relacionadas con la decodificación, como para el algoritmo en el que la función se anida. Se detalla el pseudocódigo de la función en la Figura 7.

```
OUTPUTS M
Begin
    for i=0 to P do:
        for j=0 to Q do:
            for x=T to d*2 do:
                TiempoTAux[j], TiempoHAux[j] := T, H
        if Op_i = 1 do:
            for j=0 to Q do:
                if j=0 do:
                    j* := j
                    TiempoT[j*] := TiempoTAux[j*]
                    TiempoH[j*] := TiempoHAux[j*]
                else if TiempoTAux[j] < TiempoT[j*] or (TiempoTAux[j] = TiempoT[j*]
                and TiempoHAux[j] <= TiempoH[j*]) do:
                    j* := j
                    TiempoT[j*] := TiempoTAux[j*]
                    TiempoH[j*] := TiempoHAux[j*]
            M := i, j*, TiempoT[j*], TiempoH[j*], TiempoH[j*] + duracion_operación

            if NC_i = 1 or NC_i = 2 do: # Solo para la versión Camas Integradas
                TiempoTCama, TiempoHCama := TiempoT[j*], TiempoH[j*] +
                dur_post_operacion
end
```

Figura 7. Pseudocódigo funciones de decodificación (Elaboración propia)

A continuación, en la Sección 3.1.1 se explica en detalle el funcionamiento de la función de decodificación que integra las camas en el proceso de decisión para establecer el turno y la hora en la que el paciente se puede operar. En la Sección 3.1.2 se desglosa la función que no considera las camas en el momento de asignar los turnos y horas de los quirófanos a los pacientes. Esta función de decodificación se usa en compañía de otra función, que cuenta con dos versiones distintas, que se encarga de comprobar si hay camas para los pacientes que salen de los quirófanos.

3.1.1 Camas integradas

En esta versión de la función de decodificación se exige que haya una cama disponible cuando el paciente vaya a salir de la operación, por lo que nunca se tendrán pacientes en espera entre la fase de operaciones y la fase de las camas. Es una aplicación lógica del problema que se aborda, pues un paciente que requiere ser tratado de forma intensiva después de una operación nunca puede quedar sin esa atención, por el riesgo que entrañaría para la salud del paciente. En el código esto se exige no introduciendo el paciente hasta que el momento en el que su operación acabe coincida con el momento en el que una de las camas quede libre.

A priori, sería la solución más lógica si verdaderamente se va a tener un problema en las camas y éstas se convierten en un cuello de botella. Pues lo que se vería es que acaba teniendo resultados similares a los que se tendrían en el caso de que no hubiese nunca problemas de camas y simplemente se propagase la solución de los quirófanos a las camas.

Antes de que se compruebe si en un quirófano en un turno y hora determinada se puede operar al paciente en estudio, se comprueba si este necesita de una atención en cama posterior. Si no la necesita, la función solo comprueba la disponibilidad de quirófano y cirujano asociado. Si la necesita, la función comprueba si en el turno en el que se encuentra hay alguna cama libre, al menos que se quede libre durante algún momento del turno. En caso de que no haya libre

ninguna cama del tipo que necesita, ese turno no se estudia, se pasa al siguiente y se vuelve a intentar.

Una vez que ya se conoce cuál es el mejor quirófano para operar al paciente, se actualiza el tiempo de la cama según el tiempo que el paciente tenga que permanecer en ella. Quedando fijado el nuevo tiempo al momento en el que se queda libre, puesto en turnos y horas.

3.1.2 Camas separadas

En las versiones de camas separadas, se distinguen dos partes: la primera es la función de decodificación, y la segunda es una función especial que comprueba la validez de la secuencia que se ha decodificado. La función de decodificación de ambas camas es idéntica. En ella se suprime la parte en la que se tiene que comprobar que existan camas para incluir al paciente, haciendo la función más sencilla, pues solo hay que tener en cuenta los turnos de los quirófanos y los turnos en los que el cirujano asociado al paciente trabaja. Los tiempos de los quirófanos y los cirujanos son los únicos que hay que actualizar en esta función. El pseudocódigo de esta función se representó en la Figura 7.

Ambas versiones de camas separadas responderían a lo que sería la solución normal del problema sin la restricción de las camas que se está estudiando el trabajo. La primera versión es la más pura, en la que, para que la secuencia tenga validez, la cama tiene que estar siempre disponible nada más el paciente termine de operarse. Para hacer las comprobaciones, se toma cada uno de los pacientes que han entrado en el calendario de operaciones, y, si el paciente necesita una cama, se comprueba que exista dicha cama en el momento en el que este paciente sale del quirófano. Si esta cama no existiese, es decir, no estuviese libre en el momento adecuado, la secuencia sería inválida. Si el paciente no necesita cama, se continúa estudiando el siguiente paciente. La Figura 8 contiene el pseudocódigo de esta primera versión de la función.

La segunda versión permite algo más de laxitud, siendo posible que los pacientes permanezcan en los quirófanos después de operarse si no hay camas disponibles del tipo necesario en el momento de culminación de la operación. Esto es posible siempre que, durante ese tiempo en el que el paciente tiene que permanecer en el quirófano para el posoperatorio, no haya otro paciente que tenga que entrar en el quirófano (que no exista otro paciente luego se representa con existe_ProxP=0). Si existe ese paciente en el calendario, y sigue sin haber camas libres en el momento en el que tiene que entrar, la secuencia queda invalidada. El pseudocódigo de esta función se encuentra en la Figura 9.

La matriz (*M*) contiene la información que las funciones de ambas versiones necesitan para comprobar la validez de la secuencia, pues es la que recoge los momentos en los que los pacientes que han conseguido entrar dentro de la programación con el horizonte temporal definido entran y salen de los quirófanos. En ambas funciones la información sobre la finalización de los pacientes en quirófano es imprescindible para la asignación de la camas (se representa como $TiemposSalida_{paciente}$). En la segunda versión, también es importante las horas de entrada en quirófanos, pues son el límite de tiempo que tienen los pacientes que se han tenido que quedar dentro del quirófano en la etapa de posoperatorio.

```
OUTPUTS Π^val
Begin
    Π^val := True
    for i=0 to P do:
        if i en M do:
            if NC_i = 0 do:
                continue
            else if NC_i = 0 do:
                flag^PA := 0
                for c=0 to C do:
                    if TiemposCama[c] <= TiemposSalida_i do:
                        flag^PA := 1
                        break
                if flag^PA = 0 do:
                    Π^val = False
                    break
        else do:
            break
end
```

Figura 8. Pseudocódigo de la función de comprobación de camas separadas versión 1 (Elaboración propia)

```
OUTPUTS Π^val
Begin
    Π^val := True
    for i=0 to P do:
        if i en M do:
            if NC_i = 0 do:
                continue
            else if NC_i ! = 0 do:
                flag^PA := 0
                for c=0 to C do:
                    if TiempoTCama[c], TiempoHCama[c] <= TiempposSalida_i do:
                        flag^PA := 1
                        break
                if flag^PA = 0 do:
                    if existe_ProxP = 1 do:
                        for r = 0 to C do:
                            flag^T, flag^H := TiempoTCama[c], TiempoHCama[c]
                        if flag^T, flag^H <= TiempposSalida_i do:
                            Actualización TiempoTCama, TiempoHCama
                        else do:
                            Π^val := False
                            break
        else do:
            break
end
```

Figura 9. Pseudocódigo de la función de comprobación de camas separadas versión 2 (Elaboración propia)

3.2 Algoritmo Iterativo Codicioso

Se proponen tres variantes de la metaheurística *iterated local search*. La primera variante de la metaheurística acepta todos los resultados obtenidos al invocarse la función objetivo. Aplica una primera fase en la que se obtiene el resultado para la primera secuencia, que se aporta como una entrada al algoritmo. En la segunda fase se destruye y vuelve a construir la secuencia, y en la tercera se realiza una búsqueda local en la que se estudian los vecinos de la secuencia buscando el que aporte la mejor solución a la función objetivo. La primera variante se encuentra en la Sección 3.2.1. La segunda y tercera variantes solo se emplean con la función de decodificación de camas separadas, pues son las funciones que pueden devolver secuencias no válidas. La segunda versión tiene las mismas fases que la primera, pero cuando estudia el resultado obtenido de la primera secuencia, comprueba la validez del resultado, si no es válido, incluye una fase de destrucción y construcción de la secuencia inicial. Luego, en la fase de búsqueda local incluye una penalización en la función objetivo a las secuencias que no aporten resultados válidos. Esta variante se encuentra en la Sección 3.2.2. Por último, la tercera versión difiere de la primera en la fase de búsqueda local, donde replica la fase de destrucción y construcción de la secuencia. Esta variante final se encuentra en la Sección 3.2.3.

3.2.1 Algoritmo iterativo codicioso. Variante 1, ILS_1

El algoritmo, que se basa en la *Iterated Greedy* propuesta en Ruiz & Stützle, (2007) se inicia siempre con la regla de despacho LPT (*Longest Processing Time*), por lo que la secuencia inicial está ordenada tal que los pacientes que necesitan más tiempo en el quirófano son los primeros en entrar a estos.

El algoritmo se puede ver a grandes rasgos divido en cuatro partes claras. La primera parte es

la obtención de la primera solución que proviene de decodificar la secuencia original y calcular su función objetivo. Esta primera parte solo se realiza en una ocasión, cuando se entra al algoritmo. La segunda parte se inicia con la destrucción y construcción de la secuencia, esto es, se eliminan 4 trabajos de la secuencia y se introducen en posiciones aleatorias con el objetivo de perturbar la secuencia y salir del "valle" en el que se puede encontrar la solución para explorar otras posibles soluciones locales. Con "valle" se hace referencia a un entorno de posibles soluciones que tiene como mejor solución un óptimo local. En el algoritmo habrá diversos valles, cada uno de ellos con su propio óptimo local.

Una vez reconstruida la secuencia, se entra en la fase de búsqueda local, en la que se comprueban los vecinos de la secuencia en busca del óptimo local del "valle" en el que se encuentra la solución. Estos vecinos se han determinado mediante la extracción secuencial de los pacientes contenidos en las posiciones y su inserción en otras posiciones, de forma que queden probadas todas las combinaciones de posiciones posibles en la secuencia. Esto significa que una secuencia con *n* trabajos cuenta con $(n-1)^2$ vecinos. Se consideran las posiciones y no los pacientes, ya que se emplea la regla *first-best improvement* en la búsqueda local, y esta regla impone que, si uno de los vecinos mejora en un momento a la secuencia original, se toma este vecino como secuencia sobre la que seguir calculando los demás. Fijando la atención en las posiciones, se sigue el orden de los vecinos sin que afecte el cambio en el orden de los pacientes de la secuencia.

Por último, se realiza un recocido simulado (*Simulated Annealing*) para decidir si la secuencia que sale de la búsqueda local se emplea en la siguiente ronda de búsqueda. Si el resultado final de la fase de búsqueda local mejora el resultado de la secuencia que se estaba usando antes para la misma, la nueva secuencia pasa a ser la secuencia en uso, y el resultado final es el nuevo resultado de búsqueda local (*LS* o *local search*). Si además mejora el mejor resultado que se ha obtenido hasta el momento, se establece como la mejor secuencia y su resultado como el mejor resultado hasta el momento. Si no se mejorase el resultado de la búsqueda local, se puede coger la secuencia como nueva secuencia para la próxima ronda si se cumple la condición de que un número aleatorio, calculado al inicio del algoritmo, sea menor o igual a $e^{-(resultado\_final - resultado\_LS)/temperatura}$. Este paso, dado en el recocido simulado se realiza con la intención de aumentar la perturbación de la secuencia, que puede ser insuficiente solo con la fase de destrucción y construcción que se realiza en el inicio, y se puedan estudiar mayor cantidad de máximos locales.

El algoritmo acaba cuando se cumpla el criterio de parada, que se ha obtenido de la documentación y es el resultado de multiplicar el número de pacientes (*m*) por el número de quirófanos (*Q*) por el número de días contemplados en el horizonte temporal (*d*), dividir el resultado entre 2 y multiplicarlo por 25. La fórmula usada como criterio de parada se ha obtenido de Molina (2016), siendo el multiplicador del 25 uno de los dos valores que da a la variable *v*, una de las dos posibilidades que se ofrecen en la tesis, $T = \left(\frac{m \cdot Q \cdot d}{2} \cdot 25\right)\ milisegundos$. Es decir, el algoritmo acaba cuando el momento actual (*Now*) alcance el tiempoFinal = T + tiempoInicial (que es el momento en el que inicia el algoritmo).

Esta primera variante del algoritmo siempre va a utilizar como función de decodificación la de camas integradas (expuesta en la Sección 3.1.1). No funcionaría correctamente para la función de camas separadas porque si fuese esa la decodificación empleada, tendría que venir acompañada de una función que comprobase las camas luego, abriendo la posibilidad de obtener secuencias no válidas, y en esta variante del algoritmo no se ha implementado ninguna metodología para reaccionar ante secuencias no válidas.

3.2.2 Algoritmo iterativo codicioso. Variante 2, ILS_2

La segunda variante del algoritmo también inicia con la regla LPT. Cuando la secuencia pasa por la función de decodificación de camas separadas, esta función devuelve la solución matricial. Esta solución matricial es la entrada a la función de camas separadas (pudiendo usarse la versión 1 y 2), que comprueba si el calendario contenido en la solución puede ser cumplido al considerar la restricción de camas. Si el resultado de dicha función es que la secuencia no es válida, esta secuencia pasa por una fase de destrucción y construcción como la descrita en la primera variante del algoritmo. La fase no acaba hasta que la secuencia generada pase por la función de comprobación de las camas de nuevo y esta indique que la secuencia es válida.

Ya con una secuencia válida y un primer resultado de la función objetivo, la primera fase del algoritmo ha culminado, entrando así en las tres fases que se van a repetir en el algoritmo hasta

que se alcance el criterio de parada. La primera de estas tres fases es la fase de destrucción y construcción, en la que se genera una nueva secuencia. Si esta secuencia generada no es válida, se penaliza su resultado de la función objetivo, pero continúa el algoritmo con la fase de búsqueda local. Dentro de la búsqueda local, para cada secuencia que se obtiene se comprueba la validez, y si no es válida, se le impone una enorme penalización en la función objetivo, así se asegura que nunca se va a elegir como mejora una secuencia que no es válida.

Por último, el *Simulated Annealing* y el criterio de parada del algoritmo son factores que se mantienen idénticos a los de la primera variante del algoritmo.

En la experimentación esta segunda variable se empleará en dos ocasiones, la primera usando como función de comprobación de camas la función de camas separadas versión 1, y la segunda usando la función camas separadas versión 2. No varía la función de decodificación que siempre será la de camas separadas.

3.2.3 Algoritmo iterativo codicioso. Variante 3, ILS_3

La tercera variante mantiene el uso de la secuencia LPT y la fase de destrucción y construcción que acontece si, al comprobar las restricciones de camas para el resultado obtenido de la secuencia, esta resulta no ser válida. De la misma forma se mantiene la fase posterior de destrucción y construcción con la misma penalización del resultado de la función objetivo para las secuencias no válidas resultantes.

El cambio se produce en la etapa de búsqueda local, en la que, si al comprobar la validez de una secuencia, esta resulta ser no válida, en lugar de incurrir una penalización en el resultado de la función objetivo de la secuencia, esta se vuelve a destruir y construir, exactamente de la misma forma que en todos los procesos de destrucción y construcción anteriores. Esta etapa provoca una nueva perturbación en la secuencia, por lo que en el limitado tiempo que el algoritmo tiene para operar, se podrán estudiar más “valles”. La contrapartida es que no siempre se llegará al óptimo local de estos valles, ya que, si una de las secuencias generadas durante el estudio de los vecinos sale no válida, la perturbación podrá provocar un salto a un valle diferente antes de que la búsqueda local en el valle anterior hubiese acabado.

Para acabar, el *Simultaed Annealing* y el criterio de parada no sufren ninguna variación respecto a las dos variantes anteriores.

Como ocurría con la variante 2, la variante 3 será doblemente usada en la experimentación para poder acoger las dos versiones diferentes de comprobación de disponibilidad de camas. La función de decodificación siempre va a ser la de camas separadas.

Begin

FO^{mejor}, FO^{flag}, FO^{flagLS} := FuncionObjetivo

Π^{mejor}, Π^{flagLS} := Π

while Now <= tiempoFinal **do**:

Π^{flag} := FaseDestrucciónConstrucción(Π^{flagLS})
FO^{flag} := FunciónObjetivo

do:

$flag^{imp}$:= False

FO^{nuevo} := BúsquedaDeVecino

if $FO^{nuevo} < FO^{flag}$ **do**:

$FO^{flag} := FO^{nuevo}$
$\Pi^{flag} := \Pi^{nueva}$
$flag^{imp}$:= True

while $flag^{imp}$ = True;

if $FO^{flag} < FO^{flagLS}$ **do**:

$\Pi^{flagLS} := \Pi^{flag}$
$FO^{flagLS} := FO^{flag}$

if $FO^{flag} < FO^{mejor}$ **do**:

$\Pi^{mejor} := \Pi^{flag}$
$FO^{mejor} := FO^{flag}$

else if numAleat <= $e^{-\frac{FO^{flag}-FO^{flagLS}}{T}}$ **do**:

$\Pi^{flagLS} := \Pi^{flag}$
$FO^{flagLS} := FO^{flag}$

end

Figura 9. Pseudocódigo general del algoritmo iterativo codicioso (Elaboración propia)

3.2.4 Algoritmos propuestos

Después de considerar todas las posibles combinaciones entre algoritmos, funciones de decodificación y funciones de comprobación de las camas, los algoritmos que se han utilizado para la comparación son los siguientes:

- Algoritmo de camas integradas (CI): formado por la primera variante del algoritmo iterativo codicioso que emplea la función de decodificación de camas integradas.
- Algoritmo de camas separadas método 1 versión 1 (CSM1V1): compuesto por la segunda variante del algoritmo iterativo codicioso que emplea la función de decodificación de camas separadas y la función de comprobación de camas en su primera versión. Recapitulando de la Sección 3.1.2, esta versión es la que no permite que los pacientes permanezcan en el quirófano después de ser operados.
- Algoritmo de camas separadas método 1 versión 2 (CSM1V2): integra la función de decodificación de camas separadas y la función de comprobación de camas en su segunda versión en la segunda variante del algoritmo iterativo codicioso. Recapitulando de la Sección 3.1.2, la segunda versión de la función de comprobación de camas es la que permite a los pacientes permanecer en el quirófano siempre que este no sea requerido por otro paciente durante ese mismo tiempo.
- Algoritmo de camas separadas método 2 versión 1 (CSM2V1): hace uso de la tercera versión del algoritmo iterativo codicioso. Dentro del algoritmo se emplean la función de decodificación de camas separadas y la función de comprobación de camas en su primera versión.
- Algoritmo de camas separadas método 2 versión 2 (CSM2V2): decodifica las secuencias con la función de camas separadas y comprueba la validez de dichas secuencias con la segunda versión de la función de comprobación de camas. Todo esto dentro de la tercera variante del algoritmo iterativo codicioso.

Abreviatura algoritmo	Variante de Iterated Greedy	Función de decodificación	Función de comprobación
CI	Variante 1	Camas Integradas	-
CSM1V1	Variante 2	Camas Separadas	Versión 1
CSM1V2	Variante 2	Camas Separadas	Versión 2
CSM2V1	Variante 3	Camas Separadas	Versión 1
CSM2V2	Variante 3	Camas Separadas	Versión 2

Tabla 1. Resumen de los algoritmos propuestos (Elaboración propia)

4. Experimentación

"Una experiencia nunca es un fracaso,
pues siempre viene a demostrar algo"

*- **Thomas Alba Edison***

La experimentación se compone de dos partes, la generación de los datos y la obtención y comparación de resultados. La generación de datos está compuesta por datos fijados y datos variables. Los datos fijados son los que utiliza el programa para generar las diversas instancias. Por lo que estos datos no se generan de nuevo en cada iteración del programa, si no que son fijados al principio de este. Los datos variables se regeneran cada vez que se completa una iteración del programa, es decir, cada vez que se ha pasado por los diferentes algoritmos que se comparan en el programa.

4.1 Generación de los datos

Los datos del problema pueden ser fijos o variables según las propias variables del problema. Entre los datos fijos se encuentran el número de días, fijado a 5, lo que fija al mismo tiempo el número de semanas a 1, el número de turnos que los quirófanos están disponibles y las horas que los quirófanos están disponibles durante dichos turnos.

Otros factores como el número de pacientes listados en espera o el número de cirujanos a disposición son distintos según los valores otorgados a las variables. Estos valores se toman con la literatura contenida en Molina-Pariente, Hans, et al. (2015).

Los cirujanos disponibles se determinan con la expresión: $|S| = \alpha \cdot \frac{\sum_{j \epsilon J} \sum_{h \epsilon H} r_{jh}}{l \cdot a \cdot mds}$

- α: es el factor de control, que varía entre 1.5 y 2.
- r_{jh}: capacidad del quirófano j en el turno h. Este dato también es invariable, pues se mantiene igual siempre al depender de la disponibilidad de los quirófanos y el tiempo que estos permanecen disponibles (datos que se pueden encontrar en la descripción del problema).
- l: es el número de semanas, y como se ha dicho anteriormente, este valor se mantiene fijado a 1.
- mds: es el número máximo de turnos en los que un cirujano puede operar en una semana, va a variar entre 3 y 4.

Para la determinación del número de operaciones en lista de espera, se irán generando una por una hasta que superen un porcentaje del tiempo disponible para operar a pacientes, es decir, del tiempo que los quirófanos permanecen disponibles (β). β varía entorno al 1 y el 1.25.

Además de estas variables se tienen otras dos más específicas del problema en concreto que se estudia en este trabajo, estas variables son la cantidad de camas PACU y la cantidad de camas UCI. Los valores máximos y mínimos de las camas han sido definidos en la Descripción del problema, sin embargo, en la experimentación se toman datos intermedios, de forma que para las camas PACU se toman los valores 2, 3, 4 y 5. Para las camas UCI se toman los valores 2, 4, 5 y 6.

La combinación de los parámetros α, β, mds, cantidad de camas PACU y cantidad de camas UCI generan las diferentes situaciones, y en cada situación se generan 5 instancias. Esto hace un total de 640 iteraciones a las 5 combinaciones de algoritmos que se tienen. Para cada iteración se generan valores como la prioridad de los pacientes, la fecha de lanzamiento, la fecha de vencimiento y la duración de las operaciones. A lo que hay que sumar el cirujano asociado con el paciente y los turnos en los que el cirujano está disponible.

Para la prioridad se usa la fórmula: $a \cdot mp^* + (1 - a) \cdot dwl^*$. Siendo a un factor empleado para la determinación de la importancia de cada indicador, que se va a mantener en 0.5 por establecer que poseen la misma importancia. El peso médico normalizado se obtiene con una distribución uniforme discreta [1, 5]/5 (mp^*). Por último, el número de días en lista de espera normalizado

(dwl^*), que para normalizarse es dividido por el tiempo máximo antes del tratamiento ($MTBT$).

El cálculo de la fecha de lanzamiento se hace de forma aleatoria entre el primer y último turno del horizonte temporal. La fecha de vencimiento sigue la expresión: $(MTBT - dwl) \cdot 2$. En la que $MTBT$, como se ha dicho antes, es el tiempo máximo antes del tratamiento, y depende de la urgencia del paciente. Se encuentra definido por los Servicios Nacionales de Atención Médica teniendo en cuenta criterios clínicos. Emulando al Servicio Nacional Español, la variable tomará valores aleatorios entre 45, 180 y 360. dwl, el número de días en lista de espera se genera siguiendo una distribución uniforme discreta entre [1, MTBT – 1].

Se estima la duración de cada operación de los pacientes (en horas) con el uso de una distribución log-normal con dos parámetros: duración esperada (μ), que se genera aleatoriamente entre 1, 2, 3, 4 (Marcon et al., 2003), y desviación estándar (σ) determinada aleatoriamente entre los valores $[0.1\mu, 0.5\mu]$. Dentro de la duración se encuentran contenido el tiempo de la operación del paciente, el tiempo necesario para la preparación del quirófano para la operación, y el tiempo de limpieza del mismo una vez se acaba.

La asignación del cirujano asociado al paciente se hace aleatoriamente, al ser también una decisión propia de la consulta previa, que no entra en los límites del trabajo. Por último, para decidir los turnos en los que cada cirujano puede operar se aplicará un procedimiento de dos pasos. El primero de ellos, se asigna aleatoriamente un cirujano por cada quirófano disponible en cada turno. Estos cirujanos pueden volver a asignarse si no han superado el número máximo de turnos en los que pueden operar. El segundo paso, consiste en el reparto de los turnos que los cirujanos que no hayan llegado al máximo siguen teniendo para operar. La única restricción lógica es que un cirujano no puede volver a asignarse a un turno al que ya ha sido asignado. Con este procedimiento en dos pasos se consigue que ningún quirófano vaya a quedar inactivo y que todos los cirujanos empleen todos los turnos de los que disponen en la semana que se toma como horizonte temporal.

4.2 Obtención y comparación de resultados

No existe una única forma de comparar los resultados de los algoritmos entre sí, por lo que, para tomar un modelo de referencia, se ha usado el artículo (Molina-Pariente et al., 2015). Aquí se utilizan tanto el tiempo de CPU necesario para resolver una instancia como el RPD (*Relative Percentage Deviation*). Como los algoritmos que se han implementado en este modelo están limitados por una variable de tiempo, no tiene demasiado sentido utilizar el tiempo de CPU, a si que solo se usará el RPD como comparativa.

$RPD = \frac{Best_{sol} - Current_{sol}}{Best_{sol}}$. $Best_{sol}$ es la mejor solución obtenida para una instancia, sin importar el algoritmo que haya proporcionado la solución. $Current_{sol}$ es la solución del algoritmo que se está estudiando en la instancia actual. El RPD se calcula para cada algoritmo en cada instancia. Así el algoritmo que obtiene el mejor resultado en la instancia tendrá un RPD de 0.

Para hacer más sencillo el cálculo de los RPD se ha cambiado el signo de las funciones objetivos, ahora la mejor solución es la que obtiene el máximo valor en la función objetivo. Esto es viable porque no hay valores para la función objetivo que sean superiores a 0. Con este cambio realizado, se calculan los RPD de todos los algoritmos en todas las instancias.

Con el cálculo realizado, es posible empezar a comprar algoritmos entre sí. Empleamos el ARPD (*Average Relative Percentage Deviation*) para este objetivo, $ARPD_{alg} = \sum_{i=1}^{I} \frac{RPD_{i,alg}}{I}$. $ARPD_{alg}$ es la media de los RPD de un algoritmo para una cantidad de instancias *I* estudiadas. $RPD_{i,alg}$ es el resultado del algoritmo estudiado en una instancia. Cuanto más cercano a 0 sea el ARPD de un algoritmo, mejor es el mismo comparado con los demás.

Ya que el motivo del estudio del trabajo ha sido el problema en la escasez de las camas, se empieza por estudiar cómo afecta el cambio en el número de camas en los diferentes algoritmos. Para realizar dicho estudio, se agrupan las instancias de la experimentación según las combinaciones posibles de las variables camas PACU y camas UCI. Cada combinación de las variables agrupa 200 instancias. En la Tabla 1 se presentan los valores obtenidos en el ARPD al considerar los RPD de las instancias contenidas en las combinaciones. En la Figura 10, se representan estos valores con puntos para buscar tendencias y aclarar los valores obtenidos.

PACU	UCI	CI	CSM1V1	CSM1V2	CSM2V1	CSM2V2
2	2	0,0754	0,0380	0,0309	0,0312	0,0159
2	4	0,0715	0,0310	0,0224	0,0311	0,0133
2	5	0,0890	0,0352	0,0198	0,0355	0,0174
2	6	0,0835	0,0367	0,0246	0,0335	0,0194
3	2	0,0502	0,0255	0,0264	0,0301	0,0184
3	4	0,0702	0,0291	0,0265	0,0245	0,0166
3	5	0,0618	0,0288	0,0289	0,0269	0,0277
3	6	0,0729	0,0267	0,0254	0,0243	0,0209
4	2	0,0505	0,0346	0,0277	0,0264	0,0258
4	4	0,0401	0,0334	0,0349	0,0326	0,0352
4	5	0,0510	0,0252	0,0305	0,0281	0,0317
4	6	0,0523	0,0220	0,0254	0,0257	0,0323
5	2	0,0357	0,0200	0,0278	0,0290	0,0274
5	4	0,0221	0,0350	0,0322	0,0338	0,0370
5	5	0,0336	0,0351	0,0335	0,0352	0,0257
5	6	0,0317	0,0322	0,0264	0,0373	0,0320

Tabla 2. Soluciones del ARPD para las combinaciones de PACU y UCI (Elaboración propia)

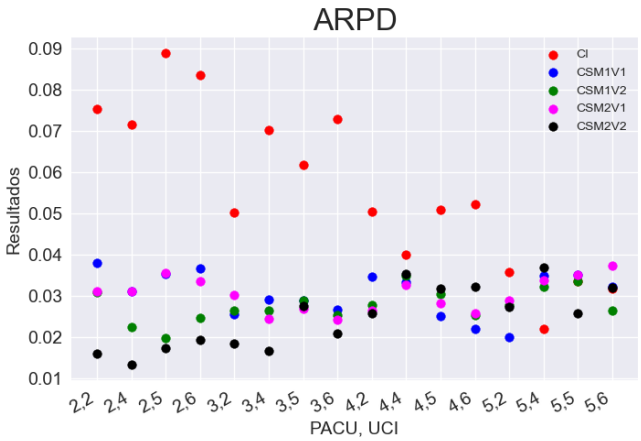

Figura 10. Representación del ARPD de los algoritmos según la cantidad de camas PACU y UCI (Elaboración propia)

En la Tabla 1, así como en las posteriores, el color verde se emplea para hacer referencia al mejor resultado del ARPD para una combinación concreta de variables, el turquesa para los segundos mejores resultados, el amarillo, el naranja y el rojo representan, respectivamente, el tercer mejor valor, el segundo peor resultado, y el peor resultado.

La figura anterior (Figura 10) muestra un resultado claro, el algoritmo de camas integradas no ofrece buenas soluciones en casi ningún caso. Mientras el resto de los algoritmos son más o menos equiparables en sus resultados, encontrándose comprendidas en un menor espacio del gráfico la mayoría de sus soluciones. En contra de lo que se podría haber pensado en un principio, el algoritmo CI funciona mejor cuando la cantidad de camas aumenta. En el gráfico es claro como la media de sus resultados baja con la subida de la cantidad de camas, siendo más influyentes las PACU que las UCI. Estos resultados se ha consolidado con más pruebas que serán expuestas a continuación. El análisis del por qué puede ocurrir esto será hecho al final, después de que se hayan podido agrupar todas las conclusiones derivadas de los distintos estudios.

Analizando en la figura otros algoritmos, es evidente que en la primera mitad de la tabla el algoritmo que mejor ha funcionado es el de CSM2V2, y según van aumentando el número de camas PACU, el algoritmo va empeorando e igualándose con todos los demás. El resto de los algoritmos se han comportado bastante planos, sin grandes cambios de tendencia. Por lo que en cómputo general el algoritmo que mejores resultados ofrecería sería el CSM2V2.

Para el siguiente estudio se han considerado el número de cirujanos y la variable β. El número de cirujanos es producto de las combinaciones de las variables α y mds, por lo que se podría decir que se están estudiando estas dos variables y β en este estudio. Los resultados obtenidos se representan a continuación, en la Tabla 2. Siendo su representación expuesta en la Figura

11.

Beta	N Cirujanos	CI	CSM1V1	CSM1V2	CSM2V1	CSM2V2
1	10	0,0332	0,0335	0,0295	0,0319	0,0287
1	8	0,0526	0,0258	0,0229	0,0297	0,0264
1	13	0,0497	0,0237	0,0206	0,0260	0,0158
1,25	10	0,0657	0,0287	0,0310	0,0287	0,0235
1,25	8	0,0703	0,0361	0,0313	0,0336	0,0221
1,25	13	0,0751	0,0344	0,0259	0,0321	0,0295

Tabla 3. Soluciones del ARPD para las combinaciones de β y cirujanos (Elaboración propia)

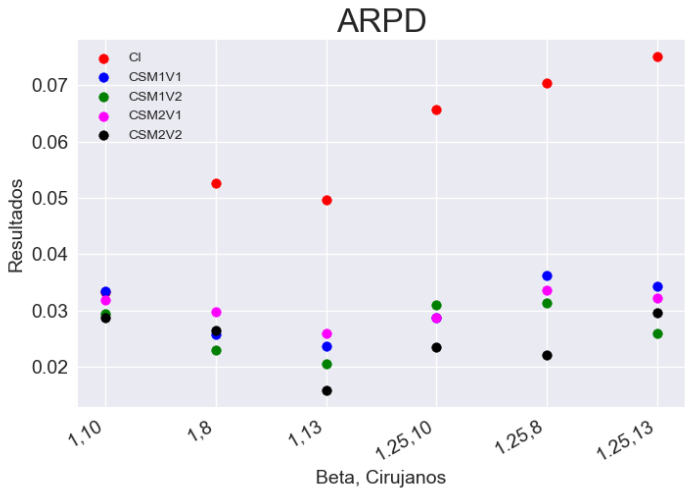

Figura 11. Representación del ARPD de los algoritmos según los valores de β y el número de cirujanos (Elaboración propia)

En este estudio, se observa que el algoritmo CI obtiene buenos resultados en el primer caso cuando la variable β es 1 y los cirujanos 10. Es fácil ver que este algoritmo funciona mejor con 10 cirujanos que con cualquier otra combinación, y que empeora con el crecimiento de la variable β, es decir, con el crecimiento en el número de pacientes. El resto de los algoritmos se mueven en un pequeño rango, sin muchas variaciones. Se vuelve a confirmar que el mejor de los algoritmos es el CSM2V2, siendo el que ofrece mejor resultado en 4 de las 6 combinaciones de variables, seguido del CSM1V2 y luego las variantes de la versión 1.

Dos últimos estudios se han realizado para terminar de confirmar resultados. Uno en el que se consideran las variables β, camas PACU, camas UCI y cantidad de cirujanos, y otro en el que se estudian las variables α, β, mds, camas PACU y camas UCI. Ambos son prácticamente iguales pues, como se ha dicho antes, la cantidad de cirujanos es una buena aproximación a las variables β y mds. Sin embargo, se ha decidido explorar más resultados y ver si había algo que resaltar. Solo se van a mostrar las tablas que contengan alguna información relevante en los resultados del ARPD.

Alpha	Beta	Mds	PACU	UCI	CI	CSM1V1	CSM1V2	CSM2V1	CSM2V2
1,5	1	3	2	2	0,0506	0,0378	0,0497	0,0361	0,0291
1,5	1	3	2	4	0,0170	0,0325	0,0343	0,0326	0,0199
1,5	1	3	2	5	0,0147	0,0313	0,0256	0,0300	0,0255
1,5	1	3	2	6	0,0449	0,0615	0,0135	0,0501	0,0344
1,5	1	3	3	2	0,0330	0,0218	0,0153	0,0305	0,0154
1,5	1	3	3	4	0,0081	0,0316	0,0293	0,0313	0,0227
1,5	1	3	3	5	0,0210	0,0242	0,0297	0,0318	0,0330
1,5	1	3	3	6	0,0749	0,0318	0,0225	0,0223	0,0204
1,5	1	3	4	2	0,0170	0,0733	0,0598	0,0656	0,0637
1,5	1	3	4	4	0	0,0501	0,0537	0,0488	0,0465
1,5	1	3	4	5	0,0088	0,0414	0,0292	0,0175	0,0384
1,5	1	3	4	6	0,0428	0,0268	0,0191	0,0278	0,0404
1,5	1	3	5	2	0,0127	0,0458	0,0464	0,0558	0,0594
1,5	1	3	5	4	0	0,0298	0,0384	0,0467	0,0516
1,5	1	3	5	5	0	0,0336	0,0412	0,0418	0,0355
1,5	1	3	5	6	0,0030	0,0446	0,0423	0,0483	0,0422

Tabla 4. Soluciones parciales del ARPD para las combinaciones de α, β, mds, camas PACU y camas UCI (Elaboración propia)

En las soluciones del ARPD lo más relevante es esta Tabla 3 que deja ver que para la combinación de $\alpha = 1.5, \beta = 1\ y\ mds = 3$, el algoritmo CI obtiene resultados bastante buenos. Esto para el caso en el que la cantidad de pacientes ronda los 55 pacientes, y el número de cirujanos los 10, siendo prácticamente independiente de la cantidad de camas que existan en el modelo.

Resultados ARPD Global				
CI	CSM1V1	CSM1V2	CSM2V1	CSM2V2
0,0557059	0,03053038	0,02770425	0,03033043	0,02478223

Tabla 5. ARPD global (Elaboración propia)

Revisando el ARPD global que se ha obtenido en la Tabla 6, los resultados de los casos particulares parecen ser completamente plausibles. El mejor algoritmo ha sido el CSM2V2, seguido de su misma versión, pero con el método 1. El CSM2V1 ha sido ligeramente mejor que el CSM1V1, y a un par de distancia se encuentra el CI.

5. Conclusiones

"La vida es el arte de sacar conclusiones suficientes a partir de datos insuficientes"

- ***Samuel Butler***

Se han compuesto 5 algoritmos para tratar un posible problema en la programación de operaciones por una escasez en las camas PACU y UCI. Todos los algoritmos han sido iterativos codiciosos, diferenciándose entre ellos en el tratamiento que le dan en las camas para integrarlas en un problema en el que típicamente solo se tienen en cuenta los quirófanos. En los algoritmos se ha tenido en cuenta que se está estudiando un horizonte temporal de una semana. El objetivo de todos es el de conseguir que los pacientes de la lista de espera sean operados antes de su fecha de vencimiento, y que todos los pacientes posibles sean operados dentro del marco temporal del problema.

En la evaluación de los algoritmos, se han creado 640 instancias para estudiar los resultados de los 5 algoritmos. Con los 3.200 resultados encima de la mesa, se han hecho comprobaciones teniendo en cuenta diferentes variables para ver cuáles de los algoritmos funcionan mejor.

En el primer estudio en el que se consideraban solo las variables de las camas, se observó que el algoritmo CI, que había sido creado con el pensamiento de que fuese una buena solución para el problema de escasez de camas, actúa justamente, al contrario, mejorando conforme aumentan la cantidad de camas. Las camas PACU son las que más afectan a su resultado, esperable al ser las camas PACU mucho más requeridas por los pacientes que las camas UCI.

Una posible explicación para este comportamiento no intuitivo es el incremento de complejidad en el algoritmo de las camas integradas frente a las separadas. La experimentación está limitada por tiempo a unos 12 segundos por algoritmo, que es el resultado medio de la operación $\left(\frac{m \cdot Q \cdot d}{2} \cdot 25\right)\ milisegundos$, usada para el cálculo del límite de tiempo del algoritmo. La función de decodificación para las camas integradas es bastante más lenta que la versión de camas separadas, pues esta tiene que comprobar en cada paso en qué momento hay camas disponibles. Esto conlleva mayor tiempo de procesamiento que las decodificaciones de camas separadas, que simplemente asignan a quirófanos sin mirar las camas para luego pasar por una función a parte que comprueba ese problema. Esa función en cuanto encuentra una irregularidad ya anuncia que la secuencia no es válida. Por esto se puede pensar que la limitación temporal provoca estos malos resultados, porque, comparativamente con la velocidad del algoritmo, las camas separadas se ejecutan mucho más rápidamente. Además, esto justificaría que el algoritmo mejore cuantas más camas hay, las camas para los pacientes se encuentran con mayor velocidad si hay camas libres y no hay que esperar a que las haya.

El algoritmo que mejor se comporta para el problema de escasez es el CSM2V2. Esto es explicable por el hecho de que las versiones 2 permiten que los pacientes se queden en los quirófanos una vez han concluido las operaciones, lo que hace que este algoritmo este contando con "camas extras" en los momentos en los que el resto están muy limitados por este problema. Por eso tiene sentido que, según el número de camas va aumentando, este algoritmo se vaya igualando con los demás en sus resultados.

El algoritmo CI, solo parece ser algo mejor que los demás para una combinación específica de todas las variables que se han tenido en cuenta en la experimentación, que es para el caso en el que la generación de pacientes se encuentra en sus mínimos valores, y los cirujanos en la parte media de la tabla, entrono a unos 10, siendo independiente de las camas. Este resultado también es visible en el gráfico del caso β, cirujanos, en el que la primera combinación de valores era la mejor para el algoritmo CI y que contenía $\beta = 1$ y cirujanos = 10.

Habiendo dejado claro cuál es el peor de los algoritmos y el mejor de ellos, se pueden extraer algunas posibles conclusiones sobre por qué se han obtenido estos resultados. Primero, el factor de que las versiones 2 permitan que los pacientes se queden en los quirófanos ha propiciado que estas versiones sean las mejores para el problema que nos ocupa. Esto ocurre sin importar el método usado, ya que la versión 2 de ambos métodos supera a la versión 1 de estos.

Segundo, independientemente del tipo de versión, los algoritmos que emplean el segundo método son por lo general mejores que los que emplean el primer método, CSM2V1 es mejor que el CSM1V1 y el CSM2V2 es mejor que el CSM1V2. De esto se puede concluir que es una buena idea hacer nuevas fases de construcción y de destrucción cada vez que se obtienen secuencias que no son válidas en el problema. Hacer esto aporta una mayor perturbación de la secuencia, por lo que se alcanzan más valles en el limitado tiempo que se tiene para el algoritmo. Que estos valles no se puedan estudiar al completo parece no haber sido algo muy influyente en los resultados obtenidos.

En lo que se refiere a los algoritmos que no son CSM2V2 o CI, se puede concluir que funcionan bien prácticamente siempre y que no tienen una tendencia clara, al menos no en los estudios realizados en este trabajo.

Con los experimentos que se han llevado a cabo, se puede concluir que los métodos utilizados hasta el momento en el hospital para organizar las camas, es decir, la propagación del resultado en quirófanos hasta las camas es bastante potente, solo habría que modificar la programación en la integración de las funciones que comprueban la validez en las camas dentro del modelo, y en la perturbación de las secuencias al obtener resultados no válidos en la función anterior. Por último, quedaría como una posibilidad dejar que los quirófanos se utilicen como camas para obtener un mayor rendimiento. Principalmente, si el hospital vuelve a tener problemas de camas por algún hecho como el acontecido por la pandemia.

6. Bibliografía

Graves, Stephen. (1981). A Review of Production Scheduling. Operations Research. 29. 646-675. https://doi.org/10.1287/opre.29.4.646

MacCarthy, Bart & Liu, Jiyin. (1993). Addressing the gap in scheduling research: a review of optimization and heuristic methods in production scheduling. International Journal of Production Research – int j prod res. 31. 59-79. https://doi.org/10.1080/00207549308956713

Framinan, J. M., Leisten, R., & García, R. R. (2014). Manufacturing Scheduling Systems: An Integrated View on Models, Methods and Tools (English Edition) (2014.a ed.). Springer.

Entender ITIL 2011 Normas y mejores prácticas para avanzar hacia ISO 20000 – Los niveles decisionales. (n.d.). Retrieved May 26, 2021, from https://www.ediciones-eni.com/open/mediabook.aspx?idR=1d149467a559fd2acbc4aab12e413dd5

Vahidian, N. (2012). Comparison of Deterministic and Stochastic Production Planning Approaches in Sawmills by Integrating Design of Experiments and Monte- Carlo simulation. https://spectrum.library.concordia.ca/975078/1/Vahidian_MASc_S2013.pdf

Roland, B., Di Martinelly, C., Riane, F., & Pochet, Y. (2010). Scheduling an operating theatre under human resource constraints. Computers & Industrial Engineering, 58(2), 212–220. https://doi.org/10.1016/j.cie.2009.01.005

González-Busto, B., & García, R. (1999). Waiting lists in Spanish public hospitals: a system dynamics approach. System Dynamics Review, 15(3), 201–224. https://doi.org/10.1002/(sici)1099-1727(199923)15:3<201::aid-sdr170>3.0.co;2-5

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

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

de Antonio Suárez, O. (2014). Una aproximación a la heuristica y metaheuristicas. inge@uan – tendencias en la ingeniería, 1(2). Recuperado a partir de http://revistas.uan.edu.co/index.php/ingeuan/article/view/217

Ruiz, R., & Stützle, T. (2007). A simple and effective iterated greedy algorithm for the permutation flowshop scheduling problem. European Journal of Operational Research, 177(3), 2033–2049. https://doi.org/10.1016/j.ejor.2005.12.009

Molina Pariente, J.M. (2016). Operating theatre planning and scheduling in real-life settings. Problem analysis, models, and solution procedures. (Tesis doctoral inédita). Universidad de Sevilla, Sevilla.

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

Medicina intensiva. (2021, 25 de febrero). *Wikipedia, La enciclopedia libre*. Fecha de consulta: 07:47, junio 20, 2021 desde https://es.wikipedia.org/w/index.php?title=Medicina_intensiva&oldid=133518354

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 & Industrial Engineering, 82, 8–20. https://doi.org/10.1016/j.cie.2015.01.006

7. Anexos

7.1 Código principal

```
using System;
using System.IO;

namespace TFG
{
    class Program
    {
        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.
            int numero_quirofanos = 3; // Hay 3 quirofanos pero no están siempre disponibles.
             Depende del día y si es horario de mañana o de tarde.
            int[,] disponibilidadquirofano_turno = new int[numero_quirofanos, dias * 2];
            // Para 5 días de programación existen 10 turnos (mañana y tarde).

            // Suponiendo que los 5 dias que se programan es una semana de lunes a viernes tanto
            por la mañana como por la tarde, los quirofanos que estarán disponibles en cada
            turno serán:
            for (int j = 0; j < dias * 2; j++)
            {
                // Para las mañanas los 3 quirófanos están disponibles
                if (j == 0 || j == 2 || j == 4 || j == 6 || j == 8)
                {
                    for (int i = 0; i < numero_quirofanos; i++)
                    {
                        disponibilidadquirofano_turno[i, j] = 1;
                    }
                }

                // De lunes a jueves por las tardes solo hay 1 quirófano disponible
                else if (j == 1 || j == 3 || j == 5 || j == 7)
                {
                    disponibilidadquirofano_turno[0, j] = 1;
                    disponibilidadquirofano_turno[1, j] = 0;
                    disponibilidadquirofano_turno[2, j] = 0;
                }

                // Para viernes por la tarde 0 quirófanos estarán disponibles
                else
                {
                    disponibilidadquirofano_turno[0, j] = 0;
                    disponibilidadquirofano_turno[1, j] = 0;
                    disponibilidadquirofano_turno[2, j] = 0;
                }
            }

            // PARAMETROS 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 quirofanos.
```

```
 Se usa para determinar el número de cirugías en lista de espera
double[] porcentaje = new double[] { 2, 3, 4 };
int[] cantidad_camas_PACU = new int[] { 2, 3, 4, 5 };
int[] cantidad_camas_UCI = new int[] { 2, 4, 5, 6 };

// DATOS

int m; // Cantidad de pacientes en lista de espera
int numero_cirujanos;
double[] duracion_operacion;
int[] dia_disponible; // Fecha de lanzamiento de la operación
int[] fecha_limite; // Fecha antes de la cual el paciente debería haber sido operado
double[] prioridad;
int[] cirujano_asociado;
double[,] horascirujano_turno; // Indica la disponibilidad de un cirujano en un
 turno
int[] necesidad_paciente_cama; // 0 si el paciente no necesita cama, 1 si necesita
 PACU, 2 si necesita UCI
double[] duracion_paciente_cama; // Depende de la cama que necesite, 0 si no
 necesita

// EXPORTACIÓN
String ruta = AppDomain.CurrentDomain.BaseDirectory;
Console.WriteLine($"{ruta}");
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);

FicheroCSV1.Write("Alpha" + ";" + "Beta" + ";" + "Mds" + ";" + "N Cirujanos" + ";" +
"N Pacientes" + ";" + "N Camas PACU" + ";" + "N Camas UCI" + ";" + "Instancia" + ";"
+ "Resultado" + ";" + "Algoritmo" + "\n");

// GENERACIÓN DE INSTANCIAS
int count = 0;
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 < cantidad_camas_PACU.Length; d++)
            {
                for (int e = 0; e < cantidad_camas_UCI.Length; e++)
                {
                    for (int f = 0; f < 5; f++)
                    {
                        generacion_datos(dias, alpha[a], beta[b], mds[c], f,
                         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,
                         out necesidad_paciente_cama, out duracion_paciente_cama);

                        // Creación de secuencia por regla de despacho LPT
                        int[] secuencia = new int[m];
                        double[] dur_operaciones = new double[m];
```

```
                                Array.Copy(duracion_operacion, dur_operaciones,
                                duracion_operacion.Length);
                                for (int u = 0; u < m; u++)
                                {
                                    secuencia[u] = u;
                                }
                                orden_mayor_menor_y_secuencia(dur_operaciones,
                                 dur_operaciones.Length, secuencia);
                                count++;

                                Console.WriteLine($"{count}. CAMAS INTEGRADAS");
                                iterated_greedy_algorithmCI(secuencia, prioridad,
                                 fecha_limite, disponibilidadquirofano_turno,
                                 duracion_operacion, dia_disponible, cirujano_asociado,
                                 horascirujano_turno, numero_quirofanos, numero_cirujanos,
                                 dias, necesidad_paciente_cama, cantidad_camas_PACU[d],
                                 cantidad_camas_UCI[e], duracion_paciente_cama, alpha[a],
                                 beta[b], mds[c], f, FicheroCSV1);
                                count++;

                                Console.WriteLine($"{count}. CAMAS SEP V1 por MÉTODO 1");
                                iterated_greedy_algorithmCSM1V1(secuencia, prioridad,
                                 fecha_limite, disponibilidadquirofano_turno,
                                 duracion_operacion, dia_disponible, cirujano_asociado,
                                 horascirujano_turno, numero_quirofanos, numero_cirujanos,
                                 dias, necesidad_paciente_cama, cantidad_camas_PACU[d],
                                 cantidad_camas_UCI[e], duracion_paciente_cama, alpha[a],
                                 beta[b], mds[c], f, FicheroCSV1);
                                count++;

                                Console.WriteLine($"{count}. CAMAS SEP V2 por MÉTODO 1");
                                iterated_greedy_algorithmCSM1V2(secuencia, prioridad,
                                 fecha_limite, disponibilidadquirofano_turno,
                                 duracion_operacion, dia_disponible, cirujano_asociado,
                                 horascirujano_turno, numero_quirofanos, numero_cirujanos,
                                 dias, necesidad_paciente_cama, cantidad_camas_PACU[d],
                                 cantidad_camas_UCI[e], duracion_paciente_cama, alpha[a],
                                 beta[b], mds[c], f, FicheroCSV1);
                                count++;

                                Console.WriteLine($"{count}. CAMAS SEP V1 por MÉTODO 2");
                                iterated_greedy_algorithmCSM2V1(secuencia, prioridad,
                                 fecha_limite, disponibilidadquirofano_turno,
                                 duracion_operacion, dia_disponible, cirujano_asociado,
                                 horascirujano_turno, numero_quirofanos, numero_cirujanos,
                                 dias, necesidad_paciente_cama, cantidad_camas_PACU[d],
                                 cantidad_camas_UCI[e], duracion_paciente_cama, alpha[a],
                                 beta[b], mds[c], f, FicheroCSV1);
                                count++;

                                Console.WriteLine($"{count}. CAMAS SEP V2 por MÉTODO 2");
                                iterated_greedy_algorithmCSM2V2(secuencia, prioridad,
                                 fecha_limite, disponibilidadquirofano_turno,
                                 duracion_operacion, dia_disponible, cirujano_asociado,
                                 horascirujano_turno, numero_quirofanos, numero_cirujanos,
                                 dias, necesidad_paciente_cama, cantidad_camas_PACU[d],
                                 cantidad_camas_UCI[e], duracion_paciente_cama, alpha[a],
                                 beta[b], mds[c], f, FicheroCSV1);
                            }
                        }
                    }
                }
            }
        }
        FicheroCSV1.Close();
    }
```

7.2 Código auxiliar

```
// GENERACIÓN DE DATOS DE PARTIDA
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, out int[] necesidad_paciente_cama, out double[]
duracion_paciente_cama)
{
    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 + 4 * 5; // Sumatorio de la capacidad de los quirófanos
     en todo el horizonte de planificación
    double a = SumRjh / 20; // 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, 4); // La  media esperada tomará un valor aleatorio
         entre 1, 2,3 o 4 horas

        double SD = mean * ((aleatorio.Next(0, 2) * 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.Normal normalDist = new
         MathNet.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);
    }

    for (int j = 0; j < m; j++)
    {
        if (duracion_operacion[j] > 6)
        {
            duracion_operacion[j] = 6;
        }
        if (duracion_operacion[j] == 0)
        {
            duracion_operacion[j] = 1;
        }
    }

    // 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 quirofanos disponibles en cada turno
        if (disponibilidadquirofano_turno[j, i] == 1)
        {
            int aux3 = -1;
            while (aux3 == -1)
            {
                // Asigno 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;
                    aux2[cirujano_elegido]++;
                    aux3 = 0;
                }
            }
        }
    }
}

// Si el cirujano puede operar en algun otro turno
for (int i = 0; i < numero_cirujanos; i++)
{
    while (aux2[i] < mds - 1)
    {
        int aux4 = -1;
        // Se escoge 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;
                aux4 = 0;
            }
        }
        aux2[i]++;
    }
}

// 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 esten disponibles en algun 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); // 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
        }

        //Necesidad de cama por parte del paciente y duración que el paciente ocupa la cama.

        necesidad_paciente_cama = new int[m];
        setval_IVector(necesidad_paciente_cama, 0);
        duracion_paciente_cama = new double[m];
        setval_IVector_double(duracion_paciente_cama, 0);

        int n_pacientes_UCI = aleatorio.Next(Convert.ToInt32(m * 0.05), Convert.ToInt32(m *
         0.1));
        int k = m - n_pacientes_UCI; // Número de pacientes que no precisan de UCI
        int n_pacientes_PACU = aleatorio.Next(Convert.ToInt32(k * 0.5), Convert.ToInt32(k *
         1));

        int s = 0;
        while(s < n_pacientes_UCI)
        {
            int w = aleatorio.Next(0, m);
            if(necesidad_paciente_cama[w] == 0)
            {
                necesidad_paciente_cama[w] = 2;
                duracion_paciente_cama[w] = aleatorio.Next(12, 37);
                s++;
            }
            else { continue; }
        }

        int t = 0;
        while (t < n_pacientes_PACU)
        {
            int w = aleatorio.Next(0, m);
            if(necesidad_paciente_cama[w] == 0)
            {
                necesidad_paciente_cama[w] = 1;
                duracion_paciente_cama[w] = aleatorio.Next(1, 4);
                t++;
            }
            else { continue; }
        }

    }

    // FUNCIONES DE DECODIFICACIÓN
    static void decodificacionCI(int[] secuencia, int[,]
    disponibilidadquirofano_turno, double[] duracion_operacion, int[]
    dia_disponible, int[] cirujano_asociado, double[,] horascirujano_turno, int
    numero_quirofanos, int numero_cirujanos, int dias, int[]
    necesidad_paciente_cama, int cantidad_camas_PACU, int cantidad_camas_UCI,
    double[] duracion_paciente_cama, out double[,] info_paciente)
    {
        info_paciente = new double[secuencia.Length, 5];
```

```
for (int l = 0; l < secuencia.Length; l++)
{
    for (int w = 0; w < 5; w++)
    {
        info_paciente[l, w] = 1001;
    }
}

double[,] tiempos_quirofanos = new double[numero_quirofanos, 2]; //Para
 cada QUIRÓFANO el turno y la hora en la que nos encontramos
double[,] tiempos_quirofanos_auxiliar = new double[numero_quirofanos, 2];
 // Inicializo ambos vectores en 0
setval_IMatriz(tiempos_quirofanos, numero_quirofanos, 2, 0);
setval_IMatriz(tiempos_quirofanos_auxiliar, numero_quirofanos, 2, 0);

double[,] tiempos_cirujanos = new double[numero_cirujanos, 2];
 //CIRUJANO, TURNO, HORAS
double[,] tiempos_cirujanos_auxiliar = new double[numero_cirujanos, 2];
 // Inicializo ambos vectores a 0
setval_IMatriz(tiempos_cirujanos, numero_cirujanos, 2, 0);
setval_IMatriz(tiempos_cirujanos_auxiliar, numero_cirujanos, 2, 0);

int[] paciente_incluido = new int[secuencia.Length];
setval_IVector(paciente_incluido, 0);

double[,] tiempos_camas_PACU = new double[cantidad_camas_PACU, 2];
double[,] tiempos_camas_PACU_auxiliar = new double[cantidad_camas_PACU,
 2];
setval_IMatriz(tiempos_camas_PACU, cantidad_camas_PACU, 2, 0);
setval_IMatriz(tiempos_camas_PACU_auxiliar, cantidad_camas_PACU, 2, 0);

double[,] tiempos_camas_UCI = new double[cantidad_camas_UCI, 2];
double[,] tiempos_camas_UCI_auxiliar = new double[cantidad_camas_UCI, 2];
setval_IMatriz(tiempos_camas_UCI, cantidad_camas_UCI, 2, 0);
setval_IMatriz(tiempos_camas_UCI_auxiliar, cantidad_camas_UCI, 2, 0);

int[] cama_asociada_quirofano = new int[numero_quirofanos];

for (int i = 0; i < secuencia.Length; i++) //Bucle de los PACIENTES en
 lista de espera
{
    for (int j = 0; j < numero_quirofanos; j++) //Bucle entre los
     QUIRÓFANOS: el trabajo se incluye en el quirófano que antes lo pueda
     procesar
    {
        igualar_tiempos_auxiliares_a_generales(tiempos_quirofanos,
         tiempos_quirofanos_auxiliar, tiempos_cirujanos,
         tiempos_cirujanos_auxiliar, tiempos_camas_PACU,
         tiempos_camas_PACU_auxiliar, tiempos_camas_UCI,
         tiempos_camas_UCI_auxiliar, cantidad_camas_PACU,
         cantidad_camas_UCI, secuencia, i, cirujano_asociado, j);

        //Actualizar tiempos a release y al momento más adelantado de
         quirófano o cirujano
        release_y_tiempos_quirofano_cirujano(dia_disponible, secuencia,
         i, tiempos_quirofanos_auxiliar, tiempos_cirujanos_auxiliar,
         cirujano_asociado, j);

        double tiempo_inicio_operacion;
        int turno_actual = Convert.ToInt32(tiempos_quirofanos_auxiliar[j,
         0]);
        int paciente_entraria_en_quirofano = 0; //Se usa para salir del
         bucle de los turnos cuando ya se ha podido meter al paciente en
         uno

        for (int x = turno_actual; x < dias * 2; x++) //Aquí meto la
```

```
 restricción del horizonte temporal (dependiente de los días)
{
    //Se comprueba que, si se necesita una cama, esta esté
     disponible en el turno
    comprueba_necesidad_camas(necesidad_paciente_cama, secuencia,
     i, out bool resultado_camas, cantidad_camas_PACU,
     cantidad_camas_UCI, tiempos_camas_UCI, tiempos_camas_PACU,
     x);

    //Si el quirófano está disponible, el cirujano también, y
     hay, al menos, una cama si el paciente la necesita, sigo
     adelante
    if (disponibilidadquirofano_turno[j, x] == 1 &&
     horascirujano_turno[cirujano_asociado[secuencia[i]], x] != 0
     && resultado_camas == true)
    {
        //Este es el mejor momento en el que puede empezar, sin
         contar con las camas
        tiempo_inicio_operacion = tiempos_quirofanos_auxiliar[j,
         1];

        tiempo_maximo_turno(x, out double tiempo_maximo);

        if (tiempo_inicio_operacion +
         duracion_operacion[secuencia[i]] <= tiempo_maximo)
        {
            if (necesidad_paciente_cama[secuencia[i]] == 1)
            {
                double mejor_turno_cama_PACU = 1000;
                double mejor_hora_cama_PACU = 1000;
                busqueda_cama(tiempos_camas_PACU_auxiliar,
                 tiempos_quirofanos_auxiliar,
                 tiempo_inicio_operacion, duracion_operacion,
                 secuencia, i, cama_asociada_quirofano,
                 mejor_turno_cama_PACU, mejor_hora_cama_PACU, j,
                 cantidad_camas_PACU, out mejor_turno_cama_PACU,
                 out mejor_hora_cama_PACU);

                //Establece los tiempos auxiliares según los
                 momentos en la hora a la que la cama esté
                 disponible en el turno

                 asignacion_temporal_camas(
                  tiempos_quirofanos_auxiliar,
                  mejor_turno_cama_PACU, mejor_hora_cama_PACU,
                  tiempo_inicio_operacion, paciente_incluido,
                  secuencia, i, paciente_entraria_en_quirofano,
                  cama_asociada_quirofano, duracion_operacion,
                  tiempos_cirujanos_auxiliar, cirujano_asociado,
                  tiempo_maximo, j, x, out
                  paciente_entraria_en_quirofano);
            }
            else if (necesidad_paciente_cama[secuencia[i]] == 2)
            {
                double mejor_turno_cama_UCI = 1000;
                double mejor_hora_cama_UCI = 1000;
                busqueda_cama(tiempos_camas_UCI_auxiliar,
                 tiempos_quirofanos_auxiliar,
                 tiempo_inicio_operacion, duracion_operacion,
                 secuencia, i, cama_asociada_quirofano,
                 mejor_turno_cama_UCI, mejor_hora_cama_UCI, j,
                 cantidad_camas_UCI,
                 out mejor_turno_cama_UCI, out
                 mejor_hora_cama_UCI);

                asignacion_temporal_camas(
                 tiempos_quirofanos_auxiliar,
```

```
                        mejor_turno_cama_UCI, mejor_hora_cama_UCI,
                        tiempo_inicio_operacion, paciente_incluido,
                        secuencia, i, paciente_entraria_en_quirofano,
                        cama_asociada_quirofano, duracion_operacion,
                        tiempos_cirujanos_auxiliar, cirujano_asociado,
                        tiempo_maximo, j, x, out
                        paciente_entraria_en_quirofano);
                    }
                    else
                    {
                        tiempos_quirofanos_auxiliar[j, 1] =
                        tiempo_inicio_operacion +
                        duracion_operacion[secuencia[i]];
                        paciente_incluido[secuencia[i]] = 1;
                        paciente_entraria_en_quirofano = 1;
                    }
                }
                //Actualizar variables si el tiempo es insuficiente en el
                 turno
                else
                {
                    tiempos_quirofanos_auxiliar[j, 0]++;
                    tiempos_quirofanos_auxiliar[j, 1] = 0;

                    tiempos_cirujanos_auxiliar[cirujano_asociado
                     [secuencia[i]], 0]++;
                    tiempos_cirujanos_auxiliar[cirujano_asociado
                     [secuencia[i]], 1] = 0;
                }

            }

            //Si alguna de las condiciones del if no se cumplen,
             actualizo las variables para pasar al siguiente turno
            else
            {//Actualizar las variables
                tiempos_quirofanos_auxiliar[j, 0]++;
                tiempos_quirofanos_auxiliar[j, 1] = 0;
                tiempos_cirujanos_auxiliar[cirujano_asociado
                 [secuencia[i]], 0]++;
                tiempos_cirujanos_auxiliar[cirujano_asociado
                 [secuencia[i]], 1] = 0;
            }

            //Si estamos en el último turno y el paciente no ha entrado
            if (x == dias * 2 - 1 && paciente_entraria_en_quirofano == 0)
            {
                tiempos_quirofanos_auxiliar[j, 0] = 1001;
                tiempos_quirofanos_auxiliar[j, 1] = 1001; //Tiempos
                 imposibles con las limitaciones del problema
            }
            //Salgo del bucle si el paciente ya ha podido entrar en el
             quirofano en el turno actual
            if (paciente_entraria_en_quirofano == 1) { break; }

        }
    }

    if (paciente_incluido[secuencia[i]] == 1)
    {
        int quirofano_asociado = 1000; //Imposible por las limitaciones
         del problema
        double turno_minimo_posible = 1001;
        double hora_minima_posible = 1001;
        for (int j = 0; j < numero_quirofanos; j++)
        {//Busco el quirófano que cumple con la regla FAM de asignación
          del paciente
```

```
                if (tiempos_quirofanos_auxiliar[j, 0] < turno_minimo_posible)
                {
                    turno_minimo_posible = tiempos_quirofanos_auxiliar[j, 0];
                    hora_minima_posible = tiempos_quirofanos_auxiliar[j, 1];
                    quirofano_asociado = j;
                }
                else if (tiempos_quirofanos_auxiliar[j, 0] ==
                 turno_minimo_posible)
                {//Si para dos quirófanos o los tres se coincide en turnos,
                  compruebo sus horas para quedarme con el mejor
                 //Se establece de forma que, a igual horaria de uno o más
                  quirófanos, me quedo con el último
                    if (tiempos_quirofanos_auxiliar[j, 1] <=
                     hora_minima_posible)
                    {
                        turno_minimo_posible = tiempos_quirofanos_auxiliar[j,
                         0];
                        hora_minima_posible = tiempos_quirofanos_auxiliar[j,
                         1];
                        quirofano_asociado = j;
                    }
                }
            }
            //Actualizo las variables generales de quirófano y cirujano
            tiempos_quirofanos[quirofano_asociado, 0] = turno_minimo_posible;
            tiempos_quirofanos[quirofano_asociado, 1] = hora_minima_posible;
            tiempos_cirujanos[cirujano_asociado[secuencia[i]], 0] =
             turno_minimo_posible;
            tiempos_cirujanos[cirujano_asociado[secuencia[i]], 1] =
             hora_minima_posible;

            info_paciente[i, 0] = secuencia[i];
            info_paciente[i, 1] = quirofano_asociado;
            info_paciente[i, 2] = turno_minimo_posible;
            info_paciente[i, 3] = hora_minima_posible;
            info_paciente[i, 4] = hora_minima_posible +
             duracion_operacion[secuencia[i]];

            if (necesidad_paciente_cama[secuencia[i]] == 1)
            {
                //En esta función normalizo los tiempos y actualizo las
                 variables de la cama
                duracion_a_turnos_horasCI(turno_minimo_posible,
                 hora_minima_posible, duracion_paciente_cama, secuencia[i],
                 tiempos_camas_PACU, cama_asociada_quirofano,
                 quirofano_asociado);
            }
            else if (necesidad_paciente_cama[secuencia[i]] == 2)
            {
                duracion_a_turnos_horasCI(turno_minimo_posible,
                 hora_minima_posible, duracion_paciente_cama, secuencia[i],
                 tiempos_camas_UCI, cama_asociada_quirofano,
                 quirofano_asociado);
            }
        }

        else
        {
            //No se ha podido introducir al paciente, dejo de buscar
            break;
        }

    }
}

static void decodificacionCS(int[] secuencia, int[,] disponibilidadquirofano_turno,
double[] duracion_operacion, int[] dia_disponible, int[] cirujano_asociado,
```

```
double[,] horascirujano_turno, int numero_quirofanos, int numero_cirujanos, int dias, out
double[,] info_paciente)
 {
     info_paciente = new double[secuencia.Length, 5];

     for (int l = 0; l < secuencia.Length; l++)
     {
         for (int w = 0; w < 5; w++)
         {
             info_paciente[l, w] = 1001;
         }
     }

     double[,] tiempos_quirofanos = new double[numero_quirofanos, 2]; //Para cada
      QUIRÓFANO el turno y la hora en la que nos encontramos
     double[,] tiempos_quirofanos_auxiliar = new double[numero_quirofanos, 2];
     //Inicializo ambos vectores en 0
     setval_IMatriz(tiempos_quirofanos, numero_quirofanos, 2, 0);
     setval_IMatriz(tiempos_quirofanos_auxiliar, numero_quirofanos, 2, 0);

     double[,] tiempos_cirujanos = new double[numero_cirujanos, 2]; //CIRUJANO, TURNO,
      HORAS
     double[,] tiempos_cirujanos_auxiliar = new double[numero_cirujanos, 2];
     //Inicializo ambos vectores a 0
     setval_IMatriz(tiempos_cirujanos, numero_cirujanos, 2, 0);
     setval_IMatriz(tiempos_cirujanos_auxiliar, numero_cirujanos, 2, 0);

     int[] paciente_incluido = new int[secuencia.Length];
     setval_IVector(paciente_incluido, 0);

     for (int i = 0; i < secuencia.Length; i++) //Bucle de los PACIENTES en lista de
      espera
     {
         for (int j = 0; j < numero_quirofanos; j++) //Bucle entre los QUIRÓFANOS: el
          trabajo se incluye en el quirófano que antes lo pueda procesar
         {
             tiempos_quirofanos_auxiliar[j, 0] = tiempos_quirofanos[j, 0];
             tiempos_quirofanos_auxiliar[j, 1] = tiempos_quirofanos[j, 1];
             tiempos_cirujanos_auxiliar[cirujano_asociado[secuencia[i]], 0] =
              tiempos_cirujanos[cirujano_asociado[secuencia[i]], 0];
             tiempos_cirujanos_auxiliar[cirujano_asociado[secuencia[i]], 1] =
              tiempos_cirujanos[cirujano_asociado[secuencia[i]], 1];

             if (dia_disponible[secuencia[i]] > tiempos_quirofanos[j, 0]) //Actualizo el
              tiempo del quirófano al momento en el que llega el paciente
             {
                 tiempos_quirofanos_auxiliar[j, 0] = dia_disponible[secuencia[i]];
                 tiempos_quirofanos_auxiliar[j, 1] = 0;
             }
             //Actualizo el tiempo quirófano o cirujano, según cuál vaya más adelantado
             if (tiempos_quirofanos_auxiliar[j, 0] <
              tiempos_cirujanos[cirujano_asociado[secuencia[i]], 0] ||
              (tiempos_quirofanos_auxiliar[j, 0] ==
              tiempos_cirujanos[cirujano_asociado[secuencia[i]], 0] &&
              tiempos_quirofanos_auxiliar[j, 1] <
              tiempos_cirujanos[cirujano_asociado[secuencia[i]], 1])) //Arreglar para las
               horas
             {
                 tiempos_quirofanos_auxiliar[j, 0] = tiempos_cirujanos
                  [cirujano_asociado[secuencia[i]], 0];
                 tiempos_quirofanos_auxiliar[j, 1] = tiempos_cirujanos
                  [cirujano_asociado[secuencia[i]], 1];
             }
             else
             {
                 tiempos_cirujanos_auxiliar[cirujano_asociado[secuencia[i]], 0] =
                  tiempos_quirofanos_auxiliar[j, 0];
```

```
            tiempos_cirujanos_auxiliar[cirujano_asociado[secuencia[i]], 1] =
             tiempos_quirofanos_auxiliar[j, 1];
        }

        double tiempo_inicio_operacion;
        int turno_actual = Convert.ToInt32(tiempos_quirofanos_auxiliar[j, 0]);
        int paciente_entraria_en_quirofano = 0; //Se usa para salir del bucle de los
         turnos cuando ya se ha podido meter al paciente en uno

        for (int x = turno_actual; x < dias * 2; x++) //Aquí meto la restricción del
         horizonte temporal (dependiente de los días)
        {
            if (disponibilidadquirofano_turno[j, x] == 1 && horascirujano_turno
             [cirujano_asociado[secuencia[i]], x] != 0)
            {//Si el quirófano está disponible en este TURNO, y si el cirujano
              también lo está, sigo con el código, si no ni entro.
                tiempo_inicio_operacion = tiempos_quirofanos_auxiliar[j, 1]; //ya
                 está actualizado

                double tiempo_maximo;
                tiempo_maximo_turno(x, out tiempo_maximo);

                if (tiempo_inicio_operacion + duracion_operacion[secuencia[i]] <=
                 tiempo_maximo)
                {//Se puede llevar a cabo la operación
                    tiempos_quirofanos_auxiliar[j, 1] = tiempo_inicio_operacion +
                     duracion_operacion[secuencia[i]];
                    paciente_incluido[secuencia[i]] = 1;
                    paciente_entraria_en_quirofano = 1;
                }
                else
                {//Actualizar variables
                    tiempos_quirofanos_auxiliar[j, 0]++;
                    tiempos_quirofanos_auxiliar[j, 1] = 0;
                    tiempos_cirujanos_auxiliar[cirujano_asociado[secuencia[i]],
                     0]++;
                    tiempos_cirujanos_auxiliar[cirujano_asociado[secuencia[i]], 1] =
                     0;
                }

            }

            else
            {//Actualizar las variables
                tiempos_quirofanos_auxiliar[j, 0]++;
                tiempos_quirofanos_auxiliar[j, 1] = 0;
                tiempos_cirujanos_auxiliar[cirujano_asociado[secuencia[i]], 0]++;
                tiempos_cirujanos_auxiliar[cirujano_asociado[secuencia[i]], 1] = 0;
            }

            if (x == dias * 2 - 1 && paciente_entraria_en_quirofano == 0) //Si
             estamos en el último turno y el paciente no ha entrado
            {
                tiempos_quirofanos_auxiliar[j, 0] = 1001;
                tiempos_quirofanos_auxiliar[j, 1] = 1001; //Tiempos imposibles con
                 las limitaciones del problema
            }
            //Salgo del bucle si el paciente ya ha podido entrar en el quirófano en
             el turno actual
            if (paciente_entraria_en_quirofano == 1) { break; }

        }
    }

    if (paciente_incluido[secuencia[i]] == 1)
    {
        int quirofano_asociado = 1001; //Imposible por las limitaciones del problema
```

```
            double turno_minimo_posible = 1001;
            double hora_minima_posible = 1001;
            for (int j = 0; j < numero_quirofanos; j++)
            {//Busco el quirófano que cumple con la regla FAM de asignación del paciente
                if (tiempos_quirofanos_auxiliar[j, 0] < turno_minimo_posible)
                {
                    turno_minimo_posible = tiempos_quirofanos_auxiliar[j, 0];
                    hora_minima_posible = tiempos_quirofanos_auxiliar[j, 1];
                    quirofano_asociado = j;
                }
                else if (tiempos_quirofanos_auxiliar[j, 0] == turno_minimo_posible)
                {//Si para dos quirófanos o los tres se coincide en turnos, compruebo
                  sus horas para quedarme con el mejor
                    if (tiempos_quirofanos_auxiliar[j, 1] <= hora_minima_posible)
                    {
                        turno_minimo_posible = tiempos_quirofanos_auxiliar[j, 0];
                        hora_minima_posible = tiempos_quirofanos_auxiliar[j, 1];
                        quirofano_asociado = j;
                    }
                }
            }
            //Actualizo las variables generales de quirófano y cirujano
            tiempos_quirofanos[quirofano_asociado, 0] = turno_minimo_posible;
            tiempos_quirofanos[quirofano_asociado, 1] = hora_minima_posible;
            tiempos_cirujanos[cirujano_asociado[secuencia[i]], 0] =
             turno_minimo_posible;
            tiempos_cirujanos[cirujano_asociado[secuencia[i]], 1] = hora_minima_posible;
            info_paciente[i, 2] = turno_minimo_posible;
            info_paciente[i, 4] = hora_minima_posible;
            info_paciente[i, 0] = secuencia[i];
            info_paciente[i, 1] = quirofano_asociado;
            info_paciente[i, 3] = hora_minima_posible - duracion_operacion
             [secuencia[i]];
        }

        else { break; }
    }
}

// ITERATED GREEDY ALGORITHM CAMAS INTEGRADAS

static void iterated_greedy_algorithmCI(int[] secuencia, double[] prioridad, int[]
 fecha_limite, int[,] disponibilidadquirofano_turno, double[] duracion_operacion, int[]
 dia_disponible, int[] cirujano_asociado, double[,] horascirujano_turno, int
 numero_quirofanos, int numero_cirujanos, int dias, int[] necesidad_paciente_cama, int
 cantidad_camas_PACU, int cantidad_camas_UCI, double[] duracion_paciente_cama, double
 alpha, double beta, int mds, int instancia, StreamWriter FicheroCSV1)
{
    Random random = new Random();
    double numAleat = random.NextDouble();
    if (numAleat == 0.0)
    {
        numAleat = 0.1;
    }

    double temperatura = 0.4;

    // Primero, con la secuencia que llega al algoritmo, se saca la primera solución
     inicial, y se elige la secuencia como la mejor, por el momento
    double[,] info_paciente;
    decodificacionCI(secuencia, disponibilidadquirofano_turno, duracion_operacion,
     dia_disponible, cirujano_asociado, horascirujano_turno, numero_quirofanos,
     numero_cirujanos, dias, necesidad_paciente_cama, cantidad_camas_PACU,
     cantidad_camas_UCI, duracion_paciente_cama, out info_paciente); //Se decodifica la
     secuencia

    // Se asigna el resultado como el mejor hasta el momento
```

```
double resultado_mejor = funcion_objetivo(info_paciente, secuencia, prioridad,
 fecha_limite, dias);
double resultado_final = resultado_mejor;
double resultado_LS = resultado_mejor;

// Se va a guardar también el resultado de la mejor secuencia:
double[,] mejor_info_paciente = new double[secuencia.Length, 5];
copiar_matriz(info_paciente, mejor_info_paciente, secuencia.Length, 5);
double[,] info_paciente_temporal = new double[secuencia.Length, 5];
copiar_matriz(info_paciente, info_paciente_temporal, secuencia.Length, 5);

int[] mejor_secuencia = new int[secuencia.Length];
Array.Copy(secuencia, mejor_secuencia, secuencia.Length);
int[] secuencia_LS = new int[secuencia.Length];
Array.Copy(secuencia, secuencia_LS, secuencia.Length);
DateTime tiempoFinal = DateTime.Now.AddMilliseconds(secuencia.Length *
 numero_quirofanos * dias / 2 * 25);

while (DateTime.Now <= tiempoFinal)
{
    // Primera fase: Destrucción y construcción
    int[] secuencia_final;
    fase_destruccion_construccion(secuencia_LS, out secuencia_final, 4);

    decodificacionCI(secuencia_final, disponibilidadquirofano_turno,
     duracion_operacion, dia_disponible, cirujano_asociado, horascirujano_turno,
     numero_quirofanos, numero_cirujanos, dias, necesidad_paciente_cama,
     cantidad_camas_PACU, cantidad_camas_UCI, duracion_paciente_cama, out
     info_paciente);

    resultado_final = funcion_objetivo(info_paciente, secuencia_final, prioridad,
     fecha_limite, dias);

    //Segunda fase: Local Search
    bool improvement;
    do
    {
        improvement = false;
        double resultado_nuevo;

        for (int i = 0; i < secuencia_final.Length; i++)
        {
            for (int j = 0; j < secuencia_final.Length; j++)
            {
                //La secuencia que se va a usar ya no es la mejor secuencia es la
                 secuencia en uso, que se llama secuencia_final
                int[] secuencia_final_aux = new int[secuencia_final.Length]; // Para
                 evitar cambios accidentales en la mejor_secuencia actual
                Array.Copy(secuencia_final, secuencia_final_aux,
                 secuencia_final.Length);
                int[] nueva_secuencia = new int[secuencia_final.Length]; // La que
                 dará la nueva secuencia en la iteración actual
                int[] secuencia_auxiliar = new int[secuencia_final.Length - 1];
                int trabajo;
                if (i != j && j != i - 1)
                {
                    trabajo = sacar_trabajo_LS(secuencia_final_aux,
                     secuencia_auxiliar, i);
                    introducir_trabajo_LS(secuencia_auxiliar, nueva_secuencia, j,
                     trabajo);
                }
                else { continue; }

                //Cuando se tiene la nueva secuencia, se decodifica y se saca su
                 valor en la función objetivo
                decodificacionCI(nueva_secuencia, disponibilidadquirofano_turno,
                 duracion_operacion, dia_disponible, cirujano_asociado,
```

```
                    horascirujano_turno, numero_quirofanos, numero_cirujanos, dias,
                    necesidad_paciente_cama, cantidad_camas_PACU, cantidad_camas_UCI,
                    duracion_paciente_cama, out info_paciente);

                    resultado_nuevo = funcion_objetivo(info_paciente, nueva_secuencia,
                     prioridad, fecha_limite, dias);

                    //Si se obtiene mejor resultado en la funcion objetivo, se usa la
                     nueva secuencia, y el nuevo valor a batir es el suyo
                    if (resultado_nuevo < resultado_final)
                    {
                        resultado_final = resultado_nuevo;
                        Array.Copy(nueva_secuencia, secuencia_final,
                         nueva_secuencia.Length);
                        copiar_matriz(info_paciente, info_paciente_temporal,
                         secuencia.Length, 5);
                        improvement = true;
                    }
                }
            }
        } while (improvement == true);

        //Simulated Annealing para decidir si la secuencia que sale de LS se emplea en
         la siguiente ronda
        if (resultado_final < resultado_LS)
        {
            Array.Copy(secuencia_final, secuencia_LS, secuencia_final.Length);
            resultado_LS = resultado_final;
            if (resultado_final < resultado_mejor)
            {
                Array.Copy(secuencia_final, mejor_secuencia, secuencia_final.Length);
                resultado_mejor = resultado_final;
                copiar_matriz(info_paciente, mejor_info_paciente, secuencia.Length, 5);
            }
        }
        else if (numAleat <= Math.Exp(-(resultado_final - resultado_LS) / temperatura))
        {
            Array.Copy(secuencia_final, secuencia_LS, secuencia_final.Length);
            resultado_LS = resultado_final;
        }
    }

    Console.WriteLine($"El mejor resultado de la FO obtenido en IGCmsInt:
     {resultado_mejor}\n");

    // Impresión
    FicheroCSV1.Write(alpha + ";" + beta + ";" + mds + ";" + numero_cirujanos + ";" +
     secuencia.Length + ";" + cantidad_camas_PACU + ";" + cantidad_camas_UCI + ";" +
     instancia + ";" + resultado_mejor + ";" + "CI" + "\n");
}

// ITERATED GREEDY ALGORITHM MÉTODO 1 VERSIÓN 2

static void iterated_greedy_algorithmCSM1V2(int[] secuencia, double[] prioridad, int[]
 fecha_limite, int[,] disponibilidadquirofano_turno, double[] duracion_operacion, int[]
 dia_disponible, int[] cirujano_asociado, double[,] horascirujano_turno, int
 numero_quirofanos, int numero_cirujanos, int dias, int[] necesidad_paciente_cama, int
 cantidad_camas_PACU, int cantidad_camas_UCI, double[] duracion_paciente_cama, double
 alpha, double beta, int mds, int instancia, StreamWriter FicheroCSV1)
{
    Random random = new Random();
    double numAleat = random.NextDouble();
    if (numAleat == 0.0)
    {
        numAleat = 0.1;
    }
```

```
double temperatura = 0.4;

// Primero, con la secuencia que llega al algoritmo, se saca la primera solución
 inicial, y se elige la secuencia como la mejor, por el momento
double[,] info_paciente;
bool secuencia_valida;
int cont = 0;
int[] sec = new int[secuencia.Length];

do // Bucle para asegurar que sale una secuencia válida antes de entrar al LS
{
    if (cont != 0)
    {
        fase_destruccion_construccion(secuencia, out sec, 4);
    }
    else
    {
        Array.Copy(secuencia, sec, secuencia.Length);
    }

    decodificacionCS(sec, disponibilidadquirofano_turno, duracion_operacion,
     dia_disponible, cirujano_asociado, horascirujano_turno, numero_quirofanos,
     numero_cirujanos, dias, out info_paciente); //Se decodifica la secuencia
    secuencia_valida = camas_separadas_V2(cantidad_camas_PACU, cantidad_camas_UCI,
     necesidad_paciente_cama, duracion_paciente_cama, info_paciente,
     secuencia.Length);

    if (info_paciente[0, 0] == 1001) { secuencia_valida = false; }
    cont++;
} while (secuencia_valida == false);

// Se asigna el resultado como el mejor hasta el momento
double resultado_mejor = funcion_objetivo(info_paciente, sec, prioridad,
 fecha_limite, dias);
double resultado_final = resultado_mejor;
double resultado_LS = resultado_mejor;

// Se va a guardar también el resultado de la mejor secuencia:
double[,] mejor_info_paciente = new double[secuencia.Length, 5];
copiar_matriz(info_paciente, mejor_info_paciente, secuencia.Length, 5);
double[,] info_paciente_temporal = new double[secuencia.Length, 5];
copiar_matriz(info_paciente, info_paciente_temporal, secuencia.Length, 5);

int[] mejor_secuencia = new int[secuencia.Length];
Array.Copy(sec, mejor_secuencia, sec.Length);
int[] secuencia_LS = new int[secuencia.Length];
Array.Copy(mejor_secuencia, secuencia_LS, mejor_secuencia.Length);
DateTime tiempoFinal = DateTime.Now.AddMilliseconds(secuencia.Length *
 numero_quirofanos * dias / 2 * 25);

while (DateTime.Now <= tiempoFinal)
{
    // Primera fase: Destrucción y Construcción
    int[] secuencia_final;
    fase_destruccion_construccion(secuencia_LS, out secuencia_final, 4);

    decodificacionCS(secuencia_final, disponibilidadquirofano_turno,
     duracion_operacion, dia_disponible, cirujano_asociado, horascirujano_turno,
     numero_quirofanos, numero_cirujanos, dias, out info_paciente);

    resultado_final = funcion_objetivo(info_paciente, secuencia_final, prioridad,
     fecha_limite, dias);

    if (camas_separadas_V2(cantidad_camas_PACU, cantidad_camas_UCI,
     necesidad_paciente_cama, duracion_paciente_cama, info_paciente,
     secuencia.Length) == false)
     { resultado_final = 1; }
```

```
//Segunda fase: Local Search
bool improvement;
do
{
    improvement = false;
    double resultado_nuevo;

    for (int i = 0; i < secuencia_final.Length; i++)
    {
        for (int j = 0; j < secuencia_final.Length; j++)
        {
            bool sec_val;
            int[] secuencia_final_aux = new int[secuencia_final.Length];
            Array.Copy(secuencia_final, secuencia_final_aux,
             secuencia_final.Length);
            int[] nueva_secuencia = new int[secuencia_final.Length];
            int[] secuencia_auxiliar = new int[secuencia_final.Length - 1];
            int trabajo;
            if (i != j && j != i - 1)
            {
                trabajo = sacar_trabajo_LS(secuencia_final_aux,
                 secuencia_auxiliar, i);
                introducir_trabajo_LS(secuencia_auxiliar, nueva_secuencia, j,
                 trabajo);
            }
            else { continue; }

            decodificacionCS(nueva_secuencia, disponibilidadquirofano_turno,
             duracion_operacion, dia_disponible, cirujano_asociado,
             horascirujano_turno, numero_quirofanos, numero_cirujanos, dias, out
             info_paciente);

            sec_val = camas_separadas_V2(cantidad_camas_PACU,
             cantidad_camas_UCI, necesidad_paciente_cama, duracion_paciente_cama,
             info_paciente, secuencia.Length);

            if (sec_val == true)
            {
                resultado_nuevo = funcion_objetivo(info_paciente,
                 nueva_secuencia, prioridad, fecha_limite, dias);
            }
            else
            {
                resultado_nuevo = 100; // Gran penalización por ser una
                 secuencia no válida
            }

            if (resultado_nuevo < resultado_final)
            {
                resultado_final = resultado_nuevo;
                Array.Copy(nueva_secuencia, secuencia_final,
                 nueva_secuencia.Length);
                copiar_matriz(info_paciente, info_paciente_temporal,
                 secuencia.Length, 5);
                improvement = true;
            }
        }
    }
} while (improvement == true);

//Simulated Annealing para decidir si la secuencia que sale de LS se emplea en
 la siguiente ronda
if (resultado_final < resultado_LS)
{
    Array.Copy(secuencia_final, secuencia_LS, secuencia_final.Length);
    resultado_LS = resultado_final;
```

```
            if (resultado_final < resultado_mejor)
            {
                Array.Copy(secuencia_final, mejor_secuencia, secuencia_final.Length);
                resultado_mejor = resultado_final;
                copiar_matriz(info_paciente, mejor_info_paciente, secuencia.Length, 5);
            }
        }
        else if (numAleat <= Math.Exp(-(resultado_final - resultado_LS) / temperatura))
        {
            Array.Copy(secuencia_final, secuencia_LS, secuencia_final.Length);
            resultado_LS = resultado_final;
        }
    }
    Console.WriteLine($"El mejor resultado de la FO obtenido en IGCmsSepV2:
     {resultado_mejor}\n");

    //Impresion
    FicheroCSV1.Write(alpha + ";" + beta + ";" + mds + ";" + numero_cirujanos + ";" +
     secuencia.Length + ";" + cantidad_camas_PACU + ";" + cantidad_camas_UCI + ";" +
     instancia + ";" + resultado_mejor + ";" + "CSM1V2" + "\n");
}

// ITERATED GREEDY ALGORITHM MÉTODO 1 VERSIÓN 1

static void iterated_greedy_algorithmCSM1V1(int[] secuencia, double[] prioridad, int[]
 fecha_limite, int[,] disponibilidadquirofano_turno, double[] duracion_operacion, int[]
 dia_disponible, int[] cirujano_asociado, double[,] horascirujano_turno, int
 numero_quirofanos, int numero_cirujanos, int dias, int[] necesidad_paciente_cama, int
 cantidad_camas_PACU, int cantidad_camas_UCI, double[] duracion_paciente_cama, double
 alpha, double beta, int mds, int instancia, StreamWriter FicheroCSV1)
{
    Random random = new Random();
    double numAleat = random.NextDouble();
    if (numAleat == 0.0)
    {
        numAleat = 0.1;
    }

    double temperatura = 0.4;

    //Primero, con la secuencia que llega al algoritmo, se saca la primera solución
     inicial, y se elige la secuencia como la mejor, por el momento
    double[,] info_paciente;
    bool secuencia_valida;
    int cont = 0;
    int[] sec = new int[secuencia.Length];

    do //Bucle para asegurar que sale una secuencia válida antes de entrar al LS
    {
        if (cont != 0)
        {
            fase_destruccion_construccion(secuencia, out sec, 4);
        }
        else
        {
            Array.Copy(secuencia, sec, secuencia.Length);
        }

        decodificacionCS(sec, disponibilidadquirofano_turno, duracion_operacion,
         dia_disponible, cirujano_asociado, horascirujano_turno, numero_quirofanos,
         numero_cirujanos, dias, out info_paciente); //Se decodifica la secuencia
        secuencia_valida = camas_separadas_V1(cantidad_camas_PACU, cantidad_camas_UCI,
         necesidad_paciente_cama, duracion_paciente_cama, info_paciente,
         secuencia.Length);

        if (info_paciente[0, 0] == 1001) { secuencia_valida = false; }
        cont++;
```

```
} while (secuencia_valida == false);

//Se asigna el resultado como el mejor hasta el momento
double resultado_mejor = funcion_objetivo(info_paciente, sec, prioridad,
 fecha_limite, dias);
double resultado_final = resultado_mejor;
double resultado_LS = resultado_mejor;

//Se va a guardar también el resultado de la mejor secuencia:
double[,] mejor_info_paciente = new double[secuencia.Length, 5];
copiar_matriz(info_paciente, mejor_info_paciente, secuencia.Length, 5);
double[,] info_paciente_temporal = new double[secuencia.Length, 5];
copiar_matriz(info_paciente, info_paciente_temporal, secuencia.Length, 5);

int[] mejor_secuencia = new int[secuencia.Length];
Array.Copy(sec, mejor_secuencia, sec.Length);
int[] secuencia_LS = new int[secuencia.Length];
Array.Copy(mejor_secuencia, secuencia_LS, mejor_secuencia.Length);
DateTime tiempoFinal = DateTime.Now.AddMilliseconds(secuencia.Length *
 numero_quirofanos * dias / 2 * 25);

while (DateTime.Now <= tiempoFinal)
{
    // Primera fase: Destrucción y Construcción
    int[] secuencia_final;
    fase_destruccion_construccion(secuencia_LS, out secuencia_final, 4);

    decodificacionCS(secuencia_final, disponibilidadquirofano_turno,
     duracion_operacion, dia_disponible, cirujano_asociado, horascirujano_turno,
     numero_quirofanos, numero_cirujanos, dias, out info_paciente);

    resultado_final = funcion_objetivo(info_paciente, secuencia_final, prioridad,
     fecha_limite, dias);

    if (camas_separadas_V1(cantidad_camas_PACU, cantidad_camas_UCI,
     necesidad_paciente_cama, duracion_paciente_cama, info_paciente,
     secuencia.Length) == false)
     { resultado_final = 1; }

    // Segunda fase: Local Search
    bool improvement;
    do
    {
        improvement = false;
        double resultado_nuevo;

        for (int i = 0; i < secuencia_final.Length; i++)
        {
            for (int j = 0; j < secuencia_final.Length; j++)
            {
                bool sec_val;
                int[] secuencia_final_aux = new int[secuencia_final.Length];
                Array.Copy(secuencia_final, secuencia_final_aux,
                 secuencia_final.Length);
                int[] nueva_secuencia = new int[secuencia_final.Length];
                int[] secuencia_auxiliar = new int[secuencia_final.Length - 1];
                int trabajo;
                if (i != j && j != i - 1)
                {
                    trabajo = sacar_trabajo_LS(secuencia_final_aux,
                     secuencia_auxiliar, i);
                    introducir_trabajo_LS(secuencia_auxiliar, nueva_secuencia, j,
                     trabajo);
                }
                else { continue; }

                decodificacionCS(nueva_secuencia, disponibilidadquirofano_turno,
```

```
                    duracion_operacion, dia_disponible, cirujano_asociado,
                    horascirujano_turno, numero_quirofanos, numero_cirujanos, dias, out
                    info_paciente);
                   sec_val = camas_separadas_V1(cantidad_camas_PACU,
                    cantidad_camas_UCI, necesidad_paciente_cama, duracion_paciente_cama,
                    info_paciente, secuencia.Length);

                   if (sec_val == true)
                   {
                       resultado_nuevo = funcion_objetivo(info_paciente,
                        nueva_secuencia, prioridad, fecha_limite, dias);
                   }
                   else
                   {
                       resultado_nuevo = 100; // Gran penalización por ser una
                        secuencia no válida
                   }

                   if (resultado_nuevo < resultado_final)
                   {
                       resultado_final = resultado_nuevo;
                       Array.Copy(nueva_secuencia, secuencia_final,
                        nueva_secuencia.Length);
                       copiar_matriz(info_paciente, info_paciente_temporal,
                        secuencia.Length, 5);
                       improvement = true;
                   }
               }
           }
       } while (improvement == true);

       //Simulated Annealing para decidir si la secuencia que sale de LS se emplea en
        la siguiente ronda
       if (resultado_final < resultado_LS)
       {
           Array.Copy(secuencia_final, secuencia_LS, secuencia_final.Length);
           resultado_LS = resultado_final;
           if (resultado_final < resultado_mejor)
           {
               Array.Copy(secuencia_final, mejor_secuencia, secuencia_final.Length);
               resultado_mejor = resultado_final;
               copiar_matriz(info_paciente, mejor_info_paciente, secuencia.Length, 5);
           }
       }
       else if (numAleat <= Math.Exp(-(resultado_final - resultado_LS) / temperatura))
       {
           Array.Copy(secuencia_final, secuencia_LS, secuencia_final.Length);
           resultado_LS = resultado_final;
       }
   }
   Console.WriteLine($"El mejor resultado de la FO obtenido en IGCmsSepV1:
    {resultado_mejor}\n");

   //Impresion
   FicheroCSV1.Write(alpha + ";" + beta + ";" + mds + ";" + numero_cirujanos + ";" +
    secuencia.Length + ";" + cantidad_camas_PACU + ";" + cantidad_camas_UCI + ";" +
    instancia + ";" + resultado_mejor + ";" + "CSM1V1" + "\n");
}

// ITERATED GREEDY ALGORITHM MÉTODO 2 VERSIÓN 1

static void iterated_greedy_algorithmCSM2V1(int[] secuencia, double[] prioridad, int[]
 fecha_limite, int[,] disponibilidadquirofano_turno, double[] duracion_operacion, int[]
 dia_disponible, int[] cirujano_asociado, double[,] horascirujano_turno, int
 numero_quirofanos, int numero_cirujanos, int dias, int[] necesidad_paciente_cama, int
 cantidad_camas_PACU, int cantidad_camas_UCI, double[] duracion_paciente_cama, double
 alpha, double beta, int mds, int instancia, StreamWriter FicheroCSV1)
```

```
{
    Random random = new Random();
    double numAleat = random.NextDouble();
    if (numAleat == 0.0)
    {
        numAleat = 0.1;
    }

    double temperatura = 0.4;

    //Primero, con la secuencia que llega al algoritmo, se saca la primera solución
     inicial, y se elige la secuencia como la mejor, por el momento
    double[,] info_paciente;
    bool secuencia_valida;
    int cont = 0;
    int[] sec = new int[secuencia.Length];

    do //Bucle para asegurar que sale una secuencia válida antes de entrar al LS
    {
        if (cont != 0)
        {
            fase_destruccion_construccion(secuencia, out sec, 4);
        }
        else
        {
            Array.Copy(secuencia, sec, secuencia.Length);
        }

        decodificacionCS(sec, disponibilidadquirofano_turno, duracion_operacion,
         dia_disponible, cirujano_asociado, horascirujano_turno, numero_quirofanos,
         numero_cirujanos, dias, out info_paciente); //Se decodifica la secuencia
        secuencia_valida = camas_separadas_V1(cantidad_camas_PACU, cantidad_camas_UCI,
         necesidad_paciente_cama, duracion_paciente_cama, info_paciente,
         secuencia.Length);

        if (info_paciente[0, 0] == 1001) { secuencia_valida = false; }
        cont++;
    } while (secuencia_valida == false);

    //Se asigna el resultado como el mejor hasta el momento
    double resultado_mejor = funcion_objetivo(info_paciente, sec, prioridad,
     fecha_limite, dias);
    double resultado_final = resultado_mejor;
    double resultado_LS = resultado_mejor;

    //Se va a guardar también el resultado de la mejor secuencia:
    double[,] mejor_info_paciente = new double[secuencia.Length, 5];
    copiar_matriz(info_paciente, mejor_info_paciente, secuencia.Length, 5);
    double[,] info_paciente_temporal = new double[secuencia.Length, 5];
    copiar_matriz(info_paciente, info_paciente_temporal, secuencia.Length, 5);

    int[] mejor_secuencia = new int[secuencia.Length];
    Array.Copy(sec, mejor_secuencia, sec.Length);
    int[] secuencia_LS = new int[secuencia.Length];
    Array.Copy(mejor_secuencia, secuencia_LS, mejor_secuencia.Length);
    DateTime tiempoFinal = DateTime.Now.AddMilliseconds(secuencia.Length *
     numero_quirofanos * dias / 2 * 25);

    while (DateTime.Now <= tiempoFinal)
    {
        // Primera fase: Destrucción y Construcción
        int[] secuencia_final;
        fase_destruccion_construccion(secuencia_LS, out secuencia_final, 4);

        decodificacionCS(secuencia_final, disponibilidadquirofano_turno,
         duracion_operacion, dia_disponible, cirujano_asociado, horascirujano_turno,
         numero_quirofanos, numero_cirujanos, dias, out info_paciente);
```

```
resultado_final = funcion_objetivo(info_paciente, secuencia_final, prioridad,
 fecha_limite, dias);

if (camas_separadas_V1(cantidad_camas_PACU, cantidad_camas_UCI,
 necesidad_paciente_cama, duracion_paciente_cama, info_paciente,
 secuencia.Length) == false)
 { resultado_final = 1; }

//Segunda fase: Local Search
bool improvement;
do
{
    improvement = false;
    double resultado_nuevo;

    for (int i = 0; i < secuencia_final.Length; i++)
    {
        for (int j = 0; j < secuencia_final.Length; j++)
        {
            bool sec_val;
            int[] secuencia_final_aux = new int[secuencia_final.Length];
            Array.Copy(secuencia_final, secuencia_final_aux,
             secuencia_final.Length);
            int[] nueva_secuencia = new int[secuencia_final.Length];
            int[] secuencia_auxiliar = new int[secuencia_final.Length - 1];
            int trabajo;
            if (i != j && j != i - 1)
            {
                trabajo = sacar_trabajo_LS(secuencia_final_aux,
                 secuencia_auxiliar, i);
                introducir_trabajo_LS(secuencia_auxiliar, nueva_secuencia, j,
                 trabajo);
            }
            else { continue; }

            decodificacionCS(nueva_secuencia, disponibilidadquirofano_turno,
             duracion_operacion, dia_disponible, cirujano_asociado,
             horascirujano_turno, numero_quirofanos, numero_cirujanos, dias, out
             info_paciente);
            sec_val = camas_separadas_V1(cantidad_camas_PACU,
             cantidad_camas_UCI, necesidad_paciente_cama, duracion_paciente_cama,
             info_paciente, secuencia.Length);

            resultado_nuevo = 0;

            if (sec_val)
            {
                resultado_nuevo = funcion_objetivo(info_paciente,
                 nueva_secuencia, prioridad, fecha_limite, dias);
            }

            else
            {
                while (sec_val == false)
                {
                    // Mientras la secuencia no sea válida se perturba la
                     secuencia
                    int[] nueva_secuencia_auxiliar;
                    fase_destruccion_construccion(nueva_secuencia, out
                     nueva_secuencia_auxiliar, 4);
                    decodificacionCS(nueva_secuencia_auxiliar,
                     disponibilidadquirofano_turno, duracion_operacion,
                     dia_disponible, cirujano_asociado, horascirujano_turno,
                     numero_quirofanos, numero_cirujanos, dias, out
                     info_paciente);
                    sec_val = camas_separadas_V1(cantidad_camas_PACU,
```

```
                                cantidad_camas_UCI, necesidad_paciente_cama,
                                duracion_paciente_cama, info_paciente, secuencia.Length);
                            if (sec_val)
                            {
                                resultado_nuevo = funcion_objetivo(info_paciente,
                                 nueva_secuencia_auxiliar, prioridad, fecha_limite,
                                 dias);
                                Array.Copy(nueva_secuencia_auxiliar, nueva_secuencia,
                                 nueva_secuencia.Length);
                            }
                        }
                    }

                    if (resultado_nuevo < resultado_final)
                    {
                        resultado_final = resultado_nuevo;
                        Array.Copy(nueva_secuencia, secuencia_final,
                         nueva_secuencia.Length);
                        copiar_matriz(info_paciente, info_paciente_temporal,
                         secuencia.Length, 5);
                        improvement = true;
                    }
                }
            }
        } while (improvement == true);

        //Simulated Annealing para decidir si la secuencia que sale de LS se emplea en
         la siguiente ronda
        if (resultado_final < resultado_LS)
        {
            Array.Copy(secuencia_final, secuencia_LS, secuencia_final.Length);
            resultado_LS = resultado_final;
            if (resultado_final < resultado_mejor)
            {
                Array.Copy(secuencia_final, mejor_secuencia, secuencia_final.Length);
                resultado_mejor = resultado_final;
                copiar_matriz(info_paciente, mejor_info_paciente, secuencia.Length, 5);
            }
        }
        else if (numAleat <= Math.Exp(-(resultado_final - resultado_LS) / temperatura))
        {
            Array.Copy(secuencia_final, secuencia_LS, secuencia_final.Length);
            resultado_LS = resultado_final;
        }
    }


    Console.WriteLine($"El mejor resultado de la FO obtenido en IGCmsSepMet2V1:
     {resultado_mejor}\n");

    //Impresion
    FicheroCSV1.Write(alpha + ";" + beta + ";" + mds + ";" + numero_cirujanos + ";" +
     secuencia.Length + ";" + cantidad_camas_PACU + ";" + cantidad_camas_UCI + ";" +
     instancia + ";" + resultado_mejor + ";" + "CSM2V1" + "\n");
}

// ITERATED GREEDY ALGORITHM MÉTODO 2 VERSIÓN 2

static void iterated_greedy_algorithmCSM2V2(int[] secuencia, double[] prioridad, int[]
 fecha_limite, int[,] disponibilidadquirofano_turno, double[] duracion_operacion, int[]
 dia_disponible, int[] cirujano_asociado, double[,] horascirujano_turno, int
 numero_quirofanos, int numero_cirujanos, int dias, int[] necesidad_paciente_cama, int
 cantidad_camas_PACU, int cantidad_camas_UCI, double[] duracion_paciente_cama, double
 alpha, double beta, int mds, int instancia, StreamWriter FicheroCSV1)
{
    Random random = new Random();
    double numAleat = random.NextDouble();
```

```
if (numAleat == 0.0)
{
    numAleat = 0.1;
}

double temperatura = 0.4;

//Primero, con la secuencia que llega al algoritmo, se saca la primera solución
 inicial, y se elige la secuencia como la mejor, por el momento
double[,] info_paciente;
bool secuencia_valida;
int cont = 0;
int[] sec = new int[secuencia.Length];

do //Bucle para asegurar que sale una secuencia válida antes de entrar al LS
{
    if (cont != 0)
    {
        fase_destruccion_construccion(secuencia, out sec, 4);
    }
    else
    {
        Array.Copy(secuencia, sec, secuencia.Length);
    }

    decodificacionCS(sec, disponibilidadquirofano_turno, duracion_operacion,
     dia_disponible, cirujano_asociado, horascirujano_turno, numero_quirofanos,
     numero_cirujanos, dias, out info_paciente); //Se decodifica la secuencia
    secuencia_valida = camas_separadas_V2(cantidad_camas_PACU, cantidad_camas_UCI,
     necesidad_paciente_cama, duracion_paciente_cama, info_paciente,
     secuencia.Length);

    if (info_paciente[0, 0] == 1001) { secuencia_valida = false; }
    cont++;

} while (secuencia_valida == false);

// Se asigna el resultado como el mejor hasta el momento
double resultado_mejor = funcion_objetivo(info_paciente, sec, prioridad,
 fecha_limite, dias);
double resultado_final = resultado_mejor;
double resultado_LS = resultado_mejor;

// Se va a guardar también el resultado de la mejor secuencia:
double[,] mejor_info_paciente = new double[secuencia.Length, 5];
copiar_matriz(info_paciente, mejor_info_paciente, secuencia.Length, 5);
double[,] info_paciente_temporal = new double[secuencia.Length, 5];
copiar_matriz(info_paciente, info_paciente_temporal, secuencia.Length, 5);

// Empieza la búsqueda local usando first-best improvement
int[] mejor_secuencia = new int[secuencia.Length];
Array.Copy(sec, mejor_secuencia, sec.Length);
int[] secuencia_LS = new int[secuencia.Length];
Array.Copy(mejor_secuencia, secuencia_LS, mejor_secuencia.Length);
DateTime tiempoFinal = DateTime.Now.AddMilliseconds(secuencia.Length *
 numero_quirofanos * dias / 2 * 25);

while (DateTime.Now <= tiempoFinal)
{
    // Primera fase: Destrucción y Construcción
    int[] secuencia_final;
    fase_destruccion_construccion(secuencia_LS, out secuencia_final, 4);

    decodificacionCS(secuencia_final, disponibilidadquirofano_turno,
     duracion_operacion, dia_disponible, cirujano_asociado, horascirujano_turno,
     numero_quirofanos, numero_cirujanos, dias, out info_paciente);
```

```
resultado_final = funcion_objetivo(info_paciente, secuencia_final, prioridad,
 fecha_limite, dias);

if (camas_separadas_V2(cantidad_camas_PACU, cantidad_camas_UCI,
 necesidad_paciente_cama, duracion_paciente_cama, info_paciente,
 secuencia.Length) == false)
 { resultado_final = 1; }

//Segunda fase: Local Search
bool improvement;
do
{
    improvement = false;
    double resultado_nuevo;

    for (int i = 0; i < secuencia_final.Length; i++)
    {
        for (int j = 0; j < secuencia_final.Length; j++)
        {
            bool sec_val;
            int[] secuencia_final_aux = new int[secuencia_final.Length];
            Array.Copy(secuencia_final, secuencia_final_aux,
             secuencia_final.Length);
            int[] nueva_secuencia = new int[secuencia_final.Length];
            int[] secuencia_auxiliar = new int[secuencia_final.Length - 1];
            int trabajo;
            if (i != j && j != i - 1)
            {
                trabajo = sacar_trabajo_LS(secuencia_final_aux,
                 secuencia_auxiliar, i);
                introducir_trabajo_LS(secuencia_auxiliar, nueva_secuencia, j,
                 trabajo);
            }
            else { continue; }

            decodificacionCS(nueva_secuencia, disponibilidadquirofano_turno,
             duracion_operacion, dia_disponible, cirujano_asociado,
             horascirujano_turno, numero_quirofanos, numero_cirujanos, dias, out
             info_paciente);
            sec_val = camas_separadas_V2(cantidad_camas_PACU,
             cantidad_camas_UCI, necesidad_paciente_cama, duracion_paciente_cama,
             info_paciente, secuencia.Length);

            resultado_nuevo = 0;

            if (sec_val)
            {
                resultado_nuevo = funcion_objetivo(info_paciente,
                 nueva_secuencia, prioridad, fecha_limite, dias);
            }

            else
            {
                while (sec_val == false)
                {
                    // Mientras la secuencia no sea válida se perturba la
                     secuencia
                    int[] nueva_secuencia_auxiliar;
                    fase_destruccion_construccion(nueva_secuencia, out
                     nueva_secuencia_auxiliar, 4);
                    decodificacionCS(nueva_secuencia_auxiliar,
                     disponibilidadquirofano_turno, duracion_operacion,
                     dia_disponible, cirujano_asociado, horascirujano_turno,
                     numero_quirofanos, numero_cirujanos, dias, out
                     info_paciente);
                    sec_val = camas_separadas_V2(cantidad_camas_PACU,
                     cantidad_camas_UCI, necesidad_paciente_cama,
```

```
                                duracion_paciente_cama, info_paciente, secuencia.Length);
                            if (sec_val)
                            {
                                resultado_nuevo = funcion_objetivo(info_paciente,
                                 nueva_secuencia_auxiliar, prioridad, fecha_limite,
                                 dias);
                                Array.Copy(nueva_secuencia_auxiliar, nueva_secuencia,
                                 nueva_secuencia.Length);
                            }
                        }
                    }

                    if (resultado_nuevo < resultado_final)
                    {
                        resultado_final = resultado_nuevo;
                        Array.Copy(nueva_secuencia, secuencia_final,
                         nueva_secuencia.Length);
                        copiar_matriz(info_paciente, info_paciente_temporal,
                         secuencia.Length, 5);
                        improvement = true;
                    }
                }
            }
        } while (improvement == true);

        //Simulated Annealing para decidir si la secuencia que sale de LS se emplea en
         la siguiente ronda
        if (resultado_final < resultado_LS)
        {
            Array.Copy(secuencia_final, secuencia_LS, secuencia_final.Length);
            resultado_LS = resultado_final;
            if (resultado_final < resultado_mejor)
            {
                Array.Copy(secuencia_final, mejor_secuencia, secuencia_final.Length);
                resultado_mejor = resultado_final;
                copiar_matriz(info_paciente, mejor_info_paciente, secuencia.Length, 5);
            }
        }
        else if (numAleat <= Math.Exp(-(resultado_final - resultado_LS) / temperatura))
        {
            Array.Copy(secuencia_final, secuencia_LS, secuencia_final.Length);
            resultado_LS = resultado_final;
        }
    }


    Console.WriteLine($"El mejor resultado de la FO obtenido en IGCmsSepMet2V2:
     {resultado_mejor}\n");

    //Impresion
    FicheroCSV1.Write(alpha + ";" + beta + ";" + mds + ";" + numero_cirujanos + ";" +
     secuencia.Length + ";" + cantidad_camas_PACU + ";" + cantidad_camas_UCI + ";" +
     instancia + ";" + resultado_mejor + ";" + "CSM2V2" + "\n");
}

//Funciones para las camas

static bool camas_separadas_V1(int camas_PACU, int camas_UCI, int[]
 necesidad_paciente_cama, double[] duracion_paciente_cama, double[,] info_paciente, int
 pacientes_totales)
{
    double[,] tiempos_camas_PACU = new double[camas_PACU, 2];
    double[,] tiempos_camas_UCI = new double[camas_UCI, 2];
    setval_IMatriz(tiempos_camas_PACU, camas_PACU, 2, 0);
    setval_IMatriz(tiempos_camas_UCI, camas_UCI, 2, 0);
    bool secuencia_valida = true;
```

```
    for (int i = 0; i < pacientes_totales; i++)
    {
        if (info_paciente[i, 0] != 1001)
        {
            if (necesidad_paciente_cama[Convert.ToInt32(info_paciente[i, 0])] == 0) {
             continue; }
            else if (necesidad_paciente_cama[Convert.ToInt32(info_paciente[i, 0])] == 1)
            {
                int paciente_asignado = 0;
                for (int j = 0; j < camas_PACU; j++)
                {
                    if (tiempos_camas_PACU[j, 0] <= info_paciente[i, 2] ||
                     (tiempos_camas_PACU[j, 0] == info_paciente[i, 2]
                        && tiempos_camas_PACU[j, 1] <= info_paciente[i, 4]))
                    {
                        duracion_a_turnos_horas(info_paciente[i, 2], info_paciente[i,
                         4], duracion_paciente_cama, Convert.ToInt32(info_paciente[i,
                         0]), tiempos_camas_PACU, j);
                        paciente_asignado = 1;
                        break;
                    }
                }
                if (paciente_asignado == 0)
                {
                    secuencia_valida = false;
                    break;
                }
            }
            else
            {
                int paciente_asignado = 0;
                for (int j = 0; j < camas_UCI; j++)
                {
                    if (tiempos_camas_UCI[j, 0] <= info_paciente[i, 2] ||
                     (tiempos_camas_UCI[j, 0] == info_paciente[i, 2] &&
                     tiempos_camas_UCI[j, 1] <= info_paciente[i, 4]))
                    {
                        duracion_a_turnos_horas(info_paciente[i, 2], info_paciente[i,
                         4], duracion_paciente_cama, Convert.ToInt32(info_paciente[i,
                         0]), tiempos_camas_UCI, j);
                        paciente_asignado = 1;
                        break;
                    }
                }
                if (paciente_asignado == 0)
                {
                    secuencia_valida = false;
                    break;
                }
            }
        }
        else { break; }
    }
    return secuencia_valida;
}

static bool camas_separadas_V2(int camas_PACU, int camas_UCI, int[]
 necesidad_paciente_cama, double[] duracion_paciente_cama, double[,] info_paciente, int
 pacientes_totales)
{
    double[,] tiempos_camas_PACU = new double[camas_PACU, 2];
    double[,] tiempos_camas_UCI = new double[camas_UCI, 2];
    setval_IMatriz(tiempos_camas_PACU, camas_PACU, 2, 0);
    setval_IMatriz(tiempos_camas_UCI, camas_UCI, 2, 0);

    bool secuencia_valida = true;
```

```
for (int i = 0; i < pacientes_totales; i++)
{
    if (info_paciente[i, 0] != 1001)
    {
        if (necesidad_paciente_cama[Convert.ToInt32(info_paciente[i, 0])] == 0) {
         continue; }
        else if (necesidad_paciente_cama[Convert.ToInt32(info_paciente[i, 0])] == 1)
        {
            int paciente_asignado = 0;
            for (int j = 0; j < camas_PACU; j++)
            {
                if (tiempos_camas_PACU[j, 0] <= info_paciente[i, 2] ||
                 (tiempos_camas_PACU[j, 0] == info_paciente[i, 2] &&
                 tiempos_camas_PACU[j, 1] <= info_paciente[i, 4]))
                {
                    duracion_a_turnos_horas(info_paciente[i, 2], info_paciente[i,
                     4], duracion_paciente_cama, Convert.ToInt32(info_paciente[i,
                     0]), tiempos_camas_PACU, j);
                    paciente_asignado = 1;
                    break;
                }
            }
            if (paciente_asignado == 0)
            {
                int proximo_paciente = proximo_paciente_en_quirofano(i,
                 info_paciente, pacientes_totales);

                if (proximo_paciente != -1)
                {
                    double minimo_turno = 0;
                    double minima_hora = 0;
                    int primera_cama = 0;
                    for (int r = 0; r < camas_PACU; r++)
                    {
                        //Se busca la primera cama disponible, junto al turno en el
                         que está disponible y la hora
                        if (r == 0)
                        {
                            minimo_turno = tiempos_camas_PACU[r, 0];
                            minima_hora = tiempos_camas_PACU[r, 1];
                        }
                        else
                        {
                            if (tiempos_camas_PACU[r, 0] < minimo_turno ||
                             (tiempos_camas_PACU[r, 0] == minimo_turno &&
                             tiempos_camas_PACU[r, 1] <= minima_hora))
                            {
                                minimo_turno = tiempos_camas_PACU[r, 0];
                                minima_hora = tiempos_camas_PACU[r, 1];
                                primera_cama = r;
                            }
                        }
                    }
                    if (minimo_turno < info_paciente[proximo_paciente, 2] ||
                     (minimo_turno == info_paciente[proximo_paciente, 2] &&
                      minima_hora <= info_paciente[proximo_paciente, 4]))
                    {
                        //Ahora comprobar que el tiempo de arriba no sea mayor al
                         tiempo que el paciente tiene asignado en CAMA
                        duracion_a_turnos_horas_V2(info_paciente[i, 2],
                         info_paciente[i, 4], duracion_paciente_cama,
                         Convert.ToInt32(info_paciente[i, 0]), tiempos_camas_PACU,
                         primera_cama);
                    }
                    else
                    {
                        secuencia_valida = false;
```

```
                    break;
                }
            }
            // Si no hay pacientes después en el mismo quirófano, se supone
             siempre que se va a poder quedar ahi
        }
    }
    else
    {
        int paciente_asignado = 0;
        for (int j = 0; j < camas_UCI; j++)
        {
            if (tiempos_camas_UCI[j, 0] <= info_paciente[i, 2] ||
             (tiempos_camas_UCI[j, 0] == info_paciente[i, 2] &&
             tiempos_camas_UCI[j, 1] <= info_paciente[i, 4]))
            {
                duracion_a_turnos_horas(info_paciente[i, 2], info_paciente[i,
                 4], duracion_paciente_cama, Convert.ToInt32(info_paciente[i,
                 0]), tiempos_camas_UCI, j);
                paciente_asignado = 1;
                break;
            }
        }
        if (paciente_asignado == 0)
        {
            int proximo_paciente = proximo_paciente_en_quirofano(i,
             info_paciente, pacientes_totales);

            if (proximo_paciente != -1)
            {
                double minimo_turno = 0;
                double minima_hora = 0;
                int primera_cama = 0;
                for (int r = 0; r < camas_UCI; r++)
                {

                    if (r == 0)
                    {
                        minimo_turno = tiempos_camas_UCI[r, 0];
                        minima_hora = tiempos_camas_UCI[r, 1];
                    }
                    else
                    {
                        if (tiempos_camas_UCI[r, 0] < minimo_turno ||
                         (tiempos_camas_UCI[r, 0] == minimo_turno &&
                         tiempos_camas_UCI[r, 1] == minima_hora))
                        {
                            minimo_turno = tiempos_camas_UCI[r, 0];
                            minima_hora = tiempos_camas_UCI[r, 1];
                            primera_cama = r;
                        }
                    }
                }
                if (minimo_turno < info_paciente[proximo_paciente, 2] ||
                 (minimo_turno == info_paciente[proximo_paciente, 2] &&
                 minima_hora <= info_paciente[proximo_paciente, 3]))
                {
                    // Se mete al paciente en la cama cuando se quede libre la
                     primera, o acaba su tiempo en quirófano si esto ocurre antes
                     de que una de las camas quede libre
                    duracion_a_turnos_horas_V2(info_paciente[i, 2],
                     info_paciente[i, 4], duracion_paciente_cama,
                     Convert.ToInt32(info_paciente[i, 0]), tiempos_camas_UCI,
                     primera_cama);
                }
                else
                {
```

```
                                secuencia_valida = false;
                                break;
                            }
                        }
                    }
                }
            }
            else { break; }
        }
        return secuencia_valida;
    }


    // FUNCIÓN OBJETIVO: minimizar la tardanza ponderada con la prioridad

    static double funcion_objetivo(double[,] info_paciente, int[] secuencia, double[]
     prioridad, int[] fecha_limite, int dias)
    {
        double funcion_objetivo = 0; // Acumuladas de todos los pacientes que han ido
         pasando
        double tardanza_ponderada; // De cada uno de los pacientes
        for (int i = 0; i < secuencia.Length; i++)
        {
            //Lo primero que se necesita saber es si el paciente se ha operado, si lo ha
             sido, debe tener la misma posición en secuencia que en info_paciente:
            int paciente_operado = 0;
            if (secuencia[i] == info_paciente[i, 0])
            {
                paciente_operado = 1;
            }

            // Ahora se distingue entre si los pacientes se han operado o no para asignar
             los días de operación
            int dia_operacion;
            if (paciente_operado == 1)
            {
                if (info_paciente[i, 2] == 0) { dia_operacion = 0; }
                else
                {
                    dia_operacion = Convert.ToInt32(Math.Round(info_paciente[i, 2] / 2,
MidpointRounding.AwayFromZero));
                }
            }
            else { dia_operacion = dias + 1; }


            // Una vez asignados los días de operación se puede ver si los pacientes van
             tarde. Se ha supuesto que, si los pacientes no han sido operados en el último
             día del horizonte temporal, estos van a ser operados el día siguiente.
            // Esto solo tiene importancia para los pacientes con fecha límite dentro del
             horizonte temporal.
            double tardanza = dia_operacion - fecha_limite[Convert.ToInt32(secuencia[i])];
            if (tardanza < 0) { tardanza_ponderada = 0; }
            else
            {
                tardanza_ponderada = tardanza / dias;
            }

            //Calculo la funcion objetivo sumando los resultados de los pacientes y
             ponderando con la prioridad de estos
            funcion_objetivo += (tardanza_ponderada - paciente_operado) *
             prioridad[secuencia[i]];
        }
        return funcion_objetivo;
    }

    // Pasar las horas de las camas a turnos y horas
```

```
static void duracion_a_turnos_horas(double turno_operacion, double
 hora_finalizacion_operacion, double[] duracion_paciente_cama, int paciente, double[,]
 tiempos_camas, int cama_actual)
{
    double auxiliar;
    int tipo;

    if (turno_operacion == 0 || turno_operacion % 2 == 0) //Significa que estoy por la
     mañana
    {
        auxiliar = 8.5 + hora_finalizacion_operacion + duracion_paciente_cama[paciente];
        tipo = 1;
    }
    else //Significa que es por la tarde
    {
        auxiliar = 15.5 + hora_finalizacion_operacion +
         duracion_paciente_cama[paciente];
        tipo = 2;
    }
    tiempos_camas[cama_actual, 0] = turno_operacion;
    double x = auxiliar / 24;
    if (tipo == 1) //Estoy colocándome en el primer turno del día al que voy
    {
        tiempos_camas[cama_actual, 0] += Math.Floor(x) * 2;
    }
    else
    {
        tiempos_camas[cama_actual, 0] += 2 * Math.Floor(x) - 1;
    }
    double y = auxiliar - 24 * Math.Floor(x);
    if (y < 8.5)
    {
        tiempos_camas[cama_actual, 1] = 0;
    }
    else if (y >= 8.5 && y < 14.5)
    {
        tiempos_camas[cama_actual, 1] = y - 8.5;
    }
    else if (y >= 14.5 && y < 15.5)
    {
        tiempos_camas[cama_actual, 0]++;
        tiempos_camas[cama_actual, 1] = 0;
    }
    else if (y >= 15.5 && y <= 19.5)
    {
        tiempos_camas[cama_actual, 0]++;
        tiempos_camas[cama_actual, 1] = y - 15.5;
    }
    else
    {
        tiempos_camas[cama_actual, 0] += 2;
        tiempos_camas[cama_actual, 1] = 0;
    }
}

static void duracion_a_turnos_horas_V2(double turno_operacion, double
 hora_finalizacion_operacion, double[] duracion_paciente_cama, int paciente, double[,]
 tiempos_camas, int cama)
{
    double auxiliar; //Hora a la que acabaría el tiempo en cama (puede pasar de las
     24horas)
    int tipo;
    double turno_auxiliar; //Turno en el que acabaria la estancia del paciente en la
     cama
    double hora_auxiliar; //Hora a la que acabaria la estancia del paciente en la cama
```

```
if (turno_operacion == 0 || turno_operacion % 2 == 0) //Significa que estoy por la
 mañana
{
    auxiliar = 8.5 + hora_finalizacion_operacion + duracion_paciente_cama[paciente];
    tipo = 1;
}
else //Significa que es por la tarde
{
    auxiliar = 15.5 + hora_finalizacion_operacion +
     duracion_paciente_cama[paciente];
    tipo = 2;
}

turno_auxiliar = turno_operacion;
double x = auxiliar / 24; //Para conocer si ha pasado más de un día en la cama
if (tipo == 1)
{
    turno_auxiliar += Math.Floor(x) * 2; //La función Floor se usa para redondear
     hacia abajo
}
else
{
    turno_auxiliar += 2 * Math.Floor(x) - 1;
}
double y = auxiliar - 24 * Math.Floor(x); //Para las horas después de ver si ha
 pasado más de un día en la cama
if (y < 8.5)
{
    hora_auxiliar = 0;
}
else if (y >= 8.5 && y <= 14.5)
{
    hora_auxiliar = y - 8.5;
}
else if (y > 14.5 && y < 15.5)
{
    turno_auxiliar++;
    hora_auxiliar = 0;
}
else if (y >= 15.5 && y <= 19.5)
{
    turno_auxiliar++;
    hora_auxiliar = y - 15.5;
}
else
{
    turno_auxiliar += 2;
    hora_auxiliar = 0;
}

// En este punto esta guardado el turno y la hora a la que el paciente saldría de
 cama si entrase tras acabar operación.
// Ahora hay que hacer la comparación entre este momento y el momento en el que la
 cama se queda libre
if (turno_auxiliar > tiempos_camas[cama, 0] || (turno_auxiliar ==
 tiempos_camas[cama, 0] && hora_auxiliar > tiempos_camas[cama, 1]))
{
    // Recalculo el tiempo que pasa en la cama y actualizo las variables de tiempo
     de la cama
    // El paciente saldráa de cama en turno_auxiliar y hora_auxiliar
    tiempos_camas[cama, 0] = turno_auxiliar;
    tiempos_camas[cama, 1] = hora_auxiliar;
}
// Si no se da el if anterior, significa que el paciente sale antes de que haya cama
 libre y antes de que otro paciente entre en quirófano, no actualizo ninguna
 variable
```

```
}

// Encontrar al próximo paciente en el quirófano (para la Versión 2)

static int proximo_paciente_en_quirofano(int indice, double[,] info_paciente, int
 pacientes_totales)
{
    double quirofano = info_paciente[indice, 1];
    int indice_devuelto = -1;
    int auxiliar = 0;

    for (int x = indice + 1; x < pacientes_totales; x++)
    {
        if (auxiliar == 0)
        {
            if (info_paciente[x, 1] == quirofano)
            {
                indice_devuelto = x;
            }
        }
        else { break; }
    }
    return indice_devuelto;
}

// Igualar tiempos auxiliares con generales

static void igualar_tiempos_auxiliares_a_generales(double[,] tiempos_quirofanos,
 double[,] tiempos_quirofanos_auxiliar, double[,] tiempos_cirujanos, double[,]
 tiempos_cirujanos_auxiliar, double[,] tiempos_camas_PACU, double[,]
 tiempos_camas_PACU_auxiliar, double[,] tiempos_camas_UCI, double[,]
 tiempos_camas_UCI_auxiliar, int cantidad_camas_PACU, int cantidad_camas_UCI, int[]
 secuencia, int orden, int[] cirujano_asociado,
    int quirofano)
{
    tiempos_quirofanos_auxiliar[quirofano, 0] = tiempos_quirofanos[quirofano, 0];
    tiempos_quirofanos_auxiliar[quirofano, 1] = tiempos_quirofanos[quirofano, 1];
    tiempos_cirujanos_auxiliar[cirujano_asociado[secuencia[orden]], 0] =
     tiempos_cirujanos[cirujano_asociado[secuencia[orden]], 0];
    tiempos_cirujanos_auxiliar[cirujano_asociado[secuencia[orden]], 1] =
     tiempos_cirujanos[cirujano_asociado[secuencia[orden]], 1];

    for (int x = 0; x < cantidad_camas_PACU; x++)
    {
        tiempos_camas_PACU_auxiliar[x, 0] = tiempos_camas_PACU[x, 0];
        tiempos_camas_PACU_auxiliar[x, 1] = tiempos_camas_PACU[x, 1];
    }
    for (int w = 0; w < cantidad_camas_UCI; w++)
    {
        tiempos_camas_UCI_auxiliar[w, 0] = tiempos_camas_UCI[w, 0];
        tiempos_camas_UCI_auxiliar[w, 1] = tiempos_camas_UCI[w, 1];
    }
}

// Actualizar los tiempos al día disponible e igualar los momentos en los que se
 encuentra el cirujano y el quirófano (a nivel auxiliar para comenzar)

static void release_y_tiempos_quirofano_cirujano(int[] dia_disponible, int[] secuencia,
 int orden, double[,] tiempos_quirofanos_auxiliar, double[,] tiempos_cirujanos_auxiliar,
 int[] cirujano_asociado, int quirofano)
{
    if (dia_disponible[secuencia[orden]] > tiempos_quirofanos_auxiliar[quirofano, 0])
    //Actualizo el tiempo del quirófano al momento en el que llega el paciente
    {
        tiempos_quirofanos_auxiliar[quirofano, 0] = dia_disponible[secuencia[orden]];
        tiempos_quirofanos_auxiliar[quirofano, 1] = 0;
    }
```

```
    //Actualizo el tiempo quirófano o cirujano, según cuál vaya más adelantado
    if (tiempos_quirofanos_auxiliar[quirofano, 0] <
     tiempos_cirujanos_auxiliar[cirujano_asociado[secuencia[orden]], 0])
    {
        tiempos_quirofanos_auxiliar[quirofano, 0] =
         tiempos_cirujanos_auxiliar[cirujano_asociado[secuencia[orden]], 0];
        tiempos_quirofanos_auxiliar[quirofano, 1] =
         tiempos_cirujanos_auxiliar[cirujano_asociado[secuencia[orden]], 1];
    }
    else if (tiempos_quirofanos_auxiliar[quirofano, 0] >
      tiempos_cirujanos_auxiliar[cirujano_asociado[secuencia[orden]], 0])
    {
        tiempos_cirujanos_auxiliar[cirujano_asociado[secuencia[orden]], 0] =
         tiempos_quirofanos_auxiliar[quirofano, 0];
        tiempos_cirujanos_auxiliar[cirujano_asociado[secuencia[orden]], 1] =
         tiempos_quirofanos_auxiliar[quirofano, 1];
    }
}

// Necesidad de camas

static void comprueba_necesidad_camas(int[] necesidad_paciente_cama, int[] secuencia,
 int orden, out bool resultado_camas, int cantidad_camas_PACU, int cantidad_camas_UCI,
 double[,] tiempos_camas_UCI, double[,] tiempos_camas_PACU, int turno)
{
    if (necesidad_paciente_cama[secuencia[orden]] == 1)
    {
        resultado_camas = cama_en_turno(cantidad_camas_PACU, tiempos_camas_PACU, turno);
    }
    else if (necesidad_paciente_cama[secuencia[orden]] == 2)
    {
        resultado_camas = cama_en_turno(cantidad_camas_UCI, tiempos_camas_UCI, turno);
    }
    else
    {
        resultado_camas = true;
    }
}

// Búsqueda de la cama necesaria
static void busqueda_cama(double[,] tiempos_camas_auxiliar, double[,]
 tiempos_quirofanos_auxiliar, double tiempo_inicio_operacion, double[]
 duracion_operacion, int[] secuencia, int i, int[] cama_asociada_quirofano, double
 mejor_turno_cama, double mejor_hora_cama, int quirofano_actual, int cantidad_camas, out
 double mejor_turno_cama_out, out double mejor_hora_cama_out)
{
    int y;
    for (y = 0; y < cantidad_camas; y++)
    {
        // Si esto se cumple, no hace falta seguir buscando
        if (tiempos_camas_auxiliar[y, 0] < tiempos_quirofanos_auxiliar[quirofano_actual,
         0] || (tiempos_camas_auxiliar[y, 0] ==
         tiempos_quirofanos_auxiliar[quirofano_actual, 0] && tiempos_camas_auxiliar[y,
         1] <= tiempo_inicio_operacion + duracion_operacion[secuencia[i]]))
        {
            cama_asociada_quirofano[quirofano_actual] = y;
            mejor_turno_cama = tiempos_camas_auxiliar[y, 0];
            mejor_hora_cama = tiempos_camas_auxiliar[y, 1];
            break;
        }

        else
        {
            if (y == 0)
            {
                mejor_turno_cama = tiempos_camas_auxiliar[y, 0];
                mejor_hora_cama = tiempos_camas_auxiliar[y, 1];
```

```
                cama_asociada_quirofano[quirofano_actual] = y;
            }
            if (tiempos_camas_auxiliar[y, 0] < mejor_turno_cama)
            {
                mejor_turno_cama = tiempos_camas_auxiliar[y, 0];
                mejor_hora_cama = tiempos_camas_auxiliar[y, 1];
                cama_asociada_quirofano[quirofano_actual] = y;
            }
            else if (tiempos_camas_auxiliar[y, 0] == mejor_turno_cama)
            {
                if (tiempos_camas_auxiliar[y, 1] < mejor_hora_cama)
                {
                    mejor_hora_cama = tiempos_camas_auxiliar[y, 1];
                    cama_asociada_quirofano[quirofano_actual] = y;
                }
            }
        }
    }
    mejor_turno_cama_out = mejor_hora_cama;
    mejor_hora_cama_out = mejor_hora_cama;

}

// Coloca los tiempos de quirófanos y cirujanos temporalmente

static void asignacion_temporal_camas(double[,] tiempos_quirofanos_auxiliar, double
 mejor_turno_cama, double mejor_hora_cama, double tiempo_inicio_operacion, int[]
 paciente_incluido, int[] secuencia, int i, int paciente_entraria_en_quirofano, int[]
 cama_asociada_quirofano, double[] duracion_operacion, double[,]
 tiempos_cirujanos_auxiliar, int[] cirujano_asociado, double tiempo_maximo, int
 quirofano_actual, int turno_actual, out int paciente_entraria_en_quirofano_out)
{
    if (mejor_turno_cama < tiempos_quirofanos_auxiliar[quirofano_actual, 0] ||
     (mejor_turno_cama == tiempos_quirofanos_auxiliar[quirofano_actual, 0] &&
     mejor_hora_cama <= tiempo_inicio_operacion + duracion_operacion[secuencia[i]]))
    {
        tiempos_quirofanos_auxiliar[quirofano_actual, 1] = tiempo_inicio_operacion +
         duracion_operacion[secuencia[i]];
        paciente_incluido[secuencia[i]] = 1;
        paciente_entraria_en_quirofano = 1;
    }
    else if (mejor_turno_cama > tiempos_quirofanos_auxiliar[quirofano_actual, 0])
    {
        tiempos_quirofanos_auxiliar[quirofano_actual, 0]++;
        tiempos_quirofanos_auxiliar[quirofano_actual, 1] = 0;
        tiempos_cirujanos_auxiliar[cirujano_asociado[secuencia[i]], 0]++;
        tiempos_cirujanos_auxiliar[cirujano_asociado[secuencia[i]], 1] = 0;
    }
    else if (mejor_turno_cama == tiempos_quirofanos_auxiliar[quirofano_actual, 0] &&
     mejor_hora_cama > tiempo_inicio_operacion + duracion_operacion[secuencia[i]])
    {
        if (mejor_hora_cama + duracion_operacion[secuencia[i]] <= tiempo_maximo)
        {
            tiempo_inicio_operacion = mejor_hora_cama;
            tiempos_quirofanos_auxiliar[quirofano_actual, 1] = tiempo_inicio_operacion +
             duracion_operacion[secuencia[i]];
            paciente_incluido[secuencia[i]] = 1;
            paciente_entraria_en_quirofano = 1;
        }
        else
        {//No daría tiempo de completarse la operacion en el turno actual
            tiempos_quirofanos_auxiliar[quirofano_actual, 0]++;
            tiempos_quirofanos_auxiliar[quirofano_actual, 1] = 0;
            tiempos_cirujanos_auxiliar[cirujano_asociado[secuencia[i]], 0]++;
            tiempos_cirujanos_auxiliar[cirujano_asociado[secuencia[i]], 1] = 0;
        }
    }
```

```
    paciente_entraria_en_quirofano_out = paciente_entraria_en_quirofano;
}

// Comprueba si hay camas en el turno

static bool cama_en_turno(int numero_camas, double[,] tiempo_camas, int turno_actual)
{
    bool resultado = false;
    for (int t = 0; t < numero_camas; t++)
    {
        if (tiempo_camas[t, 0] <= turno_actual)
        {
            resultado = true;
            break;
        }
    }
    return resultado;
}

static void duracion_a_turnos_horasCI(double turno_operacion, double
 hora_finalizacion_operacion, double[] duracion_paciente_cama, int paciente, double[,]
 tiempos_camas, int[] cama_asociada_quirofano, int quirofano_asociado)
{
    double auxiliar;
    int tipo;

    if (turno_operacion == 0 || turno_operacion % 2 == 0) //Significa que estoy por la
     mañana
    {
        auxiliar = 8.5 + hora_finalizacion_operacion + duracion_paciente_cama[paciente];
        tipo = 1;
    }
    else //Significa que es por la tarde
    {
        auxiliar = 15 + hora_finalizacion_operacion + duracion_paciente_cama[paciente];
        tipo = 2;
    }
    tiempos_camas[cama_asociada_quirofano[quirofano_asociado], 0] = turno_operacion;
    double x = auxiliar / 24;
    if (tipo == 1) //Estoy colocandome en el primer turno del día al que voy
    {
        tiempos_camas[cama_asociada_quirofano[quirofano_asociado], 0] += Math.Floor(x) *
         2;
    }
    else
    {
        tiempos_camas[cama_asociada_quirofano[quirofano_asociado], 0] += 2 *
         Math.Floor(x) - 1;
    }
    double y = auxiliar - 24 * Math.Floor(x);
    if (y < 8.5)
    {
        tiempos_camas[cama_asociada_quirofano[quirofano_asociado], 1] = 0;
    }
    else if (y >= 8.5 && y < 14.5)
    {
        tiempos_camas[cama_asociada_quirofano[quirofano_asociado], 1] = y - 8.5;
    }
    else if (y >= 14.5 && y < 15.5)
    {
        tiempos_camas[cama_asociada_quirofano[quirofano_asociado], 0]++;
        tiempos_camas[cama_asociada_quirofano[quirofano_asociado], 1] = 0;
    }
    else if (y >= 15.5 && y <= 19.5)
    {
        tiempos_camas[cama_asociada_quirofano[quirofano_asociado], 0]++;
        tiempos_camas[cama_asociada_quirofano[quirofano_asociado], 1] = y - 15;
```

```
    }
    else
    {
        tiempos_camas[cama_asociada_quirofano[quirofano_asociado], 0] += 2;
        tiempos_camas[cama_asociada_quirofano[quirofano_asociado], 1] = 0;
    }
}

//Ordenar de menor a mayor

static void orden_mayor_menor_y_secuencia(double[] vector, int length, int[] secuencia)
{
    double auxiliar;
    int auxiliar2;
    for (int i = 0; i < length; i++)
    {
        for (int j = 0; j < length; j++)
        {
            if (vector[i] > vector[j])
            {
                auxiliar = vector[j];
                auxiliar2 = secuencia[j];
                vector[j] = vector[i];
                secuencia[j] = secuencia[i];
                vector[i] = auxiliar;
                secuencia[i] = auxiliar2;
            }
        }
    }
}

//Rellenar matriz con valores iguales

static void setval_IMatriz(double[,] matriz, int filas, int columnas, double val)
{
    for (int i = 0; i < filas; i++)
    {
        for (int j = 0; j < columnas; j++)
        {
            matriz[i, j] = val;
        }
    }
}

//Rellenar vector con valores iguales

static void setval_IVector(int[] vector, int val)
{
    for (int i = 0; i < vector.Length; i++)
    {
        vector[i] = val;
    }
}

static void setval_IVector_double(double[] vector, int val)
{
    for (int i = 0; i < vector.Length; i++)
    {
        vector[i] = val;
    }
}

//Fijar el tiempo máximo de un turno

static void tiempo_maximo_turno(int turno, out double tiempo_maximo)
{
    if (turno == 0 || turno % 2 == 0)
```

```
    {
        tiempo_maximo = 6.5;
    }
    else { tiempo_maximo = 5; }
}

//Funcion de destrucción

static void sacar_trabajos(int[] secuencia_original, out int[] secuencia_destino, int[]
 secuencia_destruccion, int long_cad_destino)
{
    secuencia_destino = new int[long_cad_destino];

    int[] secuenciaO = new int[secuencia_original.Length];
    Array.Copy(secuencia_original, secuenciaO, secuenciaO.Length);
    int[] secuenciaI = new int[secuencia_original.Length - 1];

    for (int i = 0; i < secuencia_destruccion.Length; i++)
    {
        int trabajo = secuencia_destruccion[i];
        int posicion_trabajo = 0;
        int contador = 0;
        for (int j = 0; j < secuenciaO.Length; j++)
        {
            if (trabajo == secuenciaO[j])
            {
                posicion_trabajo = contador;
                break;
            }
            contador++;
        }
        sacar_trabajo_LS(secuenciaO, secuenciaI, posicion_trabajo);
        Array.Resize(ref secuenciaO, secuenciaO.Length - 1);
        Array.Copy(secuenciaI, secuenciaO, secuenciaI.Length);
        Array.Resize(ref secuenciaI, secuenciaI.Length - 1);
    }
    Array.Copy(secuenciaO, secuencia_destino, secuenciaO.Length);
}

//Secuencia aleatoria

static void secuencia_diferente_aleatoria(int[] secuencia_nueva, int long_sec_general)
{
    Random aleatorio = new Random();
    for (int j = 0; j < secuencia_nueva.Length; j++)
    {
        bool diferente = false;
        while (diferente == false) //Comprobación de que el último número aleatorio no
         coincide con otros ya cogidos
        {
            secuencia_nueva[j] = aleatorio.Next(0, long_sec_general); //Se asigna un
             número aleatorio a la nueva secuencia
            int contador = 0;
            for (int x = 0; x < j; x++) //He copiado lo de dentro del for
            {
                if (secuencia_nueva[j] != secuencia_nueva[x])
                {
                    contador++;
                }
                else if (secuencia_nueva[j] == secuencia_nueva[x])
                {
                    contador = 0;
                }
            }
            if (contador == j) { diferente = true; }
        }
    }
```

```
}

//Insercion

static void insercion(int[] secuencia_origen, int[] secuencia_intermedia, int trabajo,
 int posicion)
{
    for (int i = 0; i < secuencia_intermedia.Length; i++)
    {
        if (i == posicion)
        {
            secuencia_intermedia[i] = trabajo;
        }
        else if (i < posicion)
        {
            secuencia_intermedia[i] = secuencia_origen[i];
        }
        else if (i > posicion)
        {
            secuencia_intermedia[i] = secuencia_origen[i - 1];
        }
    }
}

//Función para sacar un trabajo y devolver el mismo en Local Search
static int sacar_trabajo_LS(int[] secuencia_original, int[] secuencia_nueva, int
 posicion)
{
    int trabajo = -1;
    for (int i = 0; i < secuencia_original.Length; i++)
    {
        if (i < posicion)
        {
            secuencia_nueva[i] = secuencia_original[i];
        }
        else if (i == posicion)
        {
            trabajo = secuencia_original[i];
        }
        else
        {
            secuencia_nueva[i - 1] = secuencia_original[i];
        }
    }

    return trabajo;
}

//Funcion para introducir un trabajo en una secuencia
static void introducir_trabajo_LS(int[] secuencia_original, int[] secuencia_nueva, int
 posicion, int trabajo)
{
    for (int i = 0; i < secuencia_nueva.Length; i++)
    {
        if (i < posicion)
        {
            secuencia_nueva[i] = secuencia_original[i];
        }
        else if (i == posicion)
        {
            secuencia_nueva[i] = trabajo;
        }
        else
        {
            secuencia_nueva[i] = secuencia_original[i - 1];
        }
    }
```

```
        }

        //Función para la fase de destrucción y de construcción
        static void fase_destruccion_construccion(int[] secuencia, out int[] secuencia_final,
         int Ntrabajos_destruccion)
        {
            int[] secuencia_inicial = new int[secuencia.Length];
            //Copio en Secuencia inicial la secuencia dada a la funcion
            Array.Copy(secuencia, secuencia_inicial, secuencia.Length);

            //Fase de destrucción selecciono aleatoriamente 4 trabajos que van a otra secuencia
            int[] secuencia_destruccion = new int[4];
            secuencia_diferente_aleatoria(secuencia_destruccion, secuencia.Length);

            //Sacar de secuencia inicial esos valores
            int[] secuencia_destino;
            sacar_trabajos(secuencia_inicial, out secuencia_destino, secuencia_destruccion,
             secuencia_inicial.Length - secuencia_destruccion.Length);

            //Random insertion de los trabajos en secuencia_destruccion
            int[] posiciones_insercion = new int[secuencia_destruccion.Length];
            secuencia_diferente_aleatoria(posiciones_insercion, secuencia_destino.Length);

            secuencia_final = new int[secuencia_inicial.Length];
            for (int i = 0; i < secuencia_destruccion.Length; i++)
            {
                int[] secuencia_origen = new int[secuencia_destino.Length + i];
                int[] secuencia_intermedia = new int[secuencia_destino.Length + i + 1];
                if (i == 0)
                {
                    Array.Copy(secuencia_destino, secuencia_origen, secuencia_destino.Length);
                }
                else if (i != 0)
                {
                    for (int j = 0; j < secuencia_origen.Length; j++)
                    {
                        secuencia_origen[j] = secuencia_final[j];
                    }
                }
                insercion(secuencia_origen, secuencia_intermedia, secuencia_destruccion[i],
                 posiciones_insercion[i]);
                for (int x = 0; x < secuencia_intermedia.Length; x++)
                {
                    secuencia_final[x] = secuencia_intermedia[x];
                }
            }
        }

        //Función para copiar matrices

        static void copiar_matriz(double[,] matriz_original, double[,] matriz_destino, int
         filas, int columnas)
        {
            for (int i = 0; i < filas; i++)
            {
                for (int j = 0; j < columnas; j++)
                {
                    matriz_destino[i, j] = matriz_original[i, j];
                }
            }
        }
    }
}
```