Trabajo Fin de Máster

Máster Universitario en Ingeniería Industrial

Desarrollo de algoritmos de colonias de hormigas para la resolución del problema de gestión del flujo de pacientes en un servicio de urgencia hospitalario

Autor: Saray Díez Soler

Tutor: José Manuel Molina Pariente

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

Sevilla, 2024

Trabajo Fin de Máster
Máster Universitario en Ingeniería Industrial

Desarrollo de algoritmos de colonias de hormigas para la resolución del problema de gestión del flujo de pacientes en un servicio de urgencia hospitalario

Autor:
Saray Díez Soler

Tutor:
José Manuel Molina Pariente
Profesor Permanente Laboral

Dpto. de Organización Industrial y Gestión de Empresas I
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla
Sevilla, 2024

Trabajo fin de Máster: Desarrollo de algoritmos de colonias de hormigas para la resolución del problema de gestión del flujo de pacientes en un servicio de urgencia hospitalario

Autor: Saray Díez Soler

Tutor: José Manuel Molina Pariente

El tribunal nombrado para juzgar el Trabajo Fin de Máster arriba indicado, compuesto por los siguientes miembros:

Presidente:

Vocales:

Secretario:

Acuerdan otorgarle la calificación de:

Sevilla, 2024

El Secretario del Tribunal

A mi familia

A Pablo

A mi tutor

Y a mí misma. Lo conseguiste.

Resumen

El trabajo fin de máster que se desarrolla a continuación busca ofrecer una mejora de los servicios de urgencia hospitalarios (SUHs) a través de la investigación operativa. Esta mejora se centra en reducir el tiempo que los pacientes transcurren en los hospitales cuando van por una urgencia a la vez que se reducen los tiempos de espera para ser atendidos por primera vez por un médico, priorizando aquellos motivos más urgentes.

Para conseguir esto se ha resuelto el problema de secuenciación de actividades relacionadas con el proceso de urgencia (PU) de cada paciente presente en el SUH. El PU de un paciente define el conjunto de actividades y recursos por los que ha de pasar el paciente en su atención en el SUH. Para la resolución del problema, se ha diseñado y desarrollado una metaheurística basada en la colonia de hormigas (Ant Colony Optimization, ACO), adaptándola a nuestro problema y ofreciendo varias versiones de ésta para finalmente compararlas entre sí y optar por la que mejores resultados ofrezca. Además, se analiza la opción de añadir una búsqueda local a los algoritmos, de forma que intensifique la búsqueda de la mejor solución. La implementación de las metaheurísticas se ha llevado a cabo en el lenguaje de programación de Python.

Para analizar el rendimiento de las metaheurísticas desarrolladas, se han creado una serie de escenarios basados en el estudio de Bedoya y Kirac [6].

Abstract

The following project seeks to offer an improvement in hospital emergency departments (EDs) through operational research. This improvement focuses on reducing the time that patients spend in hospitals when they go for an emergency while reducing waiting times to be seen for the first time by a doctor, prioritizing the most urgent causes.

To achieve this, we have solved the problem of sequencing activities related to the urgency process (PU) of each patient present in the ED. The PU of a patient defines the set of activities and resources that the patient has to go through during ED care. To solve the problem, a metaheuristic based on Ant Colony Optimization (ACO) has been designed and developed, adapting it to our problem and offering several versions of it to finally compare them and choose the one that offers the best results. In addition, the option of adding a local search to the algorithms is analyzed, in order to intensify the search for the optimal solution. The implementation of the metaheuristics has been carried out in the Python programming language.

To analyze the performance of the developed metaheuristics, a series of scenarios have been created based on the study of Bedoya and Kirac [6].

Índice

ÍNDICE DE TABLAS

ÍNDICE DE FIGURAS

1 INTRODUCCIÓN

La verdadera optimización es la contribución revolucionaria a los procesos de decisión.

- George Dantzig -

Tras los últimos años, el uso de técnicas de investigación operativa ha ido en aumento con la finalidad de mejorar la eficiencia en muchos sectores y reducir costes. Gracias a esta se puede simular u optimizar la realidad, encontrando soluciones a problemas complejos y estudiar diferentes estrategias para diferentes situaciones sin la necesidad de ponerlo en práctica en la vida real. Esto es muy útil en el ámbito sanitario, por eso es por lo que podemos encontrar cada vez más estudios que hacen uso de herramientas de simulación y optimización, dando un mejor servicio a los pacientes, a la vez que mejorar la estructura y condiciones de trabajo de los centros.

Es evidente que, tras los últimos acontecimientos, se han dado situaciones extremas para las que ningún sistema estaba preparado, y esto ha traído consigo un aumento de la demanda de servicios sanitarios a la vez que una insatisfacción generalizada de los usuarios. Esta situación ha impulsado la aparición de numerosos estudios enfocados en reestructurar los sistemas y buscar herramientas para mejorar la efectividad y agilidad del servicio y su gestión ante situaciones inesperadas, buscando al mismo tiempo servicios más rentables y de mayor calidad en términos de tiempos de procesos.

Dentro del ámbito sanitario, uno de los servicios más afectado por todo esto es el servicio de urgencia hospitalario (SUH). Este marco de estudio es uno de los más complejos debido a su aleatoriedad y variabilidad, tanto de la llegada de los pacientes como de los motivos de consulta, entre otros factores, sumando los dierentes niveles de urgencia de los casos y la necesidad de respuesta inmediata para los más urgentes. Según Bedoya & Kirac [6], aproximadamente el 2% de los pacientes que acuden al SUH pasan más de 6 horas en el hospital, y alrededor del 10% espera más de 2 horas en ser atendido por un médico por primera vez. Esto principalmente se debe a la gran congestión que sufren estos servicios y una mala gestión de los recursos.

El SUH es un componente clave de todo el sistema sanitario. Los SUHs son unidades semiautónomas que están abiertas y dotadas de personal las 24 horas del día, los 365 días del año, incluidos los días festivos. La misión principal de los SUH es atender únicamente situaciones de emergencia. Sin embargo, las visitas a los SUHs incluyen una amplia gama de enfermedades y lesiones, por lo tanto, nos encontramos con auténticas

urgencias, casos urgentes, semi-urgentes y no urgentes. Algunos SUH han aumentado sus recursos para atender todos esos casos, convirtiéndose en unidades grandes, complejas y dinámicas [11].

La mayoría de las causas de la saturación de estos entornos suele ser debido a una sobreutilización de los SUHs por parte de los pacientes con causas no urgentes. Esto se debe no solo por la inmediatez de la resolución de causas de visita si no por disconformidad y obstrucción de los servicios de atención primaria [26]. Además, una mala programación de horarios del personal o mala programación de los pacientes que se encuentran en el SUH hace que la eficacia y la calidad en cuanto a tiempos de respuesta, disponibilidad de recursos y capacidad de manejo de los servicios decaigan considerablemente.

En general, las soluciones propuestas incluyen el aumento de recursos materiales o el aumento de personal, lo cual conlleva un coste que en ocasiones no es posible afrontar. Por ello, es cada vez más común la búsqueda de nuevas estrategias de gestión para aumentar la calidad, reducir los tiempos de espera y reducir costes entre otros beneficios. Todo esto lo encontramos en la literatura dividido en 2 enfoques o niveles de decisión: el táctico y el operativo.

A nivel táctico encontramos estudios enfocados en mejorar los sistemas para medio-largo plazo buscando adaptar la capacidad y configuración del SUH para satisfacer la demanda cambiante que sufren estos servicios. Dentro de este bloque encontramos estudios enfocados en: la planificación de los recursos humanos basándose en patrones históricos y con previsiones a medio plazo, la programación de horarios de trabajo con objeto de equilibrar turnos o minimizar tiempos de inactividad y gestión de los recursos materiales asegurándose que haya disponibilidad para cualquier situación, pero equilibrando para que no haya material inutilizado.

Por otro lado, se encuentra el nivel operativo, en el cual se centra el presente trabajo. La gestión a nivel operativo se centra en la optimización del uso diario de recursos, así como la asignación a recursos y secuenciación en el tiempo de pacientes y actividades asegurando un buen flujo continuo y satisfacción de pacientes y facultativos a través de buenas decisiones para abordar las situaciones a las que se enfrentan en el día a día. Estos problemas no son triviales (catalogados como ‘NP-Hard’ por su complejidad de resolución y computación) y se necesita de algoritmos de programación complejos para poder abordar todas las restricciones a la vez que cumplir con los objetivos propuestos.

Este proyecto se enfoca en la programación de pacientes en el tiempo, más concretamente en la programación de las actividades pertenecientes a los procesos de urgencia de los pacientes que se encuentran dentro del SUH. Para ello haremos uso de la metaheurística bio-inspirada *Ant Colony Optimization* (ACO) o algoritmo de colonia de hormigas, la cual no se ha usado previamente con este objetivo. El objetivo es minimizar tanto el tiempo total de estancia de los pacientes dentro del hospital (Lenght of Stay, LOS) además de minimizar el tiempo de espera de éstos a que sean atendidos por un facultativo por primera vez (Tiempo de Espera de Primera Consulta Facultativa, TEPCOF). Abordaremos este trabajo con una programación imperativa y estructurada con Python para resolver el problema de programación de actividades en el tiempo, junto con una heurística constructiva para la determinación de las soluciones y para la optimización adaptaremos el ACO al problema. Consideraremos una serie de hipótesis para acercarnos lo máximo posible a la realidad del SUH,

resolviendo distintos escenarios para tener una visión más completa de la situación y respuestas de nuestro algoritmo. En cuanto al algoritmo de optimización, ofreceremos variantes del ACO e incluiremos una búsqueda local para comparar resultados frente a diferentes parámetros como la estanqueidad o el ARPD (average relative percentage deviation). Todo esto se abordará cumpliendo siempre con una serie de restricciones y ajustándose a los objetivos del presente proyecto, todo dentro de un tiempo de computación razonable para el tiempo de toma de decisiones en el entorno del SUH.

1.1 Objetivos

El objetivo de este trabajo es aportar a la literatura otro enfoque del problema de gestión del flujo de pacientes y optimización del SUH. Para ello se resuelve un caso práctico propuesto por Bedoya & Kirac [6], y se optimiza a través de la adaptación de la metaheurística ACO, con objeto de minimizar tanto el *LOS* como el cumplimiento del *TEPCOF*. Finalmente, con esto se buscará gestionar eficientemente el uso de los recursos disponibles en el SUH.

Los pasos y objetivos que se abordan en este trabajo fin de máster son:

- Establecer una serie de hipótesis para facilitar ciertos conceptos del problema, siempre acercándonos lo máximo posible a la realidad en los hospitales.
- Entender el funcionamiento del SUH para poder realizar un análisis lo más cercano a la realidad, con objeto de elaborar una función llamada '*Decoding'* con la que se abordará el problema de programación de los procesos de urgencia de los pacientes en el tiempo, teniendo en cuenta unas determinadas restricciones y conforme unos objetivos que serán la función objetivo. Dentro de esta función se ha implementado una heurística constructiva para elaborar soluciones válidas del problema presentado, además se muestra un diagrama de Gantt para una mayor comprensión de los resultados, y comprobar el correcto funcionamiento del *Decoding*.
- Adaptar la metaheurística ACO a nuestro problema. Con esto se conseguirá una mayor aleatoriedad de los resultados, siendo posible obtener un resultado mejor, en el menor tiempo posible.
- Calibrar los parámetros pertenecientes a los ACO elaborados para un mayor ajuste de los algoritmos. Se ajustarán según ofrezcan mejores comportamientos con los escenarios expuestos.
- Evaluar las diferentes variantes elaboradas del algoritmo de optimización para comprobar cuales ofrecen mejores resultados para nuestro problema, en las diferentes situaciones expuestas.
- Implementar una búsqueda local para potenciar la exploración de resultados.
- Evaluación de los resultados y comparativa con otras propuestas.

2 ESTADO DEL ARTE

Dentro de la gestión hospitalaria, los SUHs representan el punto más crítico de disconformidades y quejas de todos debido a su cada vez más creciente afluencia de pacientes. Su naturaleza impredecible y necesidad de atención rápida y eficiente hace que sea necesario estar en constante búsqueda de nuevas alternativas de gestión y adaptación a diferentes circunstancias, con objetivo de mejorar tanto el servicio ofrecido como la conformidad de toda la comunidad y trabajadores dentro del SUH.

En el capítulo anterior se hizo mención a los dos enfoques que podemos encontrar para mejorar el SUH, el táctico y el operativo. La optimización de los tiempos de procesos dentro del SUH se encuentra en el nivel operativo y es un tema cada vez más estudiado en la literatura, ya sea con una buena programación de pacientes (como es nuestro caso), una mejora en los horarios de personal o una reestructura del funcionamiento del SUH frente a casos no urgentes, entre otros. A menudo enfoque táctico y operativo se entrelazan con objetivos relacionados entre sí (como puede ser un sistema que simule escenarios futuros y llegado el momento se optimice la respuesta). A continuación, se revisarán diferentes estudios con objetivo de mejorar el SUH dentro del enfoque táctico y operativo.

En el enfoque táctico encontramos múltiples estudios que emplean la simulación con objeto de reducir el LOS y tiempos de espera de los pacientes. Este fue el objetivo principal de [33], emplearon la simulación basada en agentes (ABM) en época de COVID-19 para modelar el comportamiento de los pacientes con registros de los tiempos de llegada de los pacientes al SUH e información sobre el desarrollo del flujo de pacientes en el día. Además, emplearon la simulación de eventos discretos (SED) para modelar los recursos y así conseguir una mayor eficiencia del uso de estos. El modelo conceptual fue desarrollado a partir del conocimiento y experiencia de los principales interesados del hospital de estudio y se convirtió en un modelo computacional. La SED es una herramienta muy empleada en estos campos ya que consigue evaluar diferentes situaciones del SUH sin necesidad de llevarlo a la práctica y viendo cuando se ofrecen mejores resultados. En la literatura encontramos varios estudios que la emplean para modelar el flujo de pacientes en el SUH y así poder optimizar el servicio, por ejemplo [38] hace uso de la SED junto con optimización heurística para evaluar diferentes estrategias de asignación de pacientes en hospitales y optimizar tiempos de espera y el LOS. El modelo incluye estrategias tácticas de desbordamiento de pacientes para asignarlos a servicios o unidades alternativas en caso de saturación; [16] en cambio integra un algoritmo evolutivo híbrido con la SED para optimizar la asignación de recursos y horarios empleando variables estocásticas para representar la variabilidad del entorno y distribuciones estadísticas para modelar los tiempos de llegada. [16] emplea el software ExtendSim para modelar el comportamiento de los pacientes junto con un optimizador evolutivo consiguiendo optimizar la distribución de los recursos humanos y materiales consiguiendo reducir los tiempos de espera de

los pacientes y reducir costos. Por otro lado, [2] combina la SED con un algoritmo heurístico de bloqueo de flujo de pacientes (BPF) para reducir los tiempos de espera a través de la optimización en la asignación de camas y recursos.

[1] revisó varios modelos analíticos para la optimización de los recursos en el SUH con el fin de mejorar el flujo de pacientes. Entre estos se encontraba ABM y SED, los modelos de colas, la optimización de simulaciones y la modelización matemática. Concluyó con que no todos los modelos son adecuados para todas las situaciones debido a la complejidad de las interrelaciones y variabilidad en las variables, pero que las técnicas de simulación y optimización ayudaban a identificar los cuellos de botella además de proponer soluciones para mejorar el flujo de pacientes.

Además de una búsqueda de reducción de tiempos, en el SUH es importante tener en cuenta la prioridad de la urgencia del paciente, ya que como se ha mencionado anteriormente, muchos de los pacientes que acuden al SUH no son con afecciones urgentes, si no que debido a enfermedades repentinas o saturación de los servicios primarios optan por estos servicios, empeorando así el servicio que se le ofrece a aquellas personas con prioridades críticas. En esto se enfocó [9], quien quiso mejorar la gestión de la admisión de estos pacientes no críticos con un modelo basado en agentes. Al reprogramar las admisiones según la capacidad del personal y los patrones de llegada previstos según los registros históricos, logró una mejor distribución de los recursos disponibles, que a su vez se traduce en una reducción notable en el LOS y aumento en la satisfacción de los pacientes. Por otro lado, dentro del enfoque operativo, [3] aporta una heurística basada en colas de prioridad, de tal forma que los pacientes se organizan según su urgencia y aquellos con una mayor prioridad son asignados a un doctor tan pronto como sea posible. Esto se combina con una búsqueda de vecindario variable generalizada (GVNS) para optimizar el horario cada vez que llega un paciente al SUH. La gestión de colas también fue empleada por [19] para priorizar la admisión de pacientes según su categoría dentro de la escala de triaje y agudeza canadiense (CTAS), que además propuso un nuevo enfoque basado en Fuzzy-Rule (reglas difusas) para mejorar la gestión del flujo de pacientes y reducir los tiempos de espera.

Harzi M. et al. han tenido varias aportaciones a la literatura relevantes en la optimización de la programación de pacientes en el SUH, en 2017, emplearon un modelo matemático basado en programación lineal entera mixta (MILP) para optimizar la programación de pacientes en el SUH [20]. El modelo tenía en cuenta cuatro actividades: triaje, consulta, tratamiento y hospitalización. Concluyeron que el modelo MILP era eficaz siempre que el problema tuviese un máximo de 25 pacientes, ya que en problemas mayores el modelo no podía encontrar soluciones óptimas. Para resolver esto, en [21] desarrollaron un algoritmo híbrido de búsqueda local iterativa (IILS) y descenso de vecindario variable (VND) los cuales se iniciaban con una primera solución generada mediante la heurística First in First out que posteriormente se optimizaba con el algoritmo ILS/VND. El estudio concluye que la utilización de las metaheurísticas frente a la programación entera empleada en [20] en problemas medianos o grandes, ofrecía mejores soluciones además de reducir mucho el tiempo de ejecución.

En el modelado existen diferentes tipologías en función del problema que se quiere resolver, están los

problemas cerrado o abierto, continuo o discreto, determinista o estocástico y estático o dinámico. El problema de secuenciación de pacientes es catalogado como estocástico y dinámico, pero en la literatura podemos encontrar estudios que por simplicidad lo trate como un problema determinista y estático. [3] aborda el problema de ambas maneras comparando el comportamiento y los tiempos de computación entre dinámico (información disponible secuencialmente) y estático (información disponible desde el inicio). Aunque en ambas situaciones conseguían mejoras significativas, el problema dinámico tiene un mayor costo computacional, siendo un punto negativo en estos entornos. En cambio, [4] se enfocó en la programación dinámica de pacientes con la idea de integrar la información de forma anticipada y luego adaptar las asignaciones dinámicamente frente a nuevos eventos. Para ello hizo uso de un Método Anticipatorio Basado en Escenarios (SBPA) que emplea escenarios simulados para prever eventos futuros y ajustar las asignaciones de pacientes a médicos y técnicas de reoptimización heurística y búsqueda de vecindarios variables (VNS) para adaptar la programación según vayan ocurriendo sucesos de forma continua. Para poder hacer esto se emplearon datos reales de hospitales de Hong Kong y de Italia. De la misma forma [27] y [28] lo tratan como un problema dinámico de forma que se reajuste la simulación según se produzcan cambios en el sistema.

En el presente proyecto, abordaremos el problema de forma estática, de tal forma que se presenta el estado del SUH como una 'foto' en la que podemos encontrar pacientes recién llegados, pacientes que llegaron hace un tiempo, pero no son atendidos, pacientes que están siendo atendidos en ese momento y pacientes que ya han sido atendidos en alguna de sus actividades, pero se encuentran esperando a que les vuelvan a llamar. Esto facilita la computación al no tener que actualizar el sistema cada vez que algo ocurre, y solo actualizarlo cuando sea necesario volver a optimizar el problema. Esto lleva a la necesidad de hacer varias hipótesis como por ejemplo que un proceso de urgencia una vez establecido en el triaje no se variará, o que los tiempos de las actividades no varían a lo predicho en un inicio. Así lo enfocaron Bedoya & Kirac en [6], del cual recogemos en el presente proyecto toda la información relacionada al funcionamiento del SUH y las hipótesis empleadas. En el estudio realizan una SED para evaluar distintos escenarios propuestos. En el modelo de simulación desarrollado en [6] tiene en cuenta a los pacientes que son atendidos por múltiples recursos al mismo tiempo y en varias ocasiones, según los niveles de gravedad de los pacientes. Además, utilizaron tres valores diferentes para medir y comparar los efectos de las políticas operativas propuestas en el SUH, estos valores eran los LOS promedios, el TBSSPPA promedio y los niveles de utilización de recursos. Los resultados obtenidos de este estudio mostraron que es posible desarrollar diferentes políticas operativas para la asignación de recursos con el fin de mejorar la eficiencia del SUH mediante la simulación.

Este estudio también fue una base para los trabajos [5], [30] y [31], antiguos compañeros de la escuela que trataron el mismo problema y cuyos trabajos han sido de gran ayuda en la elaboración del presente estudio. [30] se centró en la optimización del SUH con metaheurísticas como el algoritmo voraz iterativo (Iterated Greedy, IG) y el algoritmo genético. [30] hizo uso del modelo matemático elaborado por [8] para el problema de secuenciación de pacientes que junto con los algoritmos propuestos consiguió reducir la saturación, minimizar los tiempos de espera y equilibrar la carga de trabajo entre facultativos. A raíz de éste, [31] propone, en vez del modelo matemático de [8], dos representaciones para la secuenciación, una llamada por pacientes,

en la que secuencia las actividades de cada paciente de forma consecutiva, y otra llamada por actividades, en las que secuencia las actividades de forma individual, pero respetando los órdenes de precedencia de cada proceso de urgencia. Por otro lado, para la optimización emplea el algoritmo de Iterated Greedy y ofrece distintas variaciones de este con el fin de compararlas y ver cual da mejores resultados para los objetivos que quiere alcanzar que son minimizar el LOS, el TEPCOF y equilibrar la carga de recursos. Por último, [5] ofrece un enfoque distinto, aunque con los mismos objetivos que los anteriores, en el que los pacientes permanecen en un BOX y los recursos acuden a ellos. [5] emplea metaheurísticas como el algoritmo genético, búsqueda tabú y el recocido simulado para la optimización del SUH. Concluye que el recocido simulado demostró ser el que ofrece mejores resultados seguido de la búsqueda tabú y por último el algoritmo genético.

Dado que el problema de optimización del SUH es NP-hard, su resolución en un tiempo razonable (polinómico) es prácticamente imposible debido a su complejidad. Por esto, en muchos casos se emplean enfoques aproximados como las metaheurísticas para su resolución [5], [30], [31]. Las metaheurísticas son técnicas de optimización de alto nivel que guían otros algoritmos heurísticos para explorar y explotar el espacio de soluciones de manera eficiente. [29] empleó la metaheurística de enjambre de partículas para resolver la programación de turnos de los médicos de un departamento de urgencias, para buscar aquella asignación de turnos y horarios que sea más apropiada teniendo en cuenta las restricciones impuestas por las normativas y políticas del hospital.

Existen varios tipos de metaheurísticas, clasificables de distintas formas. Una de ellas podría ser la siguiente:

1. Basadas en trayectorias: en este tipo se hace uso de un algoritmo de búsqueda local que sigue una trayectoria en el espacio de búsqueda para mejorar la solución que se tiene en el momento. Búsqueda local, temple simulado, búsqueda tabú, …
2. Basados en poblaciones: este enfoque considera múltiples puntos de búsqueda en el espacio que evolucionan de forma simultánea, permitiendo la exploración de varias soluciones a la vez. Encontramos dos sub-tipos: las bio-inspiradas, que se basan en procesos biológicos y naturales (algoritmos evolutivos como algoritmos genéticos, basadas en comportamiento social como la optimización basada en colonia de hormigas y basadas en el comportamiento de otras especies como el algoritmo del Cuco), y las no bio-inspiradas como pueden ser los algoritmos miméticos.
3. Basadas en el marco de referencia temporal: este enfoque se distingue de las demás por cómo manejan el tiempo durante la búsqueda de mejores soluciones. Pueden clasificarse en algoritmos que operan en tiempo discreto (evalúan las soluciones antes de avanzar al siguiente paso) y los que operan en el marco continuo de tiempo (analiza las soluciones de forma continua, siendo una exploración más fluida que la anterior).
4. Híbridas: en estas se combinan características de diferentes metaheurísticas para obtener los diferentes beneficios que cada una propone.

La metaheurística usada en el presente proyecto y en lo que principalmente se destaca de la literatura es el ACO y pertenece al segundo grupo, dentro de las bio-inspiradas.

El algoritmo ACO parte de la base del funcionamiento de una colonia de hormigas para alcanzar su alimento y llevarlo al nido de hormigas. En concreto, las hormigas son capaces de encontrar los caminos más cortos entre la fuente de alimento y el nido sin usar la vista. Esto se consigue a través de las feromonas que cada hormiga va desprendiendo por el camino. Las hormigas huelen estas feromonas y tienden a seguir aquellos caminos que tienen más cantidad de esta sustancia, ya que esta sustancia con el tiempo se evapora, y si el camino es corto, la feromona no se evapora tanto como en otros caminos en los que se tarda más en llegar al nido (y por tanto el tiempo que pasa entre que va y vuelve por el mismo camino es mayor). Al final las hormigas tenderán a ir por aquel camino cuyo nivel de feromona es mayor, teniendo así el camino una mayor afluencia de hormigas y una mayor cantidad de feromona, y con el tiempo todas las hormigas de la colmena irán por aquel camino óptimo encontrado [36].

El primer algoritmo de optimización de colonia de hormigas fue propuesto por Gambardella y Dorigo (1996) para el problema del viajante (Traveling Salesman Problem, TSP). En primer lugar, se inicializan los rastros de las feromonas, la información heurística y los parámetros. Posteriormente, iterativamente cada hormiga va seleccionando los trabajos iniciales para luego aplicar repetidamente la regla de transición para seleccionar el siguiente trabajo que realizará, así hasta construir una ruta completa de trabajos. Al construir una hormiga, tanto la información heurística como la feromona se utilizan para elegir los trabajos a los que se dirigen [36].

Por último, es importante mencionar que, además de aportar la novedad de emplear el ACO para optimizar el flujo de pacientes de un SUH, el presente trabajo aporta variaciones del ACO básico con distintos enfoques en cuanto a la representación de las hormigas (por pacientes y por actividades) y variaciones en cuanto a metodología de la generación de la solución, más concretamente modificando el primer paso que realizan las hormigas a la hora de generar un camino. Además, se combinará con una heurística constructiva de elaboración propia dentro de una función denominada '*decoding'*.

3 DESCRIPCIÓN DEL PROBLEMA

Antes de adentrarnos en la resolución del problema, debemos entender qué procesos se siguen en un SUH y cómo se gestiona con las llegadas de los pacientes.

Comenzaremos explicando cuál es el proceso que sigue un paciente en el SUH, denominado en la literatura como proceso de urgencia del paciente (PU), el cual engloba todas las actividades que éste sigue desde que llega al centro hasta que es dado de alta, o derivado a otra unidad de gestión clínica del hospital [8]. En la Figura 1 propuesta por [8] podemos ver cuáles son los flujos tanto del paciente como de la información de éste durante su estancia en el hospital. Los pacientes pueden llegar por varias vías al centro (ver Figura 1) a pie, en coche, ambulancia, helicóptero, etc., lo cual en la mayoría de los casos es un indicativo de la prioridad del paciente (entre otros). Una vez en el centro, el paciente es atendido en Admisión, los cuales cogen sus datos y el motivo de consulta (como previo análisis de sus dolencias y previsión de si fuese muy urgente su hospitalización).

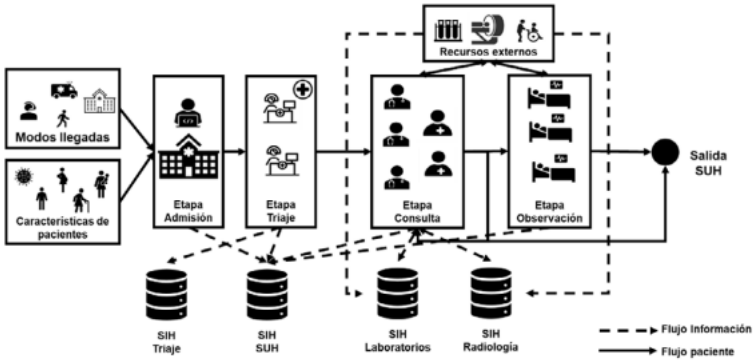

Figura 1. Procesos de un Servicio de Urgencia Hospitalario. Fuente [8]

Posteriormente, el paciente será llamado para el triaje, la espera para entrar a esta sala no suele demorarse de los 5 minutos, mientras que el mismo triaje suele durar de media 1,5 minutos [17]. El triaje es un paso importante en el proceso ya que permite una gestión del riesgo clínico para poder manejar de forma correcta y con seguridad los flujos de pacientes cuando la demanda y las necesidades clínicas superan a los recursos [32]. Para poder llevar a cabo esta gestión, a cada paciente se le asignará un nivel de prioridad en función de parámetros como la gravedad de las patologías, la edad, o el modo de llegada al hospital, que traerá consigo un PU determinado para cada nivel, que constará de las pruebas pertinentes según el motivo de urgencia.

Hasta hace unos años se tenía un sistema de 4 niveles no estructurado, aunque protocolizado desde 1988 en base a un consenso de expertos [18], pero desde 2003 en España, de la mano de la Sociedad Española de

Medicina de Urgencias y Emergencias (SEMES), se ha establecido un estándar de triaje estructurado propio, el Sistema Español de Triaje (SET), garantizando así justicia clínica para todos los pacientes que acuden al SUH, y además asegurar el cumplimiento de los principios de organización, monitorización, evaluación y comparación de los servicios de urgencias [18]. El sistema consta de 5 niveles (del 1 al 5) como se puede ver en la Tabla 1, de forma descendente de gravedad, siendo el nivel 1 la urgencia de atención inmediata o reanimación y el nivel 5 aquel que engloba los casos no urgentes [32]. El nivel de prioridad condiciona por tanto todas las actividades que se integran en el PU, los recursos requeridos para la correcta ejecución de éstas y, la urgencia y rapidez con la que el paciente debe ser atendido por primera vez por un facultativo [18].

Prioridad	***Urgencia***	***Demora máxima***
1	Prioridad absoluta con atención inmediata (reanimación).	Sin demora
2	Situaciones muy urgentes de riesgo vital, inestabilidad o dolor muy intenso.	15 minutos
3	Urgente pero estable hemodinámicamente con potencial riesgo vital que probablemente exige pruebas diagnósticas y/o terapéuticas.	60 minutos
4	Urgencia menor, potencialmente sin riesgo vital para el paciente.	120 minutos
5	No urgencia. Poca complejidad en la patología o cuestiones administrativas, citaciones, etc.	240 minutos

Tabla 1. Tiempo óptimo entre la llegada y la atención de la primera actividad del paciente (fuente [32])

De acuerdo a Bedoya & Kirac [6] una vez ya se encuentra el paciente en el área de consultas, el paciente es observado siempre por un enfermero antes de ver al doctor. Este último le hará las pruebas pertinentes o le ofrecerá el tratamiento que convenga según el caso. Tras haber recibido los resultados, el paciente retornará a la consulta con el facultativo que le ha observado en un inicio, donde se decidirá darle el alta, derivarlo a observación o a otro centro médico.

Llegados a este punto es fácil ver que el mayor problema que afecta a la tardanza reside en la toma de decisiones a la hora de atender a un paciente u otro en el tiempo, añadiendo la falta de tiempo para éste tipo de decisiones y la necesidad de cumplir unos estándares de calidad en la asistencia de los pacientes. En lo que sigue, entraremos más en detalle en aquellos parámetros necesarios para la correcta implementación del comportamiento del SUH.

3.1 Enfoque del Desarrollo del Problema

Para entender más los factores principales del SUH, hemos recogido del estudio de [6] todos los datos relevantes al funcionamiento de un SUH real. Este estudio fue aprobado tanto por el gerente del hospital como por sus empleados, alegando que era una buena representación de la situación y sistema real del SUH para el propósito del trabajo, que era el mismo que el nuestro.

En nuestro problema, la función objetivo que vamos a resolver tiene dos partes, por un lado, minimizar el LOS de un paciente en el hospital, y por otro lado maximizar el cumplimiento del TEPCOF, siendo dos de los factores que más son afectados por una buena programación de tareas y pacientes. El TEPCOF recogido por [6] viene definido por el sistema de triaje canadiense que equivale al Emergency Severity Index (ESI). Este sistema al igual que el español consta de 5 niveles, el cual cada uno lleva asociado una serie procesos personalizados para cada prioridad. Estos procesos PUs se forman por actividades las cuales necesitarán de unos determinados recursos. En el presente trabajo únicamente tendremos en cuenta los recursos de las instalaciones del hospital (estos los conforman las distintas tipologías de consultas, enfermerías, etc..) y no los recursos humanos, suponiendo que el SUH dispondrá de los necesarios y suficientes facultativos para cubrir estas instalaciones.

Los recursos que encontramos son:

- Consulta
- Consulta Asistente
- Enfermería
- Sala de rayos
- Laboratorio

En el presente trabajo se tratará el SUH como un problema estático y determinista, aunque realmente sea dinámico y estocástico debido a su variabilidad y aleatoriedad. Para que este enfoque sea válido en la representación del SUH, se entenderá el instante de la ejecución del programa como una 'foto' de la situación del SUH. En esta foto encontraremos un determinado número de pacientes *i* los cuales se diferencian por su situación en el SUH y el avance que presenten en sus actividades del PU (tipo de paciente = t_i). En el triaje, se le asigna a cada paciente un nivel de prioridad p_i el cual concuerda con el nivel ESI, que recordemos va desde el nivel 1 (más urgente) al nivel 5 (no urgente).

Según este nivel, el paciente tendrá asignado un $TEPCOF_i$ conforme a la Tabla 1. El numero de casos que cumplan con este parámetro deberá ser maximizado, y se deberá evitar que los que no lo hagan sean los pacientes de mayor urgencia.

Por otro lado, cada paciente i tendrá asignado un PU formado por un conjunto de actividades (j) únicas de ese paciente (Tabla 3). A cada actividad le corresponde un único tipo de actividad ($tipo_j$), las cuales se podrán

realizar solamente en aquel recurso del mismo tipo (tipo de recurso = TR).

En la Tabla 2 se resume qué recursos son necesarios y cuántos para cada actividad del PU según el nivel ESI de acuerdo a [6].

Nivel ESI	***Recursos***				
	Consulta	Consulta Asistente	Enfermería	Laboratorio	Rayos
1	1		2	YES	YES
2	1		2	YES	YES
3	1		1	YES	YES
4		1	1	NO	YES
5		1	1	NO	NO

Tabla 2. Recursos necesarios según el tipo de actividad y el nivel ESI. Fuente [6].

Nivel ESI	**Proceso de Urgencia**
1	[1ª ENF - 1ª CON - LAB - RAY - 2ª ENF - 2ª CON]
2	[1ª ENF - 1ª CON - LAB - RAY - 2ª ENF - 2ª CON]
3	[1ª ENF - 1ª CON - LAB - RAY - 2ª ENF - 2ª CON]
4	[1ª ENF - 1ª A.CON - RAY - 2ª ENF - 2ª A.CON]
5	[ENF - A.CON]

Tabla 3. Procesos de Urgencia que siguen los pacientes según el nivel ESI

Por último, siendo fieles a la realidad, en el instante inicial de la ejecución del programa (lo que correspondería a la 'foto' que hacemos del SUH) encontraremos pacientes de distintos tipos:

- Tipo 0: pacientes que acaban de llegar en ese instante ($t_{espera} = 0$, $t_{disponible} = 0$).
- Tipo 1: pacientes que ya habían llegado, pero aún no han sido llamados a su primera actividad ($t_{espera} > 0$, $t_{disponible} = 0$).
- Tipo 2: pacientes que ya han sido atendidos como mínimo en su primera actividad, y se encuentran en la sala de espera, sin ocupar ningún recurso ($t_{disponible} = 0$).
- Tipo 3: pacientes que están siendo atendidos en el instante inicial, estando no disponibles para ser llamados en otra de sus actividades y además de estar ocupando un recurso hasta un $t_{disponible} > 0$.

Es importante mencionar que nos referimos a 'instante inicial' como el tiempo en el que se ha ejecutado el algoritmo de optimización, mostrando la foto del estado de todos los pacientes del centro. Es por esta razón que no calcularemos el $TEPCOF_i$ para los pacientes tipo 2 y 3, ya que, al haber sido ya atendidos en su 1ª actividad, este parámetro no sería reflejo del turno presente (entendiendo como turno la ventana de tiempo entre una ejecución del algoritmo y otra).

A continuación, numeraremos todas las hipótesis y suposiciones que se harán para la completa optimización del SUH y su correcta adaptación a la realidad. Estas suposiciones son las mismas que tomaron en [6] y fueron aceptadas por el hospital de estudio:

1. Los pacientes completarán todo su proceso de urgencia asignado antes de irse.
2. Este proceso de urgencia establecido en el triaje no variará una vez ya asignado.
3. La prioridad del paciente no variará durante toda la estancia de éste en el SUH.
4. Todo el equipo necesario para realizar las pruebas se encontrará disponible a lo largo del día y de la noche.
5. Se asume que todo el personal estará disponible durante su turno para cuando se les necesite. El SUH analizado en [6] programaba las pausas del personal médico durante momentos de bajo número de llegada de pacientes y estando siempre disponible sin interrupciones.
6. No se tiene en cuenta el tiempo que tarda el personal ni los pacientes en moverse de un recurso a otro ya que se asume como un tiempo corto.
7. Cuando los pacientes salen de la sala de prueba ya tienen sus resultados listos (tiempos de espera ya incluidos dentro de las estimaciones de los tiempos de cada actividad).

4 METODOLOGÍA DE RESOLUCIÓN

En este capítulo abordaremos la explicación detallada de la función *decoding* desarrollado para el trabajo con el que conseguimos que haya una correcta programación de los pacientes en el tiempo cumpliendo todas las restricciones ya mencionadas. Encontraremos que, en el proceso de elaboración, se realizó una variante la cual resultó no dar buenos resultados, afectando claramente a los algoritmos y dando soluciones muy lejanas a la solución conseguida por el decoding principal.

Por otro lado, también definiremos en este punto los algoritmos usados y cómo se han adaptado a nuestro problema. El algoritmo con el que conseguiremos optimizar el SUH será el ACO, el cual hemos adaptado de dos formas distintas al igual que hizo [31] con el algoritmo *IG* para poder evaluar sus resultados y ver cual nos ofrece una mayor optimización con respecto a una misma solución inicial dada. Uno de los puntos de vista del ACO realizado es representar la solución con los pacientes, y se programarán en el tiempo conforme la heurística constructiva que se explicará a continuación. Por otro lado, encontraremos el enfoque de ACO por actividades, asegurando siempre las restricciones de precedencia que encontramos en los PU de cada paciente.

4.1 Función 'Decoding'

En el presente proyecto con objeto de optimizar el SUH, se ha decidido realizar desde cero el programa que se encarga de ordenar las actividades de los pacientes en el tiempo, así como la heurística constructiva que emplea éste para seleccionar las actividades. Una correcta ejecución del decoding es crucial, ya que se llamará a esta función cada vez que generemos una nueva estructura de la solución, por ello, para una mayor comprehensión del resultado, como último paso se mostrará por pantalla un diagrama GANTT que nos facilite la comprobación del correcto funcionamiento del programa. A continuación, se procede a detallar el funcionamiento y el uso de los datos sobre el SUH detallados en el capítulo anterior dentro de esta función.

Para la ejecución de esta función, se parte de una representación de la solución elaborada con la heurística de construcción (ver sección 4.1.1) o de las soluciones generadas con los algoritmos de optimización (ver sección 4.3). Estas soluciones factibles se definen como un array de las actividades ($rep_{solucion}$) a realizar en la situación del SUH a simular. Para poder asignarlas a recursos y ordenarlas correctamente en el tiempo, se llama a la función *asigna_recurso*() (ver Anexo), la cual se ha diseñado específicamente para cumplir las restricciones y objetivos del problema que tratamos, que son las que siguen:

- La primera vez que se asigne una consulta (*CON*) o consulta asistente (*CONA*) a un paciente, se le queda guardada para el resto de las actividades del mismo tipo.

- Todos los tipos de recurso (*TR*) salvo el laboratorio (*LAB*) tienen capacidad máxima de pacientes igual a 1, por lo tanto, no puede haber más de un paciente a la vez ocupando un recurso. El paciente que lo requiera tendrá que esperar a que este se libere para poder comenzar su actividad.
- El laboratorio tiene capacidad infinita ya que el tiempo de coger la muestra es mínimo, y se pueden estudiar muchas muestras a la vez. Es por esto que en las actividades de tipo LABORATORIO, se mete al paciente en el recurso en el momento que el paciente lo requiera.
- Si la actividad requiere de más de un recurso a la vez (como el caso de pacientes de prioridad 1 que necesitan dos enfermerías para una misma actividad), se debe buscar aquellos recursos que coincidan en el tiempo para realizar la actividad.

Una vez asignadas todas las actividades a los recursos y se han obtenido los tiempos de inicio y fin de éstas, podemos calcular la función objetivo (FO) de nuestro problema, empezando por el TEPCOF de cada paciente. Recordemos que esto se hace únicamente con los pacientes que no han sido vistos con anterioridad al 'instante inicial' de la 'foto'. En los casos de los pacientes de tipo 2 (ver sección 3.1.), es decir, pacientes que han llegado previos al 'instante inicial' pero aún no han sido atendidos, se añade ese tiempo de espera previo al instante inicial al tiempo que tarda en ser visto por primera vez desde el instante de la foto. En la Figura 2 se puede ver cómo se ha implementado esto en una función llamada *calcula_TEPCOF*().

```
def calcula_TEPCOF(df_pacientes, df_datos_unido):
  # Itera sobre las filas del DataFrame
  for index1, row1 in df_pacientes.iterrows():
        tiempo_primera_actividad = 1000
        # Se calcula solo en aquellos pacientes que no han sido vistos
        if row1['visto'] == 0:
          a = row1['Paciente']
          b = row1['t_llegada']
          # Guardamos el tiempo de su primera actividad (el tiempo de inicio mas pequeño)
          for index2, row2 in df_datos_unido.iterrows():
            if row2['Paciente'] == a:
              t_inicio = df_datos_unido.loc[index2,'Tiempo_inicio']
              if t_inicio < tiempo_primera_actividad:
                tiempo_primera_actividad = t_inicio

          calculo = tiempo_primera_actividad + b
        else:
          calculo = None

        # Añadimos el dato en una nueva columna, en el indice correspondiente del primer DataFrame
        df_pacientes.at[index1, 'TEPCOF_real'] = calculo

  return df_pacientes
```

Figura 2. Función que calcula el TEPCOF de cada paciente

Tras esto, pasamos a la segunda parte de nuestra FO, el LOS. El LOS se define con el tiempo total que pasan los pacientes dentro del SUH. Esto traducido a nuestra función viene dado por el tiempo más temprano registrado de llegada al SUH (esto puede ser o el tiempo 0 si no hay pacientes de tipo 2, o el t_{espera} que llevan estos pacientes en el SUH), y el tiempo de finalización de la última actividad programada.

```
def FO_solucion(df_pacientes, tiempo_final_paciente):
  min_value = 0
  tepcof_FO =0
  clave_maxima, fin_ultima_actividad = max(tiempo_final_paciente.items(), key=lambda x: x[1])
  for index, row in df_pacientes.iterrows():
    t_llegada = row['t_llegada']
    tepcof_real = row['TEPCOF_real']
    tepcof_max = row['TEPCOF']
    prioridad = row['Prioridad']
    if t_llegada != None:
      if t_llegada >= min_value :
        min_value = t_llegada
      if tepcof_real > tepcof_max:
        tepcof_FO += tepcof_real * (6-prioridad)

  FO = fin_ultima_actividad + min_value + tepcof_FO
  return FO
```

Figura 3. Función que calcula la FO del problema

Una vez obtenidos estos valores, podemos calcular nuestra FO según la $rep_{solucion}$ aportada al *decoding*. Es importante destacar que dependiendo de la prioridad del paciente que incumple el TEPCOF (si lo hubiese), esto penalizará más o menos dentro de la FO, así si el paciente es prioridad 2, el TEPCOF real de ese paciente, es decir el tiempo en el que ese paciente comienza su primera actividad, se ve multiplicado por un factor de 4. En cambio, si es prioridad 5 se multiplica por 1 (Figura 3).

Por último, para una mejor visualización de los resultados y comprobación del correcto funcionamiento de las funciones, se genera un diagrama GANTT en el que se puede ver por colores a los pacientes y sus actividades, distribuidas en el tiempo y asignadas al recurso correspondiente. Más adelante nos adentraremos en las diferentes situaciones tratadas dentro del SUH para evaluar el comportamiento de los algoritmos, pero valoraremos principalmente 3 niveles de saturación de éste (50%, 75% y 100%) además de dos turnos de trabajo (a las 12h y a las 6h) donde variarán los recursos que encontramos disponibles.

Un ejemplo de programación del SUH para un 50% de saturación a las 12h se vería de la siguiente forma:

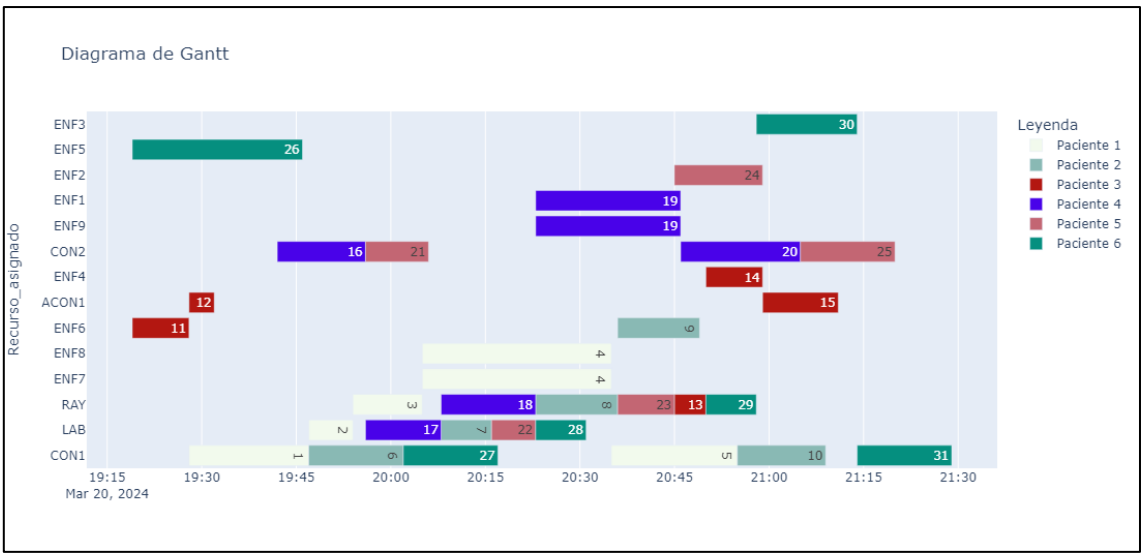

Figura 4. Representación de GANTT de una situación del SUH al 50% de su capacidad

Es importante añadir que en la visualización de la figura 4 no se muestran las actividades que están en curso en el instante 0 de la ejecución del problema. Es por esto por lo que podemos ver pacientes cuyas primeras actividades empiezan más tarde (Paciente 1 y Paciente 4), es porque estos mismos pacientes estaban ocupados hasta ese instante en otra actividad y recurso.

4.1.1 Heurística de Construcción

Una heurística de construcción es cualquier enfoque que se emplee para resolución de problemas de tal manera que se construya una solución de forma incremental rápida y simple.

En el presente trabajo se ha elaborado una de tal forma que se genera una primera solución válida. Esta primera solución no garantiza que sea la mejor, pero sirve para aportar a los algoritmos de optimización un punto de partida, guiando así a éstos hacia una mejora de la solución.

Para la construcción de esta solución, nos sujetamos en la hipótesis de que, si alternamos las actividades de los pacientes ordenando según su prioridad y después agrupando por aquellos que ya hayan sido atendidos dentro del SUH, es más probable que la mayoría cumplan el TEPCOF, priorizando que lo cumplan sobre todo aquellos con mayor nivel de urgencia. Esto supondría una mejora directa de nuestra FO. A continuación, se expone un ejemplo para una mayor visualización de este razonamiento.

***Ejemplo**: En un instante dado, el SUH se encuentra con 5 pacientes de distintos niveles de urgencia y distintas situaciones según se muestra en la tabla 4:*

Paciente	Prioridad	Visto
P1	4	1
P2	2	1
P3	3	0
P4	1	0
P5	3	1

Tabla 4. Ejemplo de situación dentro del SUH

Los pacientes 'no vistos' se identifican con un '0' en la columna '*Visto'*, y los que ya han sido atendidos con un '1'. Sabiendo esto, ordenamos de tal forma que los no vistos aparezcan primero sin importar la prioridad:

Orden 1: [P1 – P2 – P5 – P3 – P4]

A continuación, se ordenan de tal forma que los más prioritarios sean atendidos primero:

Orden 2: [P4 – P2 – P5 – P3 – P1]

Atendiendo ahora a los PU de los pacientes se definen con los siguientes arrays de actividades (Tabla 5):

Paciente	Proceso de Urgencia
1	A1, A2, A3, A4
2	A5, A6
3	A7, A8, A9, A10
4	A11, A12, A13, A14
5	A15, A16

Tabla 5. PU de los pacientes que se encuentran en el SUH

Se construye la solución añadiendo una a una las actividades de cada paciente, alternándolos. Es decir, añadimos, alternando los pacientes según el orden 2, las primeras actividades de cada paciente; a continuación, las segundas actividades, y así hasta añadir todos los PU de los pacientes. Finalmente, la representación de la solución nos quedaría de esta forma ($rep_{solucion}$):

[A11, A5, A15, A7, A1, A12, A6, A16, A8, A2, A13, A9, A3, A14, A10, A4]

Es importante destacar, que como veremos en el capítulo 5 más detalladamente, las actividades de los PU tienen siempre un orden determinado y específico, es por ello, que la primera actividad de los pacientes (no vistos) siempre será 'enfermería', la siguiente será 'consulta', etc.. Si bien es cierto que no todos los pacientes de todos los niveles ESI tienen la misma cantidad de actividades o pasan por todos los recursos, siempre se mantienen las dos primeras actividades. Por este motivo, aunque haya tantas actividades como pacientes (menos uno) entre actividad y actividad, al ser la siguiente actividad de un tipo distinto ($tipo_j$), y necesitar un tipo de recurso distinto (*TR*), encontraremos las actividades seguidas una de otra en su mayoría, sin apenas tiempos de espera entre ellas.

El objetivo es asignar desde el principio un recurso a todos los pacientes, hasta ocupar todos los recursos disponibles en ese instante o hasta que todos los pacientes ya hayan sido vistos por primera vez y penalizando menos en la función objetivo final.

4.2 Algoritmos de Optimización

En el presente trabajo, en un inicio guiaremos al algoritmo inicializando el nivel de feromonas con una primera solución que sabemos que es válida y subóptima ya explicada en el punto anterior. De tal forma el algoritmo comienza con información útil y generará variaciones de la representación de la solución (hormigas) aportando aleatoriedad a los resultados y acercándose más a conseguir una función objetivo óptima.

Dentro del algoritmo de hormigas tenemos dos variables importantes a mencionar: la matriz de feromonas y la matriz de costos.

La matriz de feromonas, como ya se ha mencionado, indica la cantidad de feromonas que hay entre los posibles caminos, esto se traduce en la calidad del camino. Cuando un camino es bueno (más corto), habrá más cantidad de feromonas y por tanto valores más altos de calidad. Esta matriz se actualiza tras cada iteración con la mejor solución de las hormigas generadas en esa iteración, aportando un valor conforme a la calidad de la solución de esa hormiga.

La matriz de costos indica cuanto 'cuesta' ir de un nodo a otro (ya sea trabajo, actividad, etc…) y sus valores dependen del problema que se quiera resolver y de la información heurística que aportemos. Además de tener en cuenta que puede haber caminos que no sean compatibles y reflejarlo en forma de coste infinito. La estructura del algoritmo de optimización es la que se muestra en la figura 5.

```
Inicializar la matriz de feromonas, la matriz de costos y los parámetros
Para cada iteración hasta la condición de parada:
    Para cada hormiga:
        Construir una solución:
            Establecer estado inicial para la hormiga
            Mientras la solución no esté completa:
                Aplicar la regla de transición probabilística para seleccionar la siguiente actividad
                Añadir la actividad seleccionada a la solución de la hormiga
            Fin Mientras
        Fin Construir una solución
        Aplicar 'Decoding' para obtener la programación y asignación de actividades
    Fin Para cada hormiga
    Comparar soluciones para quedarnos con la mejor
    Actualizar feromonas con esa mejor solución:
        Aplicar evaporación de feromonas
        Depositar feromonas basadas en la calidad de la solucion
    Fin Actualizar feromonas
Fin Para cada iteración
Seleccionar la mejor solución encontrada
```

Figura 5. Estructura del ACO propuesto

Para la inicialización en nuestro caso, se proporciona una primera solución factible generada aplicando el decoding a una representación de la solución y se inicializa la matriz de feromonas a partir de ésta. Por otro lado, se genera la matriz de costos, la cual dependiendo de la información que le demos y el problema que tratemos, se calculará de una forma u otra. El costo de ir de un nodo a otro puede ser función de la duración de la actividad a la que se dirige, o de la prioridad del paciente al que pertenece, por ejemplo.

La regla de transición que sigue el algoritmo sirve para elegir la próxima actividad o paciente al que se va a dirigir la hormiga que se está generando. Esta regla de transición sigue la siguiente formula:

$$j = \begin{cases} \arg max_{u \in S_k(i)} \{[\tau(i,u)]^{\alpha}[\eta(i,u)]^{\beta}\} & si\ q \leq q_0 \\ J & si\ q > q_0 \end{cases} \qquad (1)$$

Siendo *j* el siguiente nodo al que se dirigirá la hormiga. El parámetro τ se refiere a la cantidad de feromonas que hay del nodo *i* al nodo *u*, cuyo valor se eleva a un valor preestablecido α. Por otro lado, η hace referencia a la inversa del costo de ir del nodo *i* al nodo *u*, y se eleva a un valor β también previamente establecido. Por último, el parámetro q es un valor aleatorio el cual se compara con el parámetro q_0. Este parámetro determina la importancia entre la explotación frente a la exploración cuyo valor va entre 0 y 1 y se fija al comienzo del algoritmo. Si q es menor o igual que q_0 seguirá la fórmula de arriba, dando prioridad a la información heurística, y si es mayor que q_0, la actividad *j* se decidirá por probabilidad, siguiendo la siguiente formula:

$$p_k(i,j) = \begin{cases} \frac{[\tau(i,j)]^{\alpha}[\eta(i,j)]^{\beta}}{\sum_{u \in S_k(i)}[\tau(i,u)]^{\alpha}[\eta(i,u)]^{\beta}} & si\ j \in S_k(i) \\ 0 & en\ caso\ contrario \end{cases} \qquad (2)$$

Siendo $p_{(i,j)}$ la probabilidad de que la $hormiga_k$ estando en la actividad *i* vaya a la actividad *j*. Esta probabilidad se calcula como en (2), siempre que *j* pertenezca al conjunto de nodos posibles de llegar desde el nodo *i*. En caso de que *j* no se pueda llegar, la probabilidad será 0 y no se elegirá.

Cuando se han generado el número de hormigas determinado para cada iteración, se comparan resultados y se actualizan las feromonas (3). En esta fórmula, hay dos pasos: evaporación y actualización de feromonas. La evaporación lo que hace es restar feromonas a todos los caminos de tal forma que cuando pase el tiempo si un camino ya generado no se ha vuelto a recorrer sus feromonas van a descender siendo menos probable que se elija en un futuro. Esto también promueve la exploración de caminos de tal forma que a no ser que uno se intensifique mucho, no se estancará el algoritmo.

$$\tau(i,j) = (1 - \rho g) \cdot \tau(i,j) + \rho g \cdot \Delta\tau(i,j) \quad (3)$$

$$\Delta\tau(i,j) = \begin{cases} (L_b)^{-1} & si\ (i,j) \in mejor\ camino \\ 0 & en\ caso\ contrario \end{cases} \quad (4)$$

En la evaporación le restamos a toda la matriz un factor ρ_g (tasa de evaporación) que simulará la evaporación de las feromonas con el tiempo. Y únicamente a las parejas de nodos escogidas por la mejor hormiga, se le sumará un valor $\Delta\tau$ el cual determina la calidad de la solución generada por esa hormiga. Como en nuestro caso, cuanto menor sea la solución mejor, sumaremos un valor FO^{-100} de tal forma que si la solución de $hormiga_1 = 230$ y la de $hormiga_2 = 222$, la primera sumará 0.4347 a la matriz de feromonas y la segunda 0.4504, aportando ésta ultima una mayor cantidad y promoviendo que próximas hormigas sigan el mismo camino que ha dado esta hormiga.

Por otro lado, muchos estudios que usan el ACO realizan un último paso de búsqueda local [12][39] de tal forma que cuando encuentra una buena solución, elabora nuevas soluciones cercanas a la solución encontrada con intención de encontrar una mejor solución. Este último paso incrementa mucho el tiempo de computación y dependiendo del problema a resolver puede ser más o menos útil. En nuestro caso, como el tiempo de computación es un factor importante debido a que tratamos un problema de urgencias médicas, se añadirá una búsqueda local a los algoritmos y los compararemos con sus versiones sin ésta en el último capítulo, observando si tiene una mejora significativa o el tiempo de computación de más no es recompensado.

Para la construcción de la solución de las hormigas se puede tener dos puntos de vista: variar la posición de las actividades o variar la posición de los pacientes. En los apartados que siguen, procedemos a explicar las variaciones de este algoritmo elaboradas para la resolución de nuestro problema.

4.2.1 ACO_actividades

El ACO enfocado en las actividades se basa en generar múltiples representaciones de la solución con diferentes combinaciones de todas las actividades pertenecientes a los pacientes. Para hacer esto correctamente debemos asegurarnos de que el orden de precedencia de las actividades de cada paciente se cumple. Para ello,

se ha programado el algoritmo de tal forma que antes de seleccionar la próxima actividad *j*, asegura que todas las actividades que se han tenido que ejecutar antes de ésta ya se han añadido a la solución de la hormiga, cumpliendo así las restricciones de nuestro problema.

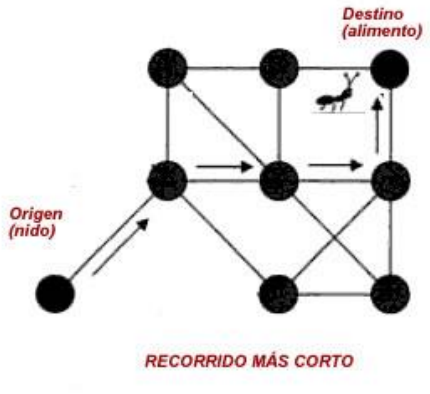

Figura 6. Diagrama de nodos de los posibles caminos de las hormigas

A la matriz de costos de este algoritmo aportaremos información tanto de las actividades que intervienen como del paciente al que pertenece la siguiente actividad a la que se dirige. Siendo *i* la actividad en la que se encuentra la hormiga, *j* la actividad a la que se quiere dirigir y *k* el paciente al que pertenece la actividad *j*:

$$coste(i,j) = 10 + tiempo\_actividad_j - (6 - prioridad\_paciente_k) \quad (5)$$

De esta forma, cuanto más dure la actividad *j*, más costará dirigirse a esa actividad. Esta decisión viene inspirada por [37] el cual defendía que al realizar primero aquellos trabajos cuya duración sea menor y dejando las actividades de mayor duración para el final conseguía mejores resultados y un tiempo total de proceso menor. Defendía que, si se escoge una actividad larga, ésta podía hacer efecto dominó alargando el tiempo total del proceso. Por otro lado, cuanto mayor prioridad tenga el paciente al que pertenece la actividad *j*, menos costará elegir esa actividad como próxima actividad, dando así más valor a aquellos pacientes prioritarios frente los menos urgentes.

Es importante destacar que aquí debemos tener cuidado con aquellas transiciones que no son posibles y respetar los órdenes de precedencia entre actividades de cada paciente, ya que si un paciente tiene un proceso de urgencia formado por [A1, A2, A3], no podemos ir de A2 a A1 ni de A3 a A2 o de A3 a A1. A esas transiciones les daremos un valor infinito para que la hormiga no tienda a escoger esa transición. También será infinito el coste de ir de una actividad a sí misma.

Seguidamente, sabiendo aquellas actividades válidas a las que la hormiga puede ir, el algoritmo sigue la regla de transición mencionada anteriormente para escoger una entre todas las actividades válidas. Esto se realiza con cada actividad a añadir en la representación de la solución de la hormiga, hasta completar todas las actividades y obtener así todo el camino seguido por la hormiga.

Cabe destacar que para que una hormiga comience a escoger su camino, necesita una actividad inicial de la que partir, y aquí es donde hemos jugado con las variantes del algoritmo.

4.2.1.1 ACO_actividades 1:

En la literatura, podemos encontrar casos en los que el nodo inicial se fija debido a que los caminos solo pueden empezar, por un lado, otros casos en los que se escoge de forma aleatoria ya que es posible empezar el camino desde cualquier nodo [36], o porque inicialmente no se tiene información de las feromonas [37]. Por otro lado, si se puede empezar desde varios nodos, se puede añadir un nodo de INICIO [12] desde el cual la hormiga elija el nodo de partida siguiendo la misma regla de transición que sigue con el resto de los nodos del camino. Este último caso es el empleado en el primer algoritmo ACO.

El primer nodo que se añade a la matriz es un nodo de inicio desde el cual la hormiga solo podrá dirigirse a aquellas primeras actividades de cada paciente. La decisión de la primera actividad hacia la que se dirija se tomará conforme a la regla de transición explicada anteriormente (1), la cual puede ser tanto por probabilidad como por la información heurística proporcionada.

4.2.1.2 ACO_actividades 2:

Debido a que el objetivo con el algoritmo de optimización es dar aleatoriedad a las soluciones para estudiar la mayoría de las soluciones posibles, se ha planteado una variante del algoritmo de hormigas donde las hormigas comienzan por una actividad elegida de forma totalmente aleatoria entre las actividades posibles. De esta forma, y a diferencia del ACO_actividades 1, todas las actividades tienen las mismas oportunidades de ser elegidas comienzo de la solución.

Esta variante da pie a una mayor aleatoriedad y variabilidad de los resultados, pudiendo ofrecer soluciones no exploradas por otros algoritmos. Recordemos que debemos asegurar que el orden de precedencia entre actividades se cumple, es por esto que, aunque se escoja de forma aleatoria la primera actividad, debemos asegurar que empieza por una de las primeras actividades de los procesos de urgencia de los pacientes.

4.2.1.3 ACO_actividades 3:

Por último, y siendo inspirado por el funcionamiento de una búsqueda local o recocido simulado, en esta tercera variante la primera actividad de las hormigas en cada iteración será la primera actividad de la mejor solución entre todas las mejores soluciones anteriores.

Como inicialmente solo tenemos la primera solución inicial dada, las hormigas de la primera ronda escogerán la primera actividad de esa primera representación de la solución inicial. A posteriori, irá comparando las mejores soluciones de cada iteración y quedándose con la primera actividad de la mejor solución encontrada hasta el momento, guiando así al algoritmo hacia un óptimo local.

4.2.2 ACO_pacientes

El enfoque por pacientes del ACO es más simple en cuanto a restricciones, ya que no existe ningún orden de precedencia entre pacientes, y en cuanto a variabilidad en los resultados, ya que el número de pacientes que se encuentra en el SUH es mucho menor que el número de actividades a programar. Si bien es cierto que, como veremos más adelante en el análisis de los resultados del algoritmo, este tipo de ACO llega muchas más veces a un mejor valor de la función objetivo que el ACO por actividades, y tarda menos en ejecutarse ya que tiene que hacer menos comprobaciones en cada hormiga para elegir el próximo paciente al que se dirige ésta.

En este caso, la información heurística que aportaremos en la matriz de costos viene referida únicamente al paciente en función de la prioridad del paciente destino y de si ha sido visto o no con anterioridad. Siendo *i* el paciente en el que se encuentra la hormiga, y *j* el paciente al que se quiere dirigir, el coste(*i*,*j*) viene dado por:

$$Coste(i,j) = (1 / (6 - prioridad_j)) * (1 - 0.2 * visto_j) \qquad (6)$$

Por tanto, a mayor prioridad del paciente *j*, menor coste. $visto_j$ toma valores de 0 en caso de si no ha sido atendido en ninguna de sus actividades anterior al instante 0 de ejecución y 1 en caso de haber sido atendido por un facultativo en el instante 0 y por tanto llevar más tiempo en el servicio de urgencias. En caso de ser del primer grupo, el coste para elegir ese paciente será mayor que un paciente perteneciente al segundo grupo, dando prioridad así a aquellos pacientes que ya se encontraban en el servicio de urgencias. Por último, el coste de ir de un paciente a sí mismo será infinito, de tal forma que una hormiga nunca elija ese camino.

Para este enfoque nuestra solución inicial se debe dar como una secuencia de pacientes ([P1, P2, P3, P4]) con el cual inicializaremos la matriz de feromonas. Es importante mencionar que para obtener la FO obtenida por cada hormiga, debemos pasar el array de pacientes de la solución de la hormiga a una secuencia de actividades para que el decoding pueda determinar el valor de la función objetivo. Esto se realiza a través de la heurística de construcción empleada para la primera solución dada.

Para la decisión del primer paciente a elegir, como en este caso sí podemos empezar desde cualquier nodo, se escoge siguiendo la regla de transición (1) al igual que el resto de las pacientes que se irán añadiendo a la solución.

4.3 Búsqueda Local

Dentro de la búsqueda de soluciones a problemas complejos, la búsqueda local es una herramienta comúnmente empleada. Este tipo de heurísticas se presentan de diferentes formas que combinándolas con metaheurísticas permiten obtener muy buenos resultados.

En el presente trabajo, se ha elaborado una búsqueda local de tal forma que añadiéndola a los algoritmos ACOs propuestos, pretendiendo explorar aún más el entorno descubriendo nuevas soluciones. Esta búsqueda local se activa tras cada iteración del ACO y, aunque la solución proporcionada no mejore a ninguna de las mejores soluciones obtenidas anteriormente por el algoritmo de optimización, se añadirá como posible

solución dentro de la colonia de hormigas.

Esta búsqueda local coge la secuencia de pacientes aportada por la primera solución inicial y extrae dos pacientes aleatorios. Para cada paciente, estudiará la posición que mejor resultados obtenga y devolverá dicha solución. Dado que la solución se presenta en array de pacientes, se le aplicará la heurística del punto 4.2. para convertirlo en un array de actividades y que el decoding pueda secuenciarlas en el tiempo para obtener la FO (Figura 7).

Mas adelante comprobaremos si realmente supone una mejora frente a los propios ACO elaborados en el trabajo.

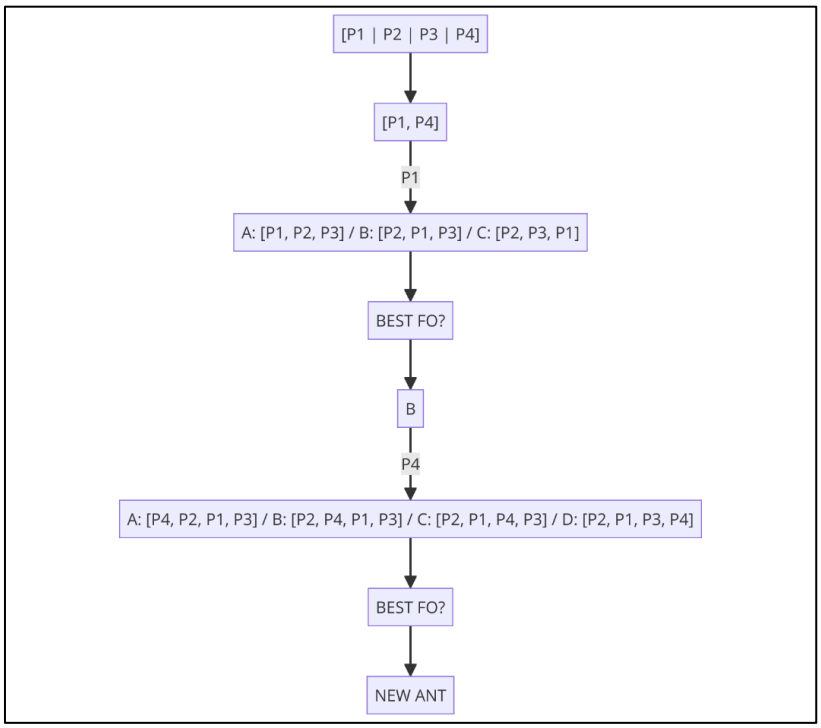

Figura 7. Diagrama del funcionamiento de la búsqueda local

5 ANÁLISIS COMPUTACIONAL

En el capítulo que sigue, expondremos todas las herramientas utilizadas para realizar el código del trabajo, además de adentrarnos más específicamente en la jerarquía del programa generado y modo de ejecución.

Además, se explicarán todos los parámetros usados en los algoritmos de optimización junto con las calibraciones necesarias de éstos para conseguir explotar al máximo los resultados sin dejar de cumplir con los objetivos del problema.

5.1 Herramientas utilizadas

Para el desarrollo del proyecto se ha usado el entorno de Google Colab como cuaderno para desarrollar el código a la vez que se estructura el programa por apartados, dando una facilidad de legibilidad y la oportunidad de combinar código y texto para clarificar ciertos puntos del programa. Este entorno está habilitado para distintos lenguajes incluido Python que es el usado en el presente trabajo por su facilidad de uso y su multitud de librerías, las cuales dan mucho juego y flexibilidad al programador.

El ordenador empleado ha sido un LENOVO ThinkinPad E14 GEN 2.

El código detallado y las funciones vienen en el ANEXO 1, pero en los apartados que siguen pasaremos brevemente por cada una de ellas explicando su funcionalidad junto con las librerías utilizadas para su realización.

5.1.1 Librerias

Las principales librerías utilizadas a lo largo del código han sido Pandas, Numpy y Matplotlib. El resto de las librerías parten de la biblioteca estándar de Python como puede ser: random, datetime, math, json… de las cuales utilizamos módulos específicos según necesidad.

- Pandas: esta librería se utiliza principalmente para el análisis y manipulación de datos. Con ella podemos trabajar con estructuras de datos flexibles, además de una multitud de herramientas para trabajar con los datos y series temporales. La principal estructura y por la principal razón que se incluye en este trabajo son los DataFrames, que son tablas bidimensionales con etiquetas tanto en filas como columnas, siendo muy fácil trabajar con ellas a través de funciones de Pandas.
- Numpy: librería que ofrece un objeto que se basa en una matriz multidimensional de alto rendimiento, y herramientas para trabajar con estos objetos. Esta librería es muy útil para todo tipo de operaciones matemáticas y estadísticas con datos numéricos. Además, ofrece una amplia variedad de funciones matemáticas ya integradas que facilitan este tipo de aplicaciones en la computación científica.
- Matplotlib: esta librería es una de las mejores opciones que hay para la visualización de datos en 2D en

Python. Podemos encontrar una gran variedad de gráficos para cubrir todas nuestras necesidades (de líneas, histogramas, gráficos de barras…) y se combina muy bien con Numpy y Pandas para visualizar los datos de la manera más efectiva y clara. Principalmente la añadimos para generar el diagrama de GANTT y hacer las gráficas de los análisis.

Además de estas tres principales, como ya se ha mencionado, Python ofrece una gran biblioteca de librerías con múltiples módulos con los que se puede abarcar las funcionalidades más básicas y esenciales para generar tareas de programación como puede ser generar un numero random, crear y manipular fechas y horas o incluso realizar las funciones matemáticas más comunes además de importar y extraer datos de diferentes fuentes.

5.2 Jerarquía del código

El código realizado se compone de una serie de funciones las cuales hemos dividido en apartados en nuestro cuaderno de Google Colab como podemos ver en el ANEXO 1. Entre ellas se encuentran las funciones para generar las variables necesarias de forma aleatoria, las funciones para generar los pacientes, las funciones centradas en la secuenciación de actividades y las funciones pertenecientes a los algoritmos, tanto los que tienen búsqueda local como los que no.

Todas estas funciones son las empleadas para la correcta resolución del problema planteado. En el caso de cambiar tanto los recursos como tiempos de actividades, entre otros, solo habría que modificar los valores de entrada a las funciones.

Para generar situaciones del SUH, se tienen en cuenta dos situaciones del SUH definidas por [31], el entorno de las 12h y el entorno de las 6h. Posteriormente, se generan tantos pacientes como se quiera estudiar, en el presente trabajo se abordan tres niveles de saturación del SUH: 50% , 75% y 100%. A continuación, se expone todo lo necesario para esta generación de instancias y de pacientes, definiendo los parámetros necesarios para ello.

5.2.1 Generación de instancias

5.2.1.1 Entornos

Para la generación de instancias tenemos dos entornos a definir dentro del SUH, el entorno de las 12h y el de las 6h. En cada uno podemos encontrar distintos recursos disponibles como vemos en la tabla 6.

Recurso	12h	6h
Consulta	2	1
Consulta Asistente	2	0
Enfermería	9	6
Sala de rayos	1	1
Laboratorio	1	1

Tabla 6. Cantidad de recursos disponibles en cada entorno

5.2.1.2 Saturación del SUH

Con el entorno definido, se comienza a generar pacientes de forma aleatoria ajustados al nivel de saturación requerido. El nivel de saturación se mide para cada tipo de recurso (*TR*) (y no para cada recurso (R) por separado). Esto significa que cuando alguno de los TR alcance un tiempo definido de ocupación por las actividades de los pacientes, el programa deja de generar pacientes. Este tiempo viene definido por el *LOSmedio* (Lenght of Stay medio) [6], el cual se interpreta como la duración media que un paciente suele estar en el SUH. Este tiempo viene definido para cada nivel de prioridad (Figura 8).

Table 3. Results of the validation run of the developed model.

	ESI level				
	1	2	3	4	5
Simulation average LOS (hrs.)	2.53	3.15	2.81	1.54	1.30
Real average LOS (hrs.)	3.15[a]	3.14	2.86	1.49	1.31
Real maximum LOS (hrs.)	3.15[a]	13.95	26.87	9.40	4.00
Simulation average time to be seen by a P/PA (hrs.)	0.05	1.10	1.21	1.00	1.23
Real average time to be seen by a P/PA (hrs.)	0.03[a]	1.14	1.27	1.03	1.25

[a]Corresponding to one patient during the period of time analyzed.
ESI: Emergency Severity Index; LOS: length-of-stay; P: physician; PA: physician assistant.

Figura 8. Tiempos medios reales y de simulación del LOS y TEPCOF. Fuente [6].

En este trabajo, se tomará un único *LOSmedio* para todos los niveles de la siguiente forma [31]:

$$LOSmedio = \frac{(3.15 + 3.14 + 2.86 + 1.49 + 1.31) * 60}{5} = 143{,}4\ minutos$$

Por lo tanto, si queremos estudiar un entorno del SUH al 50%, debemos parar de generar pacientes cuando el tiempo de ocupación de alguno de los TR alcance el 50% de 143,4 minutos.

5.2.1.3 Generación de pacientes

Una vez definido el entorno que queremos estudiar, procedemos a la generación de pacientes de forma aleatoria. Esto se consigue generando un bucle en el cual se van generando uno a uno pacientes, con todos los datos necesarios (recogidos en capítulo 3) para que queden definidos. Entre esos datos encontramos tiempo de llegada del paciente al hospital, tiempo de disponibilidad en caso de estar ocupando algún recurso, su prioridad, entre otros. También se generan todos los datos necesarios de las actividades de cada paciente (duración de la actividad, tipo de recurso que necesita y cantidad de esos recursos que necesita) y la información de los recursos existentes en ese momento (tiempo de disponibilidad).

Comenzamos por los datos que se tienen de entrada. Lo primero que se selecciona de forma aleatoria es la prioridad del paciente que se está generando en ese momento. Según [6] las probabilidades de que un paciente que acude al SUH tenga una determinada prioridad son las mostradas en la Figura 9. Por lo tanto, este dato se generará con una función pseudo-random que se ajusta a estos porcentajes cada vez que genera un paciente. Como vemos en el rosco de porcentajes de la figura 9, sobre todo encontraremos pacientes de prioridad 3 y 4.

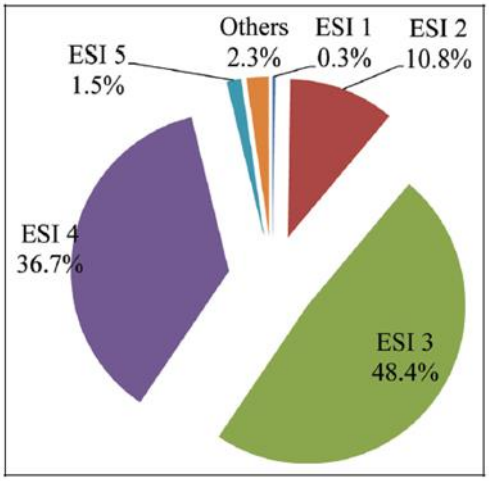

Figura 9. Porcentajes de asistencia de los pacientes según su prioridad

Una vez tenemos la prioridad, se pueden generar automáticamente el resto de los datos pertenecientes al paciente y al SUH. Los datos relativos al proceso de urgencia son generados conforme al nivel de prioridad del paciente tal y como vimos en el capítulo 3 y de forma aleatoria se establecerá una situación para ese paciente dentro del hospital (ver capítulo 3, tipos de pacientes). Dependiendo también de la prioridad y del punto de proceso en el que se encuentra una actividad de un paciente, se define el tiempo de la actividad siguiendo lo establecido por [6] (Figura 10).

Process	Type of patient				
	ESI 1	ESI 2	ESI 3	ESI 4	ESI 5
Sign-in	Triangular (3, 5, 10)	Triangular (3, 5, 10)	Triangular (3, 5, 10)	Triangular (3, 5, 10)	Triangular (3, 5, 10)
Triage	Triangular (0.5, 1, 1.5)	Triangular (2, 3, 5)	Triangular (5, 7, 10)	Triangular (5, 7, 10)	Triangular (5, 7, 10)
Registration	Triangular (3, 8, 12)	Triangular (3, 8, 12)	Triangular (3, 8, 12)	Triangular (3, 8, 12)	Triangular (3, 8, 12)
First visit (nurse)	Triangular (20, 30, 75)	Triangular (18, 28, 45)	Triangular (15, 22, 30)	Triangular (5, 10, 20)	Triangular (2, 4, 8)
First visit (P/PA)	Triangular (10, 20, 25)	Triangular (10, 20, 25)	Triangular (5, 15, 20)	Triangular (2, 3, 5)	Triangular (2, 3, 5)
Second visit (nurse)	Triangular (20, 30, 40)	Triangular (15, 20, 30)	Triangular (10, 15, 20)	Triangular (5, 8, 12)	-
Second visit (P/PA)	Triangular (20, 30, 40)	Triangular (15, 20, 25)	Triangular (8, 15, 18)	Triangular (5, 10, 15)	-
X-Ray	Triangular (3, 5, 15)	Triangular (5, 10, 20)	Triangular (5, 10, 20)	Triangular (2, 5, 10)	-
Blood	Triangular (3, 5, 15)	Triangular (3, 5, 15)	Triangular (3, 5, 15)	-	-
CT-Scan	Constant 20	Constant 15	Constant 15	-	-

P: physician; PA: physician assistant.

Figura 10. Tiempos de las actividades de los PU según la prioridad

Es evidente que no podemos fijar los tiempos de las diferentes actividades que encontramos en los PU de los pacientes, sin embargo, en [6] realizaron una modelización de estos tiempos tras entrevistas con los trabajadores del SUH, los cuales siguen una distribución triangular de 3 valores distintos para cada actividad, diferenciando el orden de ésta, y el nivel ESI al que pertenece.

Esto se hace en la función *generar_pacientes_y_recursos()*, la cual generará pacientes hasta que los recursos superen el nivel de saturación que se le indique para el escenario que se quiera simular. Para generar estos datos de entrada, necesitamos establecer:

- Procesos de urgencia (*procesos_por_prioridad):* esta variable establece a cada paciente su proceso de urgencia según su prioridad asistencial y de la franja horaria en la que nos encontramos. También en función de en qué actividad se encuentra dentro de este proceso de urgencia, se le establecerá un tiempo a la actividad. Esto se establece a través de una función triangular conforme lo establecido en la Figura 10.
- Tipos de recursos (*TR*): los tipos de recursos disponibles que hay en el hospital y el número de cada uno. Según el turno horario en el que nos encontremos habrá disponibles unos recursos u otros (Tabla 6).
- Recursos (*R*): los diferentes recursos que encontramos en el hospital con su número identificativo.
- Tiempo de espera hasta la primera consulta con un facultativo *(TEPCOF)*: este tiempo es establecido por el SET y depende de la prioridad establecida a cada paciente (Tabla 1).
- Capacidad máxima de pacientes en un recurso (*CR*): Dentro de los recursos encontramos tipos en los que solo se puede estar un paciente a la vez o recursos como el laboratorio en el cual se pueden estar varios a la vez (Tabla 2).

  Con esto obtenemos toda la situación del SUH y sus pacientes necesarios para simular un escenario próximo a la realidad. Todos estos datos se guardan en DataFrames, ya que esta estructura facilita la visualización de estos y su manipulación.

- ***df_pacientes***:

	Paciente	TEPCOF	Prioridad	t_llegada	visto	t_disponible	tipo_paciente	PU
0	1	NaN	3	NaN	1	7	4	[CON, LAB, RAY, ENF, CON]
1	2	NaN	2	NaN	1	20	4	[CON, LAB, RAY, ENF, CON]
2	3	60.0	3	34.0	0	0	2	[ENF, CON, LAB, RAY, ENF, CON]
3	4	60.0	3	0.0	0	0	1	[ENF, CON, LAB, RAY, ENF, CON]

Figura 11. DataFrame que recoge los datos de los pacientes del SUH

En esta estructura de datos encontramos las siguientes columnas:

- **Paciente**: número identificativo de cada paciente.
- **Tepcof**: Tiempo máximo que el paciente debe de tardar en ser atendido según su prioridad. Para aquellos pacientes que no calcularemos su TEPCOF el valor será NaN ya que ya han sido atendidos en sus primeras actividades.
- **Prioridad**: prioridad asignada al paciente según patología.
- **T_llegada**: tiempo en el que ha llegado el paciente. Este apartado será 0 si acaba de llegar (paciente tipo 1) o será mayor que 0 en caso de ya encontrarse en el hospital, pero no haber sido atendido (paciente tipo 2). Para aquellos pacientes ya atendidos su valor será NaN ya que no calcularemos su TEPCOF (pacientes tipo 3 y 4).
- **Visto**: 1 si un paciente ya ha sido atendido en alguna de sus actividades, 0 si no.
- **T_disponible**: tiempo en el que el paciente se libera de la actividad en la que se encuentra si es su caso.
- **Tipo_paciente**: tipo de paciente establecido al generar pacientes. 1= el paciente acaba de llegar, 2= el paciente ya se encontraba en el hospital, pero no ha sido atendido, 3= el paciente ya ha sido atendido pero no se encuentra en ninguna actividad si no en la sala de espera, y 4= el paciente ya ha sido atendido y además se encuentra ocupando algún recurso.
- **PU**: proceso de urgencia que tiene que seguir el paciente a partir de la ejecución del problema.

- ***df_datos:***

	Actividad	Paciente	Prioridad	TR	Recursos_Necesarios	Tiempo
0	1	1	3	RAY	1	15
1	2	1	3	ENF	1	13
2	3	1	3	CON	1	15
3	4	2	2	CON	1	20
4	5	2	2	LAB	1	7
5	6	2	2	RAY	1	15

Figura 12. DataFrame que recoge los datos de las actividades de los pacientes

En este dataframe encontramos un despliegue de todas las actividades que se tienen que realizar en todos los pacientes, con toda la información necesaria para poder programarlas en el tiempo como el tipo de recurso en el que tiene que ser asignada, el número de recursos que se necesitan para poder realizarla y la duración de ésta. Además del paciente al que pertenece y la prioridad de éste para completar los datos.

- ***df_recursos_final:***

	recurso	t_disponible
0	CON1	0
1	ENF1	2
2	ENF2	0
3	ENF3	0
4	ENF4	0
5	ENF5	0
6	ENF6	0
7	LAB	0
8	RAY	0

Figura 13. DataFrame que recoge los datos
de los recursos en el instante inicial

En el dataframe, encontramos los datos necesarios del estado de los recursos disponibles en el momento de la simulación. Podemos ver si algún recurso está siendo ocupado por algún paciente en el momento 0 y el tiempo en el que se liberaría.

- ***df_recursos_ocupados:***

	recurso	t_disponible	TR
0	ENF1	2	ENF

Figura 14. DataFrame que recoge los recursos ocupados

Esta tabla únicamente sirve para contabilizar el tiempo inicial en el que los recursos están ocupados dentro de la saturación del hospital, ya que como veremos más adelante, este tiempo no se muestra en el GANTT y por tanto no se refleja de otra forma.

- ***orden_actividades_por_paciente***:

Esta variable tiene forma de *diccionario.* Nos sirve para saber en todo momento el orden que deben tener las actividades dentro de cada paciente, sabiendo el número identificativo de éstas.

5.2.1.4 Decoding

Finalmente, con los datos de la instancia generados, ejecutamos la función 'decoding' para ordenar los pacientes en el tiempo y obtener una primera solución factible. Como salida de esta función obtenemos la FO definida para nuestro problema y un diagrama de GANTT para una mayor legibilidad del resultado. Un ejemplo de una representación del SUH al 50% de saturación en el turno de las 12h sería el que se muestra en la figura 15.

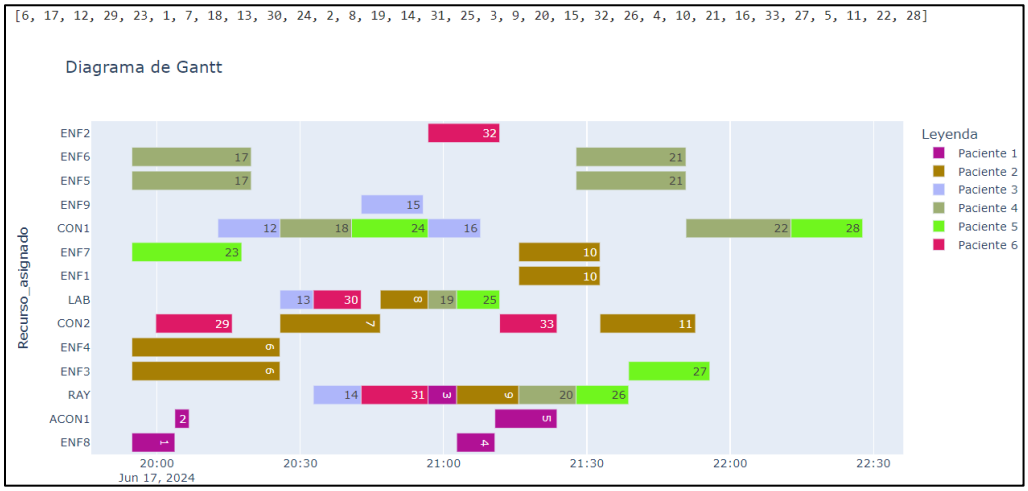

Figura 15. Ejemplo de una 'foto' del SUH a las 12h y al 50%

Es importante mencionar, que aquellos pacientes cuyo estado inicial se encuentra dentro de algún recurso no se representa en el diagrama de GANTT (vemos como el paciente 3 o el 6 empiezan su proceso de urgencia en una consulta (CON) en vez de en enfermería (ENF) y no empiezan en el minuto 0, si no a partir de su tiempo de disponibilidad que son el minuto 18 y el minuto 5 respectivamente), pero sí se tiene en cuenta a la hora de calcular la saturación de los recursos.

A partir de esta solución damos paso a los algoritmos de optimización para que mejoren el resultado de la FO obtenida. Para ello nos llevamos los datos generados (DataFrames) y la solución generada ($rep_{solucion}$ y FO).

5.3 Calibración de parámetros

En este apartado se calibran todos los parámetros que encontramos en los algoritmos, evaluando qué valores se escogen para nuestro trabajo, teniendo en cuenta nuestro objetivo y los experimentos computacionales realizados. Con esto podremos evaluar la repercusión de estos parámetros en nuestros algoritmos, además de entender la tendencia y funcionamiento de éstos.

Como se mencionó en el apartado anterior, los algoritmos propuestos para el problema de optimización del SUH son generalmente dos y podríamos dividirlos en ACO_actividades y ACO_pacientes. Los parámetros que componen un ACO son:

- Numero de hormigas ($n_{hormigas}$): define el número de soluciones (hormigas) diferentes que se generan en cada iteración. Posteriormente se compararán para seleccionar, entre todas ellas, la que aporta el mejor resultado de nuestro problema. La matriz de feromonas se actualiza una vez con cada iteración, y consta de dos pasos, primero se evaporan una determinada cantidad feromonas de toda la matriz y posteriormente se suma un factor de calidad únicamente en aquellos puntos que componen el camino de la hormiga con mejor solución.
- Tasa de evaporación (ρ): este parámetro indica la velocidad a la que se va a ir evaporando la feromona. Si aumentamos este valor, las feromonas se evaporarán a más velocidad dando pie a una mayor exploración de nuevos caminos, en cambio una tasa de evaporación demasiado alta podría impedir que el algoritmo converja a una única mejor solución. Por el contrario, un valor bajo de este parámetro puede ayudar a reforzar los mejores caminos encontrados anteriormente, encontrando más rápido la mejor solución, pero también puede ser que se pierdan buenas soluciones si la convergencia ocurre demasiado pronto.
- α: este parámetro indica la influencia de la feromona a la hora de seleccionar la próxima actividad de cada hormiga y por tanto el camino completo. Al aumentar su valor se promueve la exploración más intensiva de rutas ya existentes. Su valor recomendado es de 0 a 5.
- β: con este parámetro se controla la influencia de la información heurística aportada al algoritmo. Un valor más alto de β hará que la decisión de la hormiga de escoger el próximo nodo se base más en la información heurística aportada, promoviendo así la exploración de caminos que pueden ser buenos según esa información. La información heurística viene dada por la inversa del costo de la transición, de tal manera que cuanto mayor sea el coste, menos probabilidad de que la hormiga escoja ese camino.
- q_0: este parámetro puede tomar un valor entre 0 y 1 y se comparará con un valor *random*, cada vez que la hormiga tiene que decidir el siguiente nodo al que se va a mover. En caso de que ese valor *random* sea menor que q0, la decisión se tomará en función del mejor resultado de entre todas las posibilidades siguiendo la fórmula (1). En caso contrario, se escogerá el siguiente nodo en función de la probabilidad que tiene la hormiga de ir a ese nodo, según la fórmula (2).

Los parámetros que evaluamos fueron el número de hormigas que se genera por iteración y q_0, el resto de los valores de α, β y ρ se establecieron conforme a [36] para reducir el tiempo del análisis. Además se establecerá un tiempo de ejecución fijo para cada problema según la fórmula (7) [31]:

$$tiempo\ de\ ejecución\ = \left(\frac{número\ de\ recursos\ x\ numero\ de\ actividades}{2}\right) x\ 375\ [milisegundos] \quad (7)$$

En [36] resuelve el problema de Flow shop en talleres con múltiples objetivos y para ello emplea el algoritmo de colonia hormigas, siendo la misma base que el problema de estudio del presente proyecto. [36] realizó 10 pruebas de evaluación con los valores de α ∈ {0, 0.5, 1, 2, 5} , β ∈ {0, 0.5, 1, 2, 5 }, ρl ∈ {0, 0.1, 0.2, 0.3, 0.5} (tasa de evaporación local), ρg ∈ {0, 0.1, 0.2, 0.5} (tasa de evaporación global) y q_0 ∈ { 0, 0.25, 0.5, 0.75, 0.9}, fijando los valores de nº de hormigas a 20 y tiempo máximo de ejecución del algoritmo t_{max} =1000. Los mejores valores del análisis computacional con único objetivo minimizar el tiempo total de los procesos fueron: $\alpha = 2$, $\beta = 0.5$, $\rho l = 0.2$, $\rho g = 0.2$ y $q_0 = 0.9$, estableciendo estos parámetros como fijos en el resto de

evaluación del problema. En el presente trabajo se emplearán los mismos parámetros a excepción del valor de q_0 el cual se ha decidido calibrar entre los valores q_0 = {0.4, 0.6, 0.8, 0.9}. El motivo de variar este parámetro es por ser un valor importante en las decisiones de las hormigas, de esta forma podremos estudiar a ver si dando más posibilidad a la etapa de exploración (que con un q_0 de 0.9), el algoritmo ofrece mejores resultados (recordemos que en caso de que un valor random entre 0 y 1 sea menor que q_0, las hormigas usarán la información existente y en caso contrario usarán una regla probabilística para elegir de forma pseudo-random la siguiente transición). Por otro lado, a diferencia del ACO de [36], en nuestro algoritmo solo actualizamos las feromonas de forma 'global', es decir, solo se actualiza la matriz de feromonas con la mejor solución de entre todas las hormigas de una iteración, por lo que solo tendremos un único parámetro de tasa de evaporación ρ fijado a 0.2.

Para la calibración se han ejecutado con cada algoritmo 10 instancias de cada problema. Los escenarios que hemos estudiado son los recursos disponibles que hay a las 12h o a las 6h, y 3 niveles de saturación 50%, 75% y 100%, haciendo un total de 6 escenarios o problemas siendo un total de 60 experimentos con cada algoritmo para la calibración.

A continuación, mostraremos los resultados de esta fase de experimentación y estableceremos los parámetros con mejores resultados para cada caso.

5.2.2 Calbración q_0

Los valores escogidos para estudiar este parámetro son el 0.4, 0.6, 0.8 y 0.9. En la literatura generalmente el valor más escogido para este parámetro es el último, pero en el presente estudio se comprobó que dándole más oportunidad al algoritmo a que explore, acababa encontrando el óptimo antes o incluso le daba tiempo a estancarse en él, mientras que con valores más altos se quedaba en soluciones subóptimas.

A continuación, se muestran los resultados de los 2 escenarios que presentamos en el trabajo, realizando 30 pruebas con cada uno con diferentes niveles de saturación. Además, para cada valor de q_0 se han probado distintos valores de número de hormigas, teniendo más información de cómo se comporta el algoritmo con cada valor del parámetro a estudiar.

- **Entorno de las 12h:**

Abordamos primero el comportamiento de los algoritmos para distintas q_0. Tras los resultados de 30 instancias para el escenario de las 12h y estando el SUH al 50%, 75% y 100% de su capacidad, se han ejecutado los algoritmos para los valores de q_0 = {0.4, 0.6, 0.8, 0.9} y para cada uno de ellos se ha estudiado el comportamiento con los valores de $n_{hormigas}$ = {50, 100, 500, 1000 }. La media de los resultados de todos los algoritmos se muestra en la Figura 16.

Figura 16. Resultados de distintas q_0 en el turno de las 12h

La línea roja representa la primera solución inicial que aportamos al algoritmo (punto de inicio) para que arranque con una referencia factible (por eso se muestra constante en todos los valores). Como vemos los algoritmos fácilmente mejoran esta primera solución, encontrando los valores más bajos de la función objetivo con una q_0 = 0,4. Si bien es cierto que tras encontrar un pico en 0,8 las soluciones vuelven a mejorar con el valor que más abunda en la literatura para este parámetro, es evidente que nuestro problema en este entorno funciona mejor con un valor de q_0 más bajo, y por tanto con una mayor probabilidad de explorar el entorno de soluciones y no explotar las soluciones encontradas por las hormigas iniciales.

- **Entorno de las 6h:**

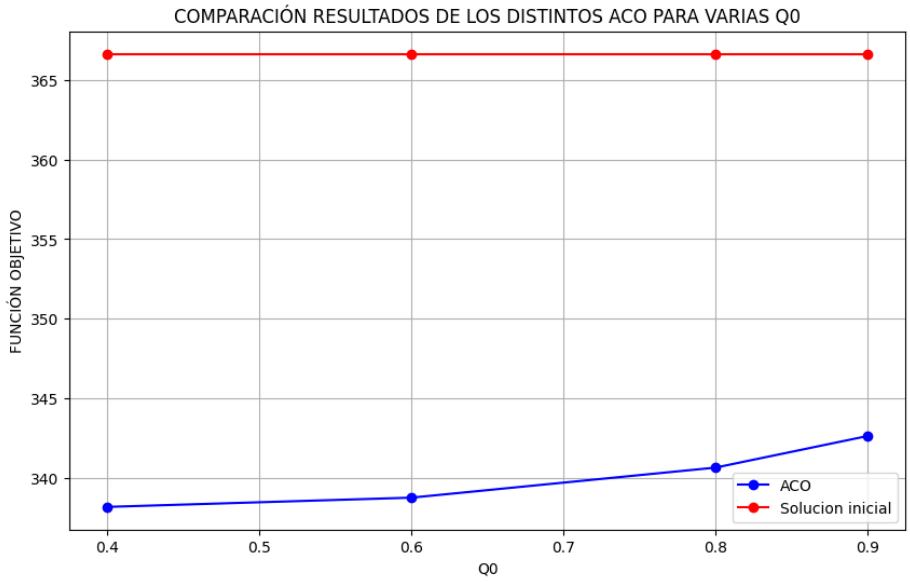

Figura 17. Resultados para distintas q_0 en el turno de las 6h

Nos situamos ahora en el SUH a las 6h en el cual encontramos menos recursos disponibles (9), y por tanto menos cantidad de pacientes y actividades necesarias para la saturar el SUH. Al encontrarnos en un problema más pequeño que el anterior, las posibles distintas combinaciones para las hormigas se reducen, siendo más sencillo encontrar la mejor solución del problema (o encontrarla antes). Esto explica que veamos en la Figura 17 una curva mucho menos pronunciada para este escenario, ya que los algoritmos no necesitan explorar mucho el entorno para encontrar la mejor solución.

Aun así, aunque con diferencias mínimas entre valores, observamos que el mejor comportamiento se encuentra con una q_0 = 0,4, al igual que en el escenario de las 12h.

5.2.3 Calibración $n_{hormigas}$

El número de hormigas determina la cantidad de hormigas que genera nuestro algoritmo en cada iteración. Recordemos que nuestro algoritmo funciona con una matriz de feromonas, la cual se actualiza con cada iteración en función de la mejor solución encontrada entre todas las hormigas generadas en ésta. Cuantas más feromonas haya por un camino, significa que es mejor frente a otros con menos cantidad de feromonas. Es por esto que, cuantas más hormigas se generen por iteración, más tiempo de computación necesario para que el algoritmo converja en alguna solución. Teniendo en cuenta que, para un tiempo fijo, cuanto mayor sea el $n_{hormigas}$ menos iteraciones realizará, actualizando menos veces la matriz de feromonas (menos información), y explotando menos el entorno. En cambio, cuantas más hormigas genere en una iteración, más capacidad de exploración a encontrar nuevas soluciones, y más posibilidades de encontrar la mejor solución en cada iteración.

Generalmente, si el problema es grande, puede ser conveniente un mayor número de hormigas para darle la capacidad al algoritmo a explorar todas las posibles soluciones, pero si es pequeño, puede ser innecesario generar tantas hormigas en cada vuelta ya que no notaríamos ninguna diferencia significativa a medida que avanzan esas iteraciones.

Dado que el tiempo de computación del algoritmo se fija en función de la magnitud del problema, la diferencia radica en el número de iteraciones que puede realizar el algoritmo con las diferentes cantidades de hormigas, la convergencia que tiene cada caso en encontrar la mejor solución y la calidad de estas soluciones. Debemos encontrar un equilibrio entre estos factores para determinar nuestro número de hormigas.

En el escenario de las 12h, podemos encontrar alrededor de 35 actividades (8 pacientes) cuando está al 50%, 58 actividades (11 pacientes)cuando está al 75%, y una media de 85 actividades (16 pacientes) cuando está al 100%. Por otro lado, en el escenario de las 6h encontramos que el problema varía de media entre 15 actividades (3 pacientes) cuando está al 50%, 24 actividades (5 pacientes) al 75% y 32 actividades (6 pacientes) cuando se encuentra al 100%.

- **Entorno de las 12H**

En la situación del SUH a las 12h, tenemos 15 recursos en total, dándonos una variedad de media entre 30 actividades hasta 90 actividades. Esto se traduce en que el tiempo de computación varía desde 84,375 seg a 253,125 seg siguiendo la fórmula (7). En la Figura 18 podemos observar la tendencia de los algoritmos a medida que se aumenta el número de hormigas con un tiempo fijo.

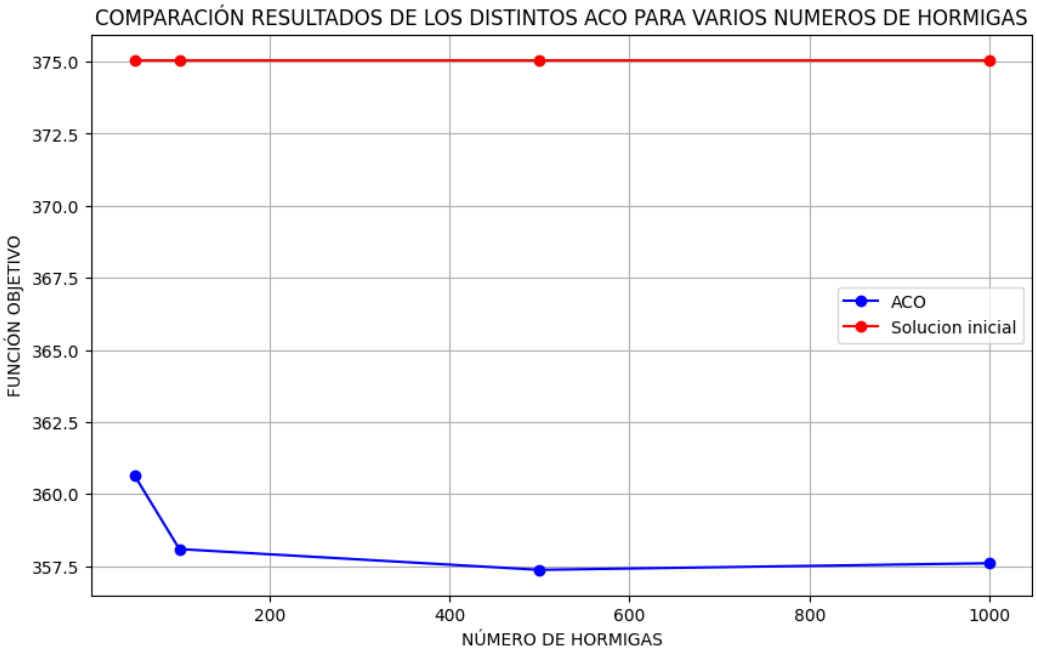

Figura 18. Resultados de los ACO para distintos valores de $n_{hormigas}$ en el escenario de las 12h

En el eje X encontramos los distintos valores de hormigas estudiados, y en el eje Y la media de la FO obtenida con este parámetro. Se puede observar que, para el turno donde encontramos una mayor afluencia de pacientes que es la situación de las 12h, la calidad de la solución mejora al aumentar el número de hormigas (menor FO).Aunque de igual forma, la diferencia de la solución entre 100 y 1000 no se hace significativa, es probable que dándole más tiempo de computación al algoritmo con más hormigas pueda dar mejores resultados. En el entorno en el que trabajamos el tiempo es algo crucial y de gran importancia, por tanto, la duración total de la computación no debería de ser mayor del tiempo en el que tiene que ser atendido un paciente de prioridad 2 (ya que el paciente 1 no puede esperar más de 1 minuto en ser atendido), esto son 15 minutos cómo máximo. Si suponemos este tiempo máximo con 1000 hormigas, el algoritmo hace casi 7 iteraciones, por lo que, aunque da buenos resultados, no llega a converger hacia ninguna solución en ese tiempo. Por este motivo, estableceremos el número de hormigas del turno de las 12h a 500 hormigas por vuelta.

- **Entorno de las 6H:**

Figura 19. Resultados de los ACO para distintos valores de $n_{hormigas}$ en el escenario de las 6h

Nos situamos ahora en el turno de las 6 de la mañana, donde las urgencias cuentan con 9 recursos disponibles para atender a todos los pacientes que llegan. Es por esto, que las urgencias se saturan antes, siendo el problema que resolver mucho más pequeño que el turno de las 12h.

En la Figura 19 podemos observar cómo los algoritmos dan buenos resultados entre 100 y 500 hormigas, empeorando la FO al aumentarlas a 1000.

Como comentamos anteriormente, en problemas pequeños un mayor número de hormigas puede ser innecesario, incluso puede dar lugar a converger demasiado rápido hacia una solución sub-óptima. Es por esto que, aunque las diferencias sean mínimas entre los valores, estableceremos el número de hormigas del turno de las 6h a 100 hormigas.

6 RESULTADOS

A lo largo del siguiente capítulo abordaremos el análisis de los algoritmos y sus variantes con la finalidad de observar el comportamiento de éstos ante las situaciones expuestas. Para ello se han evaluado diferentes parámetros entre los que se encuentran la estanqueidad (cuyos resultados manifiestan la necesidad de una búsqueda local), la capacidad para obtener nuevas mejores soluciones y el ARPD (Averange Relative Percentage Deviation) [13,25,31,34].

Es importante mencionar que el proceso de análisis se comenzó con observar el estancamiento de los algoritmos ACO1, ACO2, ACO3 y ACO4, cuyos resultados manifestaron la necesidad de añadir a éstos una búsqueda local. A raíz de esto, se continuó con el análisis con los 4 ACO básicos más los 4 ACO con la búsqueda local elaborada (ver secciones 4.2 y 4.3).

Una vez calibrados los parámetros de los algoritmos, para cada caso de los próximos análisis estableceremos aquellos valores de los parámetros que han dado los mejores resultados. Además, el tiempo de computación que se emplea en esta parte es el máximo que se permite en el SUH que son 15 minutos. Un resumen de estos valores se ven en la tabla 6:

Parámetro	Entrono de las 12h	Entorno de las 6h
$n_{hormigas}$	500	100
q_0	0,4	0,4
Tasa de evaporación (ρ)	0,2	0,2
α	2	2
β	0,5	0,5
Tiempo de computación	15 min	15 min

Tabla 6. Parámetros empleados en el análisis de los algoritmos

Para este estudio se han ejecutado 60 instancias con cada algoritmo: 30 pertenecientes al turno de las 12h y 30 pertenecientes al turno de las 6h. Para una mejor visualización del comportamiento de los ACO, se ha establecido el máximo tiempo de computación permitido para todas las situaciones, esto es el máximo tiempo que un paciente de nivel ESI 2 debería esperar para ser atendido por primera vez, es decir 15 minutos en total.

6.1 Estanqueidad

En este apartado valoraremos la eficacia de las diferentes variantes del algoritmo de optimización propuesto en términos de estanqueidad. En nuestro contexto, la estanqueidad hace referencia a la rapidez con la que el algoritmo converge hacia una solución, esto incluye la información de cuántas iteraciones necesita para encontrar la mejor solución y si se queda estancado en un valor local sin mejorar significativamente durante muchas iteraciones.

Para valorar esto definimos tres valores:

- Número de iteraciones para encontrar la mejor solución (NIMS).
- Tasa de convergencia: cuánto varía la calidad de la solución en cada iteración.
- Duración del estancamiento: el número de iteraciones durante las cuales la solución no mejora.

A continuación, mostraremos los resultados de los distintos ACO desarrollados para cada situación del SUH.

HORA	SAT %	ALGORTIMO	NIMS	Tasa de Convergencia	Número de iteraciones estancadas	Media de iteraciones de la situación
12	0,5	ACO1	3,4	-0,153	22,3	31
12	0,75	ACO1	7,7	-0,720	14,1	22
12	1	ACO1	5,4	-1,500	9,1	12
12	0,5	ACO2	3,1	-0,126	24,8	31
12	0,75	ACO2	7,2	-0,659	15,6	22
12	1	ACO2	4,5	-0,925	10	12
12	0,5	ACO3	3,2	-0,092	25,4	31
12	0,75	ACO3	8,5	-0,457	15,8	22
12	1	ACO3	4,9	-1,749	9,7	12
12	0,5	ACO4	1,1	-0,006	25,7	31
12	0,75	ACO4	2,4	-0,044	17,4	22
12	1	ACO4	3	-0,252	10,9	12

Tabla 7. Resultados análisis estanqueidad en el turno de las 12h

Desglosamos los resultados por turno y por nivel de saturación, ya que, según el tamaño del problema, para un tiempo de computación fijo, el algoritmo ACO hará un número variable de iteraciones. En el entorno de las 12h, para una saturación del 50% con un tiempo de ejecución máximo de 15 minutos, los algoritmos realizan

alrededor de 31 iteraciones. Para una saturación de 75% realizan alrededor de 22 y para el 100% de saturación los algoritmos dan 12 iteraciones. En la Tabla 8 se encuentran los resultados de este entorno.

Observamos que en general el número de iteraciones hasta la mejor solución (NIMS) es generalmente bajo para saturaciones bajas, lo que indica que los algoritmos encuentran la mejor solución rápidamente. A medida que crece el problema vemos que tarda más en encontrar la mejor solución, destacando que converge más lento con una saturación del 75% que del 100%.

Por otro lado, los valores de tasa de convergencia son generalmente bajos, indicando que hay poca mejoría de los resultados tras cada iteración, sin embargo, es importante señalar que a medida que crece el tamaño del problema, también aumenta la mejora por iteración. Esto se justifica porque al haber mayor número de actividades, el número de combinaciones válidas de estas también crece, lo que hace más probable que, tras cada combinación, se encuentren soluciones distintas que superen a la mejor solución hallada hasta ese momento. Por último, en la duración del estancamiento debemos tener en cuenta las iteraciones que realizan los ACO en cada situación, observando que dos tercios de las iteraciones de la ejecución son iteraciones que no mejoran la solución.

Atendemos ahora los resultados relevantes al turno de las 6h (Tabla 8).

HORA	SAT %	ALGORTIMO	NIMS	Tasa de Convergencia	Número de iteraciones estancadas	Media de iteraciones de la situación
6	0,5	ACO1	1,6	-0,001	274	450
6	0,75	ACO1	2,1	-0,014	187,5	270
6	1	ACO1	1,2	-0,006	162,4	180
6	0,5	ACO2	1,4	-0,003	289,4	450
6	0,75	ACO2	2	-0,006	200,9	270
6	1	ACO2	1,6	-0,008	168,4	180
6	0,5	ACO3	1,4	-0,003	291,8	470
6	0,75	ACO3	3,3	-0,012	201,2	270
6	1	ACO3	1,2	-0,001	168,6	180
6	0,5	ACO4	1	0	263,8	470
6	0,75	ACO4	1,1	-0,001	190	270
6	1	ACO4	1	0	161,8	180

Tabla 8. Resultados análisis estanqueidad en el turno de las 6h

Recordemos que el número de hormigas en esta situación se reducía a 100 por iteración, permitiendo realizar más iteraciones en cada ejecución. Encontramos, por lo tanto, que para en la situación del 50% de saturación, los algoritmos alcanzan a realizar 450 iteraciones. Para una saturación del 75% hacen alrededor de 270 iteraciones y para el 100% nos encontramos en torno a 180 iteraciones.

Sabiendo esto, observamos en los resultados equivalencias con el escenario anterior dentro de las variaciones de cada uno. Vemos que de igual manera el ACO4 es el que más pronto encuentra la mejor solución estancándose en ésta durante casi toda la ejecución y sin mejorar tras las iteraciones (quedándose probablemente en una solución subóptima, lo cual valoraremos en la siguiente sección). Encontramos también la misma peculiaridad en la situación del 75% de saturación, en ésta tarda más rondas en converger hacia una solución que con el SUH al 50% o al 100%. Al tratarse de una situación del SUH con poca actividad, era de esperar que los algoritmos encontraran aquella solución óptima rápidamente, lo que lleva a converger poco tras cada iteración (ni mejorando ni empeorando) y traduciéndose en una larga duración del estancamiento, prácticamente el 80% de las iteraciones están estancadas sin mejorar la solución encontrada por las hormigas.

En conjunto, los resultados muestran una rápida convergencia a la mejor solución, pero suponiendo un gran estancamiento en la mayoría de las iteraciones de la ejecución. Viendo que la mejoría de las soluciones es mínima con cada vuelta, nos sugiere que probablemente haya soluciones que los algoritmos no estén explorando.

Partiendo de esto y relativo a lo anterior, se decidió elaborar una búsqueda local para completar los algoritmos y comprobar si los resultados obtenidos son mejorables. Esta variante inicialmente no fue valorada por restricciones de tiempo, aunque finalmente resultó ser un algoritmo bastante rápido como para incluirlo en los ACO tras cada iteración y no percibir apenas una disminución en el número de iteraciones.

La búsqueda local codificada tarda en ejecutarse alrededor entre 0.68 y 1 segundo, por lo que, si nuestro algoritmo de optimización realiza 30 iteraciones, solamente sumará al tiempo de computación medio minuto, siendo un coste pequeño para los beneficios obtenidos

Como podemos ver en la Tabla 10, aunque no se muestre una gran mejoría en la duración del estancamiento, y generalmente tarde más en encontrar la mejor solución, las iteraciones muestran una gran mejora con cada una de ellas, lo que nos da una idea preconcebida de que seguramente encontrarán mejores soluciones las variantes con búsqueda local que sus versiones simples, principalmente en el turno de las 12h. En lo que sigue estudiaremos esta comparativa más en detalle, apoyando los resultados obtenidos en este análisis.

HORA	SAT %	ALGORTIMO	NIMS	Tasa de Convergencia	Número de iteraciones estancadas	Media de iteraciones de la situación
12	0,5	BL+ACO1	6	-2,254	19,7	30
12	0,75	BL+ACO1	7,1	-2,436	14,5	20
12	1	BL+ACO1	7	-4,906	9,22	10
12	0,5	BL+ACO2	6,9	-3,521	21,7	30
12	0,75	BL+ACO2	8,44	-3,551	16,44	20
12	1	BL+ACO2	7,22	-4,732	9,55	10
12	0,5	BL+ACO3	5,1	-2,427	22	30
12	0,75	BL+ACO3	4,2	-1,367	17	20
12	1	BL+ACO3	7,22	-4,732	9,55	10
12	0,5	BL+ACO4	4,5	-0,011	22,1	30
12	0,75	BL+ACO4	2,3	-0,014	20,2	20
12	1	BL+ACO4	3,22	-0,102	9,33	10
6	0,5	BL+ACO1	1,7	-0,010	247,1	450
6	0,75	BL+ACO1	3,8	-0,023	189,7	250
6	1	BL+ACO1	1,67	-0,021	131,16	160
6	0,5	BL+ACO2	1,6	-0,008	270,9	450
6	0,75	BL+ACO2	3	-0,016	201,9	250
6	1	BL+ACO2	1,83	-0,0109	142,5	160
6	0,5	BL+ACO3	1,9	-0,008	274,1	450
6	0,75	BL+ACO3	1,7	-0,004	204,5	250
6	1	BL+ACO3	1,22	-0,166	99,22	160
6	0,5	BL+ACO4	1	0	276,6	450
6	0,75	BL+ACO4	1	0	210,1	250
6	1	BL+ACO4	1	0	144,83	160

Tabla 9. Resultados del análisis de estancamiento de los algoritmos con búsqueda local

6.2 ARPD y Mejores soluciones encontradas

En la presente sección, valoraremos la validez y efectividad de los algoritmos implementados a lo largo del trabajo. Se compararán las 4 variantes del algoritmo de colonia de hormigas cuyas diferencias residen en la selección de actividades iniciales y la configuración de las hormigas. Además, se estudiarán junto a estas las variantes añadiéndoles la búsqueda local (capítulo 4, punto 4). Para realizar este análisis se han elegido los parámetros de ARPD y el de mejores soluciones encontradas. El Averange RPD sigue la fórmula (8):

$$RPD = \frac{Best_{sol} - Some_{sol}}{Best_{sol}} * 100 \quad (8)$$

$Best_{sol}$ hace referencia a la mejor solución encontrada de entre todos los algoritmos para un problema. $Some_{sol}$ es la mejor solución encontrada por cada algoritmo de estudio en ese mismo problema. A continuación, se realiza la media de los resultados obtenidos por cada algoritmo en los diferentes problemas estudiados obteniendo el porcentaje de desviación de cada algoritmo a la mejor solución obtenida. Por lo tanto, cuanto mayor sea su valor, más lejos se encuentra ese algoritmo de la mejor solución.

Por último, para el cálculo de mejores soluciones encontradas, se hace un recuento de cuántas veces se ha 'sustituido' la 'mejor solución' dentro del tiempo de computación establecido, para cada algoritmo.

Para este estudio se han ejecutado 60 instancias con cada algoritmo: 30 pertenecientes al turno de las 12h y 30 pertenecientes al turno de las 6h. Para una mejor visualización del comportamiento de los ACO, se ha establecido el máximo tiempo de computación permitido para todas las situaciones, esto es el máximo tiempo que un paciente de nivel ESI 2 debería esperar para ser atendido por primera vez, es decir, 15 minutos en total.

En la siguiente sección, dividiremos los estudios en las configuraciones del SUH a las 12h y a las 6h para una mejor comparativa de los resultados.

- **Entorno de las 12h:**

En la figura 14 podemos ver la capacidad de encontrar mejores soluciones por cada algoritmo en problemas de mayor tamaño del SUH según la configuración de las 12h. En el eje Y encontramos la media de mejores soluciones que encuentra cada algoritmo en el tiempo de ejecución definido. Esto quiere decir, que, en 15 minutos, los algoritmos encuentran como máximo 3,5 nuevas soluciones que sustituyan a las anteriores encontradas. Los resultados de las 30 instancias muestran que todos los algoritmos cuyos métodos se basan en hormigas constituidas por actividades (ACO1, ACO2 y ACO3) mejoran considerablemente este parámetro al añadir una búsqueda local tras cada iteración. Esto indica que la búsqueda local efectivamente ayuda al algoritmo a explorar más el entorno sin estancarse en óptimos locales.

Por un lado, ACO1+BL y ACO2+BL son los dos que más destacan en la búsqueda de mejores soluciones dentro del tiempo de computación. Por otro lado, como era de esperar en este análisis del algoritmo basado en pacientes (ACO4), tanto su versión simple como aquella a la que se le añade una búsqueda local son los algoritmos menos efectivos en encontrar nuevas y mejores soluciones. Esto se debe a que, al basar su configuración de hormigas en los pacientes, las combinaciones posibles son considerablemente menores. Como resultado, el algoritmo tiende a estancarse más rápidamente en una solución subóptima y, en muchos casos, encuentra la mejor solución posible de acuerdo con este enfoque con pocas iteraciones, siendo innecesarias más iteraciones ya que no es posible alcanzar mejor solución a la encontrada. Es interesante ver que, en este último algoritmo, su versión con búsqueda local no mejora a la versión sin esta.

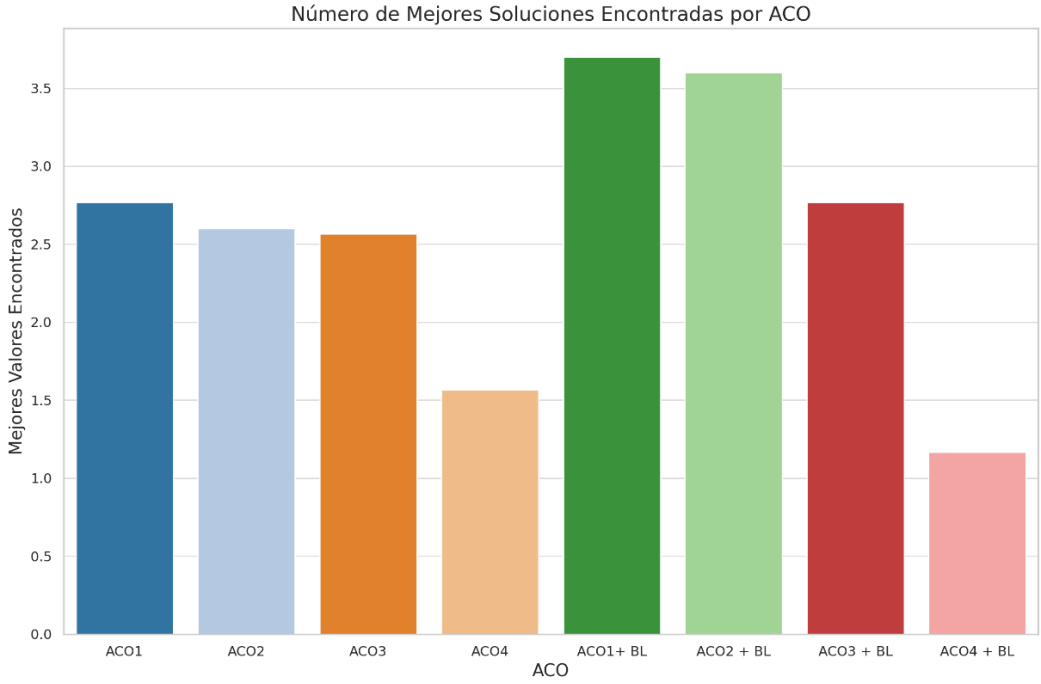

Figura 20. Número de mejores soluciones encontradas en el entorno de las 12h del SUH

Para completar la visión de la eficiencia de los algoritmos, analizamos el ARPD bajo distintos niveles de saturación (figura 21). En el eje X encontramos los porcentajes de los niveles de saturación y en el eje Y el valor de media del RPD de la fórmula (8).

Este parámetro muestra las desviaciones porcentuales relativas de las mejores soluciones encontradas por cada algoritmo frente a la mejor solución conocida en general, por ello, un valor más bajo de ARPD significa que las soluciones generadas por ese ACO se acercan mucho a la mejor solución, y por tanto es eficiente encontrando soluciones de alta calidad. Un valor nulo de este parámetro indica que siempre encuentra la mejor solución.

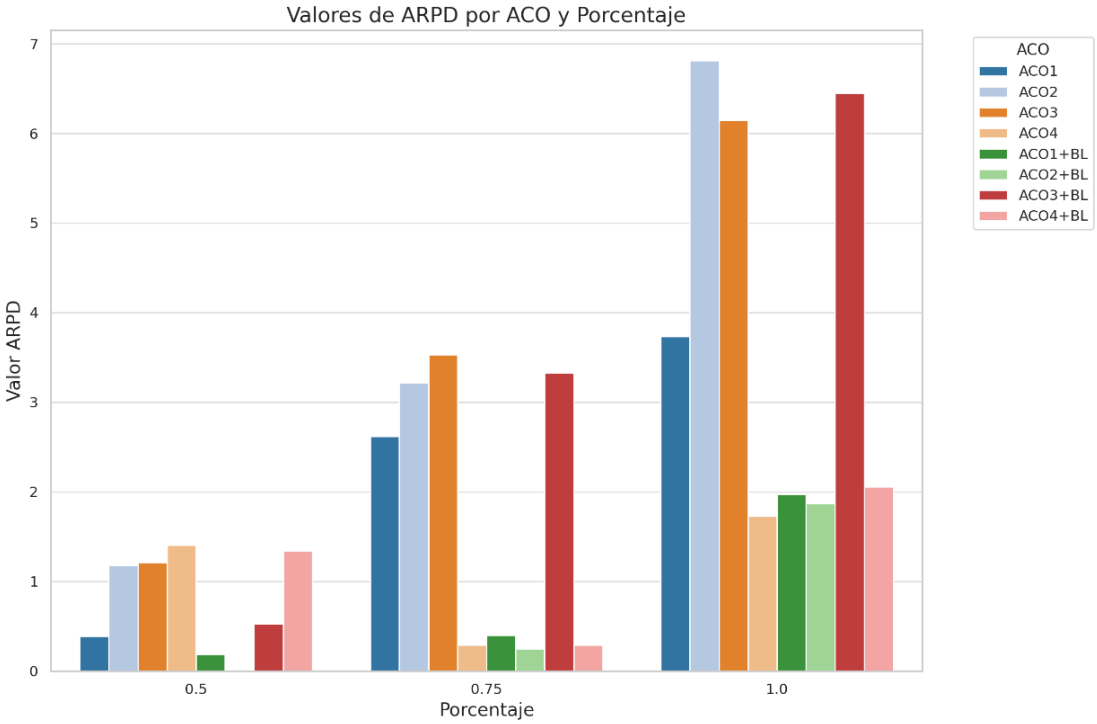

Figura 21. ARPD de los algoritmos en distintas situaciones del entorno de las 12h

En este análisis podemos observar que, en escenarios de baja saturación, ACO2+BL encuentra siempre las mejores soluciones (ARPD igual a 0%). Tras él le siguen el ACO1+BL y su versión sin búsqueda local. Por otro lado, el ACO3 muestra una gran mejora en cuanto a calidad de soluciones al añadir la búsqueda local, aunque menos eficiente que los anteriores.

Es importante destacar que el ACO4 y su variante muestran malos resultados en escenarios con baja saturación, pero destacan como los más eficientes frente a sus compañeros en situaciones con alta saturación. Con estos resultados enfrentados a los anteriores comprendemos que el comportamiento que siguen estos algoritmos al mover pacientes en vez de actividades es muy efectivo encontrando los valores más bajos de los problemas, aunque eso no necesariamente esté relacionado con encontrar muchas “mejores soluciones” dentro de un tiempo de computación.

En cuanto a los algoritmos que manejan actividades, observamos que la búsqueda local les beneficia mucho a medida que se aumenta la saturación del SUH, a excepción del ACO3, el cual muestra los peores resultados, sin notar apenas el beneficio al añadir una búsqueda local.

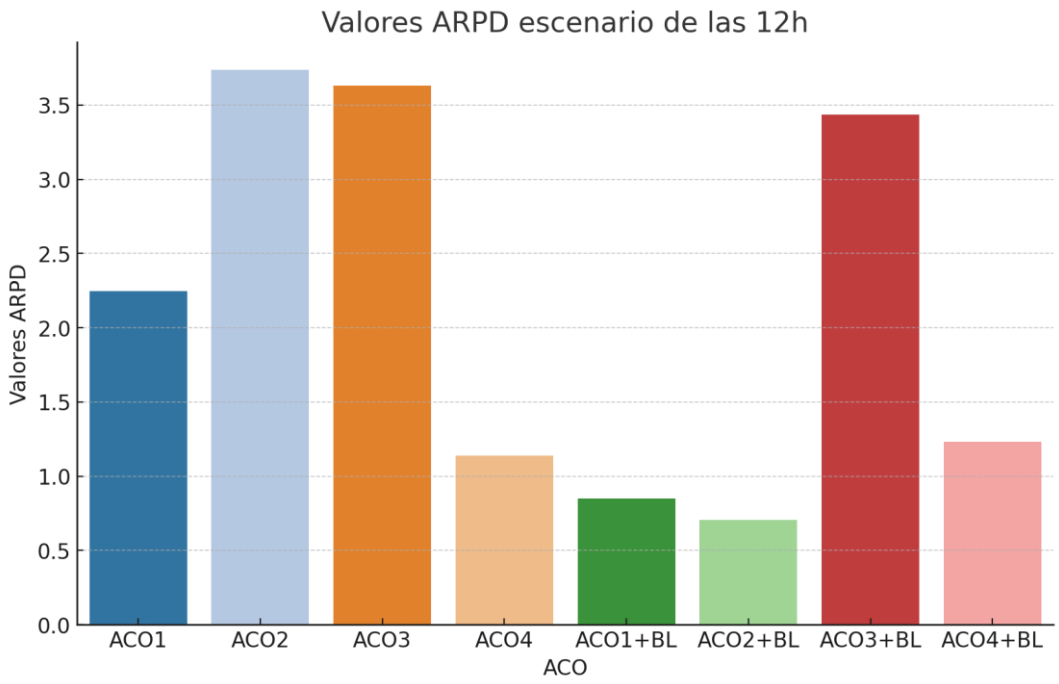

Figura 22. Valores ARPD globales del entorno de las 12h

En resumen, es evidente que la inclusión de una búsqueda local beneficia la búsqueda de mejores soluciones mejorando también la eficiencia de los algoritmos de hormigas. Aunque ACO4 no mejora con esta búsqueda local (por lo tanto, su inclusión en esta situación no sería beneficiosa) y tampoco destaca en términos de mejores soluciones encontradas, sí muestra una gran robustez y eficiencia en la calidad de sus soluciones, sobre todo con altas saturaciones, aunque no alcanza la mejor solución global (Figura 22).

Cabe destacar también que tanto ACO1+BL y ACO2+BL ofrecen muy buenos resultados en términos de mejores soluciones encontradas y en ARPD siendo algoritmos muy completos y robustos para el problema que trabajamos. Por último, el ACO3 consigue una gran mejoría al añadir la búsqueda local, pero no destaca frente a sus compañeros en ninguno de los parámetros valorados, encontrando generalmente soluciones bastante alejadas de la mejor solución global.

- **Entorno de las 6h**:

Nos situamos en el turno de las 6h del SUH , lo cual significa que encontramos disponibles menos recursos, siendo más fácil saturar las urgencias, suponiendo un problema de menor tamaño que resolver. Esto se muestra en los valores de mejores soluciones encontradas de la Figura 23, en el cual el algoritmo que más soluciones encuentra no llega a 2, frente a las del turno de las 12h que superan las 3.

De igual forma que en el turno anterior, los algoritmos que encuentran más soluciones son el ACO1+BL y el ACO2+BL, seguidos de sus variantes sin búsqueda local, aunque mostrando gran diferencia al añadir ésta. También vemos que los algoritmos ACO4 y ACO4+BL son los que menos encuentran, aunque no situándose lejos de sus compañeros, debido a que, al disminuir el problema, el número de soluciones válidas se ven reducidas a su vez. Por otro lado, es interesante ver que tanto al ACO3 como al ACO4 no les beneficia añadir una búsqueda local, en términos de mejores soluciones encontradas.

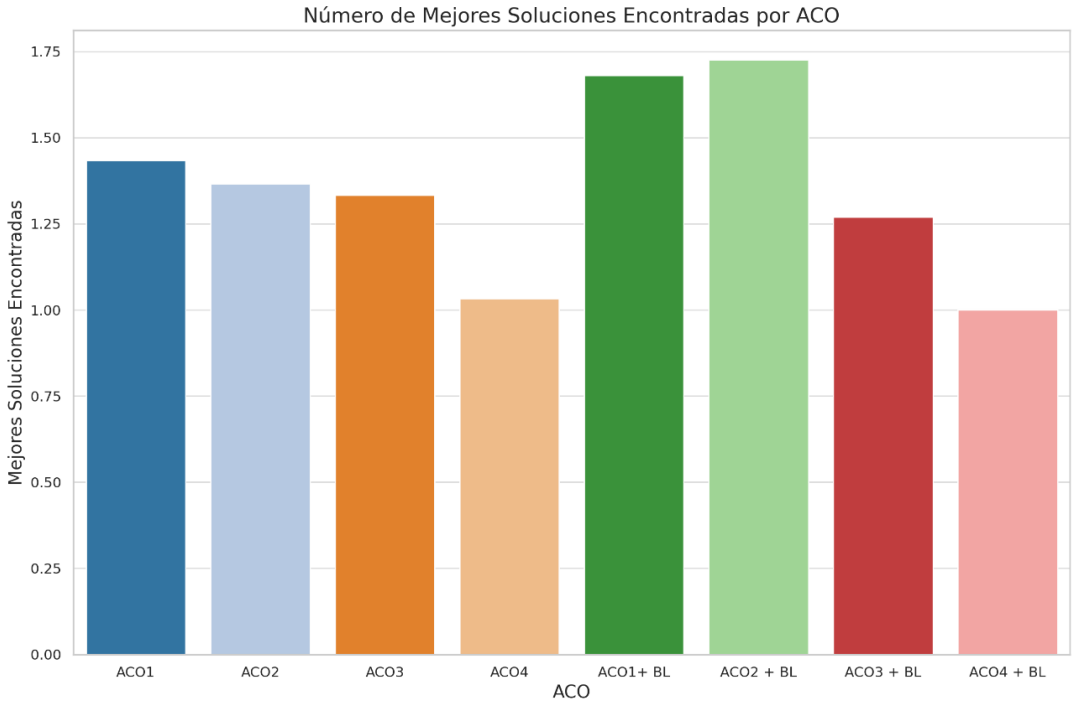

Figura 23. Mejores soluciones encontradas en el entorno de las 6h

Si observamos ahora el diagrama de barras de la Figura 24, encontramos gran diferencia entre los resultados de los diferentes algoritmos, algo que en la situación de las 12h no veíamos, la cual nos transmitía unos comportamientos más similares entre algoritmos. En la figura 25 podemos observar el ARPD global obtenido de los algoritmos en este entorno, apoyando la figura anterior en los resultados y clarificando el análisis.

Por un lado, encontramos que ACO4+BL muestra gran robustez en escenarios con alta saturación (ARPD = 0%). Por otro lado, aunque su variante no ofrezca los mejores resultados, se mantiene casi constante en todas las situaciones cerca de sus competidores ACO1 y ACO2. El algoritmo ACO3, aunque se ve en ocasiones mejorado por su variante con búsqueda local, los resultados no son generalmente buenos en ningún nivel de saturación, siendo el peor de los 4 algoritmos (Figura 25). Finalmente observamos que ACO1 y ACO2 se mantienen ofreciendo muy buenos resultados, aunque mejorando significativamente al añadirles la búsqueda local, consiguiendo prácticamente siempre la mejor solución encontrada por todos los algoritmos (ACO2+BL muestra un ARPD casi del 0%), por lo que se posicionan como los más efectivos y robustos en cualquier situación.

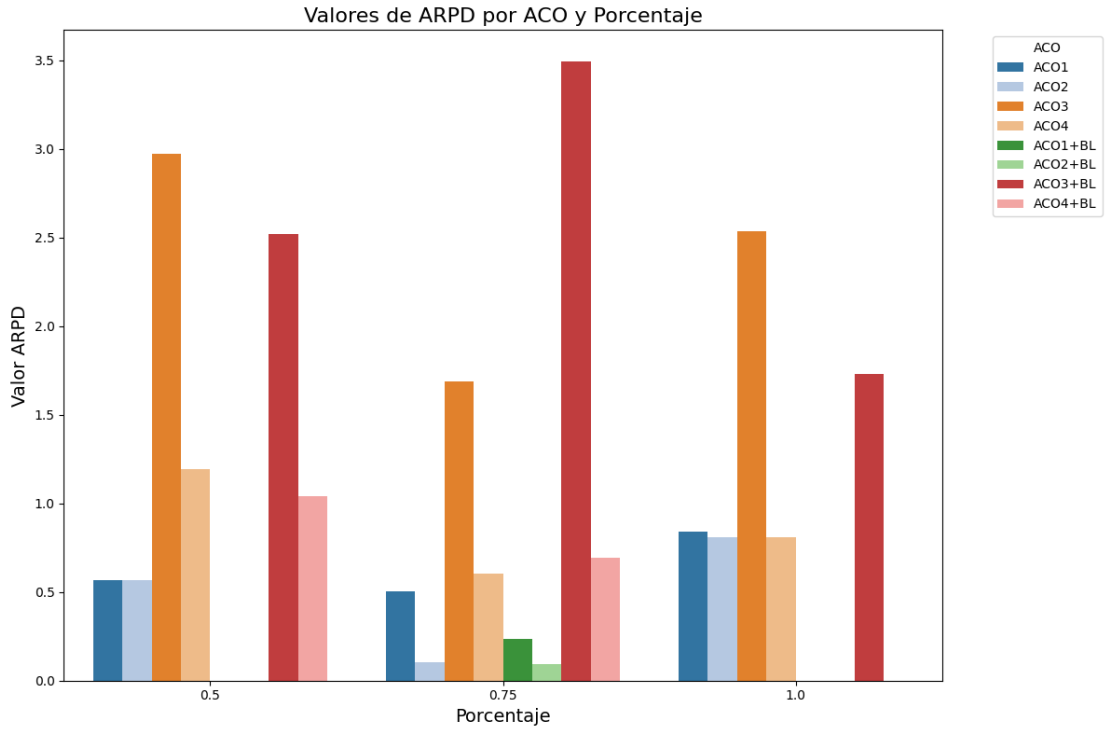

Figura 24. ARPD de los algoritmos en distintas situaciones del turno de las 6h

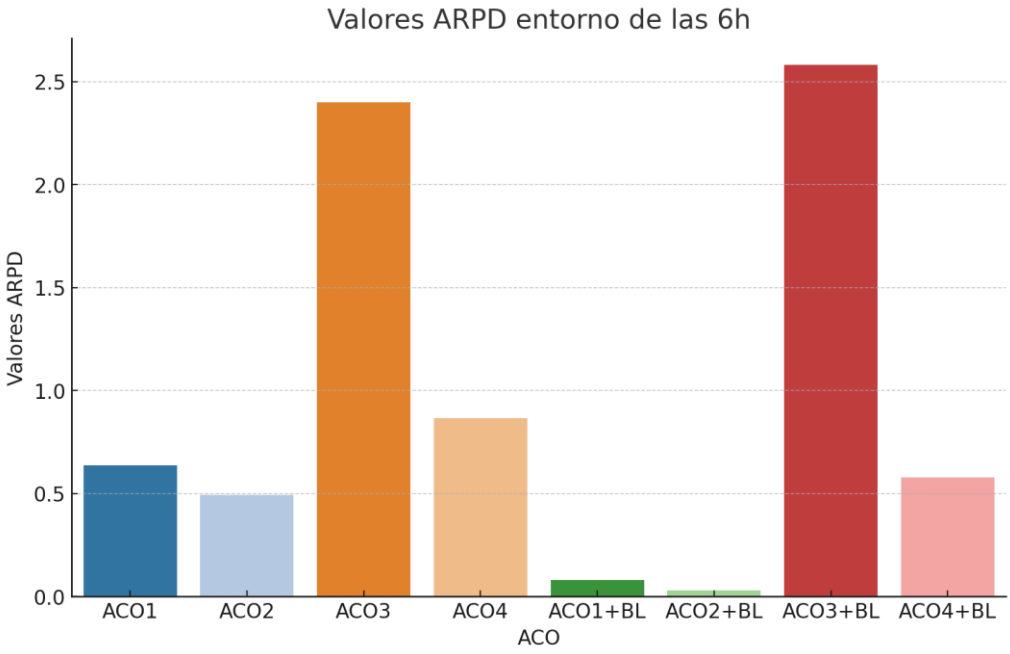

Figura 25. Valores ARPD globales para el entorno de las 6h

En conclusión, la búsqueda local ha resultado ser una herramienta crucial para la mejora de resultados y efectividad de los algoritmos de hormigas, sobre todo aquellos cuya constitución de hormigas se basa en actividades. El ACO4 destaca por su robustez en las soluciones en ambos turnos, siendo una opción muy interesante para aquellas situaciones con alta saturación en las que el tiempo es escaso, y buscamos menos exigencia en los resultados a cambio de una buena solución rápida, ya que si al encontrar la mejor solución

(posible por este algoritmo) a las pocas iteraciones, se podría disminuir el tiempo de ejecución al mínimo marcado por la fórmula 8 y obtener los mismos resultados. En cambio, si necesitamos algoritmos donde con seguridad se encontrarán las mejores soluciones posibles, ACO1+BL y ACO2+BL son las opciones más efectivas. ACO3 y su variante se quedan muy atrás de los otros algoritmos, siendo, aunque efectivos encontrando mejores soluciones, poco robustos a la hora de acercarse a la mejor solución global posible.

7 CONCLUSIONES

Debido a la sobresaturación de los servicios sanitarios, la necesidad de buscar nuevas soluciones que les permita mejorar la gestión de éstos. Uno de los servicios más afectados es el SUH. el cual recibe al día cientos de pacientes con distintos niveles de urgencia y distintos motivos patológicos a los que debe atender en el mínimo tiempo posible. En los últimos años la demanda en los SUHs ha ido en aumento, y con ello los estudios enfocados en optimizar y mejorar estos servicios.

En el presente trabajo se ha abordado el problema de optimización del SUH a través de una correcta secuenciación de actividades y/o pacientes en el tiempo. Para ello se ha optado por la adaptación de algoritmos ACO a nuestro problema, siendo una metaheurística nunca empleada antes con este objetivo, e interesante de estudiar debido a sus buenos resultados en otros campos.

Para el desarrollo del trabajo fin de máster se ha elaborado una función llamada 'decoding' con la cual se aborda la problemática de secuenciación de actividades y asignación de éstas a recursos. La función objetivo que se plantea pretende minimizar tanto el LOS como el TEPCOF de los pacientes. Para ello se ha optado por una programación imperativa y estructurada en lenguaje Python por su simplicidad y legibilidad, además de tener multitud de librerías que permiten implementar este tipo de problemas. Posteriormente, se ha adaptado la metaheurística ACO a nuestro problema con diferentes versiones: 3 versiones cuya configuración de la solución se basa en las actividades y 1 versión basada en los pacientes. Además, se ha elaborado una búsqueda local para su posterior implementación en los 4 ACOs y comparación de resultados.

Las diferencias entre los ACO1, ACO2 y ACO3 reside en la forma que las hormigas comienzan a generar una nueva solución, aunque las tres variantes componen sus hormigas de las actividades de los pacientes. El ACO1 escoge la primera actividad de cada hormiga según la regla de transición empleada por el algoritmo de hormigas. Por otro lado, ACO2 selecciona esa primera actividad de forma aleatoria entre las primeras actividades de todos los pacientes (para cumplir siempre los órdenes de presencia). Y el ACO3 empieza siempre por la primera actividad de la mejor solución (hormiga) que se ha encontrado hasta ese momento. Por último, a diferencia de estos, el ACO4 genera sus hormigas con los pacientes, de forma que no tiene restricciones de precedencia que cumplir como los anteriores.

El empleo del ACO en sus distintas variantes ha supuesto una mejora significante para la gestión del flujo de pacientes y ha permitido analizar el problema y el comportamiento en diferentes escenarios del SUH. En los resultados han destacado los algoritmos ACO1 y ACO2 mostrando un rendimiento por encima de los demás y manteniendo los mejores resultados en la mayoría de las situaciones estudiadas. Estos algoritmos han conseguido optimizar notablemente la FO al incorporar la búsqueda local tras cada iteración, logrando la mayoría de las veces la mejor solución encontrada entre todos los algoritmos evaluados. Esto posiciona a

ACO1+BL y ACO2+BL como las mejores opciones para obtener soluciones efectivas y robustas en la secuenciación de actividades en el SUH, garantizando nuestro objetivo principal que es minimizar los tiempos del SUH.

Por otro lado, el ACO4 ha mostrado un comportamiento robusto en cuanto a obtener buenas soluciones de forma rápida, siendo una opción interesante en situaciones del SUH con alta saturación en las que el tiempo es reducido. También muestra este comportamiento su variante con búsqueda local, aunque puede ser prescindible debido a la casi imperceptible variedad de los resultados con su variante básica, incluso llegando a empeorar los resultados ligeramente en entornos con muchos pacientes.

En cambio, el ACO3 y su variante ACO3+BL no han alcanzado los resultados esperados posicionándose en último lugar. Aunque han resultado efectivos a la hora de encontrar nuevas soluciones, apenas consiguen acercarse a la mejor solución global posible, denotando falta de robustez, y siendo una mala solución para el problema del SUH en comparación con sus compañeras.

Estos resultados han demostrado que la herramienta de búsqueda local es crucial en la optimización de resultados y ha mostrado un cambio efectivo en los ACO, especialmente en aquellos cuyas configuraciones de hormigas se componen de actividades. Esta búsqueda local ha permitido no solo intensificar la búsqueda de mejores soluciones, además aporta una capacidad de exploración del espacio de soluciones que permite obtener los mejores resultados.

En conclusión, se ha demostrado que la adaptación del ACO junto con una búsqueda local a la problemática de secuenciación de actividades y asignación de estas a recursos del SUH es una buena opción para la optimización de los tiempos de estancia de los pacientes y la tardanza en ser atendidos por primera vez. Estos algoritmos han permitido una flexible adaptación del problema estudiado permitiendo un análisis completo del entorno, denotando gran calidad en las soluciones y pudiendo abordar los distintos niveles de carga que se encuentran en el SUH (hasta un 100%). Se ha conseguido así, optimizar los tiempos, así como abordar soluciones factibles en tiempos considerables dentro del SUH para una toma de decisiones real, contribuyendo a la mejora de estos servicios.

Para futuros trabajos, sería interesante explorar otras heurísticas de construcción para la secuenciación de pacientes para continuar mejorando la eficiencia y efectividad en la gestión del SUH. En cuanto a la optimización con colonia de hormigas, se ha observado que los parámetros que emplea en su configuración afectan a la calidad de las soluciones. En el presente trabajo se optó por evaluar 2 parámetros de 5, mostrando resultados distintos a los comúnmente empleados en la literatura, siendo interesante en futuros trabajos una mayor profundidad de análisis de todos para una conseguir una total adaptabilidad al problema estudiado. Por último, sería de gran interés elaborar una búsqueda local que explorase el entorno de soluciones con las actividades en vez de pacientes como se ha elaborado en el trabajo, en la cual sacase las actividades de un

único paciente y las colocase de forma ordenada y respetando los órdenes de precedencia, en aquellas posiciones que mejores resultados obtuviese de la FO.

BIBLIOGRAFÍA

[1] Ahsan, K. B., Alam, M. R., Morel, D. G., & Karim, M. A. (2019). Emergency department resource optimisation for improved performance: a review. Journal of Industrial Engineering International, 15(S1), 253-266. https://doi.org/10.1007/s40092-019-00335-x

[2] Allihaibi, W. G., Cholette, M. E., Masoud, M., Burke, J., & Karim, A. (2020). A heuristic approach for scheduling patient treatment in an emergency department based on bed blocking. International Journal of Industrial Engineering Computations, 565–584. https://doi.org/10.5267/j.ijiec.2020.4.005

[3] Alves de Queiroz, T., Iori, M., Kramer, A., & Kuo, Y.-H. (2021). Scheduling of patients in emergency departments with a variable neighborhood search. En Variable Neighborhood Search (pp. 138-151). Springer International Publishing.

[4] Alves de Queiroz, T., Iori, M., Kramer, A., & Kuo, Y.-H. (2023). Dynamic scheduling of patients in emergency departments. European Journal of Operational Research, 310(1), 100-116. https://doi.org/10.1016/j.ejor.2023.03.004

[5] Barroso, E. M. (2023). Desarrollo de algoritmos aproximados para la secuenciación de tareas del personal sanitario en un SUH. Universidad de Sevilla.

[6] Bedoya-Valencia, L., & Kirac, E. (2016). Evaluating alternative resource allocation in an emergency department using discrete event simulation. Simulation, 92(12), 1041-1051. https://doi.org/10.1177/0037549716673150

[7] Bernal-Delgado, E., Peiró, S., & Sotoca, R. (2006). Prioridades de investigación en servicios sanitarios en el Sistema Nacional de Salud. Una aproximación por consenso de expertos. Gaceta sanitaria, 20(4), 287-294. https://doi.org/10.1157/13091144

[8] B. Puebla Martínez and R. Vinader-Segura, Ecosistema de una pandemia, COVID 19 : transformación mundial. in Colección Conocimiento Contemporáneo ; 7. Madrid: Dykinson, 2021

[9] Bruballa, E., Wong, A., Rexachs, D., & Luque, E. (2020). An intelligent scheduling of non-critical patients admission for emergency department. IEEE access: practical innovations, open solutions, 8, 9209-9220. https://doi.org/10.1109/access.2019.2963049

[10] Cabrera, E., Taboada, M., Iglesias, M. L., Epelde, F., & Luque, E. (2012). Simulation optimization for healthcare emergency departments. Procedia Computer Science, 9, 1464–1473. https://doi.org/10.1016/j.procs.2012.04.161

[11] Chouba, I., Yalaoui, F., Amodeo, L., Arbaoui, T., Blua, P., Laplanche, D., & Sanchez, S. (2019). An efficient simulation-based optimization approach for improving emergency department performance. Studies in Health Technology and Informatics, 264, 1939–1940. https://doi.org/10.3233/SHTI190723

[12] De la Rosa, J. B. J. B. J. A. F. M. (n.d.). Ant Colony Optimization) para la resolución de problemas en líneas de producción. Departamento de Organització de Empresas , Universidad Politécnica de Cataluña

[13] E.-G. Talbi, Metaheuristics: From design to implementation. Wiley, 2009.

[14] ESTUDIO DEL TRIAGE Y TIEMPOS DE ESPERA EN UN SERVICIO DE URGENCIAS HOSPITALARIO. (s. f.). 1Library.co. Recuperado 28 de mayo de 2024, de https://1library.co/document/zlg3xjmo-estudio-triage-tiempos-espera-servicio-urgencias-hospitalario.html

[15] Ghazi W.A., Masoud M., Elgenawy M. (2021) Solving the Emergency Care Patient Pathway by a New Integrated Simulation Optimisation Approach

[16] Feili, H. R. (2013). Improving the health care systems performance by simulation optimization. Journal of Mathematics and Computer Science, 07(01), 73-79. https://doi.org/10.22436/jmcs.07.01.08

[17] Gestión clínica de un servicio de urgencias hospitalario mediante un cuadro de mando asistencial específico. (s. f.). Revistaemergencias.org. Recuperado 28 de mayo de 2024, de https://revistaemergencias.org/numeros-anteriores/volumen-24/numero-6/gestion-clinica-de-un-servicio-de-urgencias-hospitalario-mediante-un-cuadro-de-mando-asistencial-especifico/

[18] Gómez Jimenez. Sistema Español de Triaje: grado de implantación y posibilidades de desarrollo futuras: https://revistaemergencias.org/wp-content/uploads/2023/08/Emergencias-2011_23_5_344-345.pdf

[19] Hamzaoui, I., Bouzir, A., & Benammou, S. (2021). Towards A fuzzy rule-based approach for patient flow management in emergency department during the COVID-19: Sahloul hospital case study, Tunisia. 2021 International Conference on Engineering and Emerging Technologies (ICEET).

[20] Harzi, M., Condotta, J.-F., Nouaouri, I., & Krichen, S. (2017). Scheduling patients in emergency department by considering material resources. Procedia Computer Science, 112, 713-722. https://doi.org/10.1016/j.procs.2017.08.153

[21] Harzi, M., Condotta, J.-F., Nouaouri, I., & Krichen, S. (2018). Using the hybrid ILS/VND method for solving the patients scheduling problem in emergency department: a case study. Procedia Computer Science, 126, 733-742. https://doi.org/10.1016/j.procs.2018.08.007

[22] Herrera, F. (s. f.). Introducción a los Algoritmos Metaheurísticos. Ugr.es. Recuperado 28 de mayo de 2024, de https://sci2s.ugr.es/sites/default/files/files/Teaching/OtherPostGraduateCourses/Metaheuristicas/Int-Metaheuristicas-CAEPIA-2009.pdf

[23] Helena Ramalhinho Dias Lourenço, D. S. de la F. (n.d.). Métodos de solución de problemas de asignación de recursos sanitarios. Fbbva.Es. Retrieved June 25, 2024, from https://www.fbbva.es/wp-content/uploads/2017/05/dat/DT_2004_01.pdf

[24] Hu, H., He, J., He, X., Yang, W., Nie, J., & Ran, B. (2019). Emergency material scheduling optimization model and algorithms: A review. Journal of Traffic and Transportation Engineering (English Edition), 6(5), 441-454. https://doi.org/10.1016/j.jtte.2019.07.001

[25] J. Derrac, S. García, S. Hui, P. N. Suganthan, and F. Herrera, "Analyzing convergence performance of evolutionary algorithms: A statistical approach," Inf. Sci. (Ny), vol. 289, no. C, pp. 41–58, 2014.

[26] Sánchez, S. L., & González, M. L. (2014). Optimización de recursos y calidad de servicio en las consultas de urgencias de un centro de atención primaria. Pensamiento matemático, 4(2), 105–123. https://dialnet.unirioja.es/servlet/articulo?codigo=5995038

[27] Kırış, Ş., Yüzügüllü, N., Ergün, N., & Alper Çevik, A. (2010). A knowledge-based scheduling system for Emergency Departments. Knowledge-Based Systems, 23(8), 890-900. https://doi.org/10.1016/j.knosys.2010.06.005

[28] Lee, S., & Lee, Y. H. (2020). Improving emergency department efficiency by patient scheduling using deep reinforcement learning. Healthcare (Basel, Switzerland), 8(2), 77. https://doi.org/10.3390/healthcare8020077

[29] Lo, C.-C., & Lin, T.-H. (2011). A Particle Swarm Optimization approach for physician scheduling in a hospital emergency department. 2011 Seventh International Conference on Natural Computation.

[30] Medina, G. (s. f.). Algoritmos de Optimización para el servicio de urgencias: Caso de estudio en el Hospital Universitario Virgen del Rocío.

[31] Moyano, J. P. (2023). Desarrollo de algoritmos aproximados para la resolución del problema de gestión del flujo de pacientes en un servicio de urgencia hospitalario. Universidad de Sevilla.

[32] Soler, W., Gómez Muñoz, M., Bragulat, E., & Alvarez, A. (2010). Triage: a key tool in emergency care. An. del sistema sanitario de Navarra, 33 Suppl 1, 55-68. https://scielo.isciii.es/pdf/asisna/v33s1/original8.pdf

[33] Terning, G., Brun, E. C., & El-Thalji, I. (2022). Modeling patient flow in an emergency department under COVID-19 pandemic conditions: A hybrid modeling approach. Healthcare (Basel, Switzerland), 10(5), 840. https://doi.org/10.3390/healthcare10050840

[34] Tzeng, Y.-R., Chen, C.-L., & Chen, C.-L. (2012). A hybrid EDA with ACS for solving permutation flow shop scheduling. The International Journal of Advanced Manufacturing Technology, 60(9-12), 1139-1147. https://doi.org/10.1007/s00170-011-3671-1

[35] Valenzuela-Núñez, G. (2024). Smart Medical Appointment Scheduling: Optimization, Machine Learning, and Overbooking to Enhance Resource Utilization CATALINA.

[36] Vasquez, M. E. M. (2018). Secuenciación de operaciones de manufactura en la linea de producción automatizada mediante la aplicación de Algoritmo de Colonia de Hormigas. Universidad Pontificia Bolivariana

[37] Yagmahan, B., & Yenisey, M. M. (2008). Ant colony optimization for multi-objective flow shop scheduling problem. Computers & Industrial Engineering, 54(3), 411-420. https://doi.org/10.1016/j.cie.2007.08.003

[38] Zhang, H., Best, T. J., Chivu, A., & Meltzer, D. O. (2020). Simulation-based optimization to improve hospital patient assignment to physicians and clinical units. Health Care Management Science, 23(1), 117–141. https://doi.org/10.1007/s10729-019-09483-3

[39] Flórez, E., Gómez, W., & Bautista, L. (n.d.). An ant colony optimization algorithm for job shop scheduling problem. Arxiv.org. Retrieved June 30, 2024, from https://arxiv.org/pdf/1309.5110

[40] Sistema Estructurado de Triaje - Wikipedia, la enciclopedia libre

ANEXO

A lo largo del presente anexo, se mostrará en orden todas las funciones y celdas necesarias para llevar a cabo el proyecto. El código se elaboró en un cuaderno de Google Colab donde permitía separar cada celda y las diferentes secciones, ofreciendo una mayor facilidad al programador para ejecutar aquello que precisa en cada momento, y navegar por las distintas funciones.

Para una mayor claridad, se han separado las secciones de la siguiente forma:

- Para generar un nuevo problema y obtener una primera solución de secuenciación de actividades se deben ejecutar los apartados A.a y A.b en ese orden.
- Posteriormente, se puede optimizar la solución por diferentes vías:
  - ACO1 : ejecutamos todo el punto B
  - ACO2 o ACO3: ejecutamos todo el punto C
  - ACO4: ejecutamos todo el punto D
  - ACO1 + BL : ejecutamos B.a, seguido de E.a y por último B.c
  - ACO2 + BL o ACO3 + BL: ejecutamos C.a, seguido de E.b y por último C.c
  - ACO4 + BL : ejecutamos D.a, seguido de E.c y por último D.c

A) Funciones 'decoding'

```
import pandas as pd
import plotly.express as px
import pandas as pd
import numpy as np
from datetime import datetime, timedelta
import random
import copy
from operator import itemgetter
import math

def establece_prioridad(numero_pacientes, max_prioridad):
  ESI = range(1,max_prioridad+1)
  probabilidades = [0.76, 11.26, 48.86, 37.16, 1.96]

  # Normalizar las probabilidades
  probabilidades_normalizadas = [p / sum(probabilidades) for p in probabilidades]

  array_prioridades = np.random.choice(ESI, numero_pacientes,
p=probabilidades_normalizadas)
  return array_prioridades

def define_tepcof_paciente(prioridad, TEPCOF):
    valor=0
```

```
    if prioridad==1:
        valor=TEPCOF['1']

    elif prioridad==2:

        valor=TEPCOF['2']

    elif prioridad==3:
        valor=TEPCOF['3']

    elif prioridad==4:
        valor=TEPCOF['4']

    elif prioridad==5:
        valor=TEPCOF['5']

    return valor
def generar_pacientes_y_recursos(array_prioridades, procesos_por_prioridad, TEPCOF,
tipos_recursos, recursos):
    pacientes = []
    recursos_ocupados = []
    PU_i = []
    matriz = []
    disponible = copy.copy(tipos_recursos)
    recursos_disponibles = copy.copy(recursos)
    pacientes_tipo_4 = 0
    act=0
    orden_actividades_por_paciente = {}
    df_recursos_ocupados = pd.DataFrame(columns=['recurso', 't_disponible', 'TR'])
    # Recorre paciente a paciente
    for paciente_id , prioridad in enumerate(array_prioridades,1):

      # Asigna PU segun prioridad
      PU_i = (procesos_por_prioridad[prioridad])

      TEPCOF_i = define_tepcof_paciente(prioridad, TEPCOF)

      tipo_paciente = random.choice([1, 2, 3, 4])

    ## Elige tipo_paciente

      if tipo_paciente == 1:
          paciente = {
              "Paciente": paciente_id,
              "TEPCOF": TEPCOF_i,
              "Prioridad": prioridad,
              "t_llegada": 0,
              "visto": 0,
              "t_disponible": 0,
```

```
            "tipo_paciente":1,
            "PU": [sublista[0] for sublista in PU_i]
          }
    elif tipo_paciente == 2:
        t_llegada_max = TEPCOF_i * (1+(prioridad/5))
        t_llegada = math.ceil(random.uniform(1, t_llegada_max))

        paciente = {
            "Paciente": paciente_id,
            "TEPCOF": TEPCOF_i,
            "Prioridad": prioridad,
            "t_llegada": t_llegada,
            "visto": 0,
            "t_disponible": 0,
            "tipo_paciente":2,
            "PU": [sublista[0] for sublista in PU_i]
          }

    elif tipo_paciente == 3:

        PU_last = PU_i[:][:-1]
        punto_parada = random.choice(PU_last)
        indice_sublista = PU_i.index(punto_parada)

        # Crear la nueva lista a partir del índice encontrado
        nuevo_PU_i = PU_i[indice_sublista + 1:]

        PU_i = nuevo_PU_i

        paciente = {
            "Paciente": paciente_id,
            "TEPCOF": None,
            "Prioridad": prioridad,
            "t_llegada": None,
            "visto": 1,
            "t_disponible": 0,
            "tipo_paciente": 3,
            "PU": [sublista[0] for sublista in PU_i]
          }

    else:

          PU_last = PU_i[:][:-1]
          flag = 0

          # Elegimos una actividad que esté dentro del PU de ese paciente
          ## Vamos recorriendo actividad por actividad para ver si en alguna hay
recursos disponibles
          for i in range(0, len(PU_last)):
```

```
                recurso_elegido = PU_last[i]
                if int(recurso_elegido[1]) <= int(disponible[recurso_elegido[0]]): # Si
hay disponibles, deja de buscar
                    flag = 1
                    break

            if flag == 1: # Paciente tipo 4
              pacientes_tipo_4 += 1
              # Encontrar el índice de la sublista en la lista original
              indice_sublista = PU_i.index(recurso_elegido)

              # Crear la nueva lista a partir del índice encontrado
              nuevo_PU_i = PU_i[indice_sublista + 1:]
              PU_i = nuevo_PU_i

              t_disponible = math.ceil(random.uniform(1, recurso_elegido[3]))
              paciente = {
                "Paciente": paciente_id,
                "TEPCOF": None,
                "Prioridad": prioridad,
                "t_llegada": None,
                "visto": 1,
                "t_disponible": t_disponible,
                "tipo_paciente": 4,
                "PU": list(map(itemgetter(0), PU_i))
              }

              if not recurso_elegido[0].startswith('LAB'): # Si es LAB no hace falta
bloquear el recurso
                  disponible[recurso_elegido[0]] -= 1
                  # Especificamos recurso que esta ocupando
                  for i in recursos_disponibles:
                      if i.startswith(recurso_elegido[0]):
                          recurso_ocupado = {
                            "recurso": i,
                            "t_disponible": t_disponible,
                            "TR": recurso_elegido[0]
                          }
                          recursos_disponibles.remove(i)
                          recursos_ocupados.append(recurso_ocupado)
                          break

            else:
                  t_llegada_max = TEPCOF_i * (1+(prioridad/5))
                  t_llegada = random.uniform(1, t_llegada_max)

                  paciente = {
                      "Paciente": paciente_id,
                      "TEPCOF": TEPCOF_i,
```

```
                    "Prioridad": prioridad,
                    "t_llegada": t_llegada,
                    "visto": 0,
                    "t_disponible": 0,
                    "tipo_paciente":2,
                    "PU": list(map(itemgetter(0), PU_i))
                  }

      pacientes.append(paciente)

      # Establecemos tiempos actividades
      array_actividades = []
      for actividad in PU_i:
          act+=1
          tiempo = math.ceil(random.triangular(int(actividad[2]), int(actividad[4]),
int(actividad[3])))
          matriz.append({
                  'Actividad': act,
                  'Paciente': paciente_id,
                  'Prioridad': prioridad,
                  'TR': actividad[0],
                  'Recursos_Necesarios': actividad[1],
                  'Tiempo': tiempo
              })
          array_actividades.append(act)
      orden_actividades_por_paciente[paciente_id] = array_actividades # esto solo lo
queremos para ACO_Actividades

    # Crear DataFrame de pacientes
    df_pacientes = pd.DataFrame(pacientes)
    # 1. Ordenar el DataFrame por la columna 'Visto', de tal manera que salgan los ya
atendidos primero
    df_pacientes = df_pacientes.sort_values(by='visto', ascending=False)
    # 2.Ordenamos por prioridad
    df_pacientes = df_pacientes.sort_values(by='Prioridad', ascending=True)

    # Crear DataFrame de recursos ocupados
    if len(recursos_ocupados) != 0 :
      df_recursos_ocupados = pd.DataFrame(recursos_ocupados)

    # Crear DataFrame de todos los recursos con t_disponible = 0 para aquellos que no
estén en df_recursos_ocupados
    df_recursos_totales = pd.DataFrame([{"recurso": recurso, "t_disponible": 0} for
recurso in recursos])

    #Combinar DataFrames para obtener el DataFrame final de recursos
    if len(df_recursos_ocupados) > 0:
      df_ocupados = df_recursos_ocupados.drop('TR', axis=1)
      df_recursos_final = pd.concat([df_recursos_totales, df_ocupados],
```

```
ignore_index=True).groupby("recurso").max().reset_index()

    else:
      df_recursos_final = df_recursos_totales

    # DF datos
    df_datos = pd.DataFrame(matriz)


    return df_pacientes, df_recursos_final, recursos, df_datos, df_recursos_ocupados,
orden_actividades_por_paciente

def asigna_recurso(tipos_recurso, recursos_disponibles, capacidad_recursos,
recurso_actividad, actividad ,tiempo_paciente, tiempo_actividad, paciente,
cantidad_recurso_actividad, tiempos_finalizacion_recurso, tiempos_finalizacion_paciente,
consultas_asignadas,tiempos_finalizacion_actividad,tiempo_inicio_actividad,
recurso_asignado):
  asignado=0
  recursos = copy.copy(recursos_disponibles)
  recursos_tipo_disponibles = {}
  recursos_tipo_no_disponibles = {}
  recursos_asignados = []
  asigna_consulta = 0

  if recurso_actividad in ['CON','ACON']:
    # Si aun no se le ha asignado una consulta a ese paciente
    if consultas_asignadas[paciente] == 0:
     for recurso in recursos:
        if recurso.startswith(recurso_actividad):
          tiempo = tiempos_finalizacion_recurso.get(recurso,0)
          recursos_tipo_disponibles[recurso] = tiempo

     asigna_consulta = 1

    else:
      recurso = consultas_asignadas[paciente]
      tiempo = tiempos_finalizacion_recurso.get(recurso,0)
      recursos_tipo_disponibles[recurso] = tiempo

  else:
    for recurso in recursos:

      if recurso.startswith(recurso_actividad):

          # Mirar si se trata de un recurso cuya capacidad está limitada o no
          if capacidad_recursos[recurso_actividad] > 1:
            recursos_tipo_disponibles[recurso] = 0 # siempre menor que t_paciente
            asignado +=1
            break
```

```
          else:
            tiempo = tiempos_finalizacion_recurso.get(recurso,0)
            recursos_tipo_disponibles[recurso] = tiempo


  tiempo_inicio = 0
  for i in range(cantidad_recurso_actividad):
     # Ver cuando se libera ANTES alguno de los recursos
      clave_minima = min(recursos_tipo_disponibles, key=lambda k:
recursos_tipo_disponibles[k])

      # Guardamos el tiempo del recurso para compararlo con proximos recursos si fuese
necesario
      tiempo_inicio_ant = tiempo_inicio
      # La actividad empezará cuando paciente y recurso estén disponibles (tiempo mayor
entre los dos)
      tiempo_inicio = max(tiempos_finalizacion_recurso.get(clave_minima,0),
tiempo_paciente, tiempo_inicio_ant)
      # Lo guardamos en el array de recursos que sí vamos a asignar
      recursos_asignados.append(clave_minima)
      # Lo eliminamos para que ya no sea el valor mas pequeño del diccionario
      del recursos_tipo_disponibles[clave_minima]

  tiempo_inicio_actividad[actividad] = tiempo_inicio
  tiempos_finalizacion_actividad[actividad] = tiempo_inicio + tiempo_actividad

  # Guardamos el tiempo en el que los recursos y paciente se liberarán tras esta
actividad
  for clave in recursos_asignados:
    tiempos_finalizacion_recurso[clave]= (tiempo_inicio + tiempo_actividad)

    if asigna_consulta == 1: # Que ese paciente aun no tiene consulta asignada
      consultas_asignadas[paciente] = clave

  tiempos_finalizacion_paciente[paciente] = tiempo_inicio + tiempo_actividad

  # Guardamos el recurso que se ha asignado a la actividad para el GANTT
  recurso_asignado[actividad] = [clave for clave in recursos_asignados]


def calcula_saturacion(df_datos, df_recursos_ocupados, nivel,tipos_recurso):
  ocupacion_recursos = {}
  saturacion = {}
  LOSmedio = ((3.15+3.14+2.86+1.49+1.31)*60)/5
  flag = 0
  # Iterar sobre las filas del DataFrame y calcular la ocupación acumulada de cada
recurso
  for index, row in df_datos.iterrows():
      recurso = row['TR']
      tiempo_actividad = row['Tiempo']
```

```
      # Si el recurso ya está en el diccionario, sumar el tiempo de esta actividad
      if recurso in ocupacion_recursos:
          ocupacion_recursos[recurso] += tiempo_actividad
      # Si el recurso no está en el diccionario, agregarlo con el tiempo de esta
actividad
      else:
          ocupacion_recursos[recurso] = tiempo_actividad

  for index, row in df_recursos_ocupados.iterrows():
    recurso = row['TR']
    t_ocupado = row['t_disponible']
    if recurso in ocupacion_recursos:
        ocupacion_recursos[recurso] += t_ocupado
    else:
        ocupacion_recursos[recurso] = t_ocupado

  # Mostrar la ocupación acumulada de cada recurso
  for recurso, ocupacion in ocupacion_recursos.items():
      n = tipos_recurso[recurso]
      sat = LOSmedio * n
      saturacion[recurso] = (ocupacion/sat)*100
      print(f"Recurso {recurso}: Ocupación acumulada = {ocupacion} minutos, saturación =
{saturacion[recurso]}")
      if saturacion[recurso] > nivel:
          flag = 1

  return flag


def FO_solucion(df_pacientes, tiempo_final_paciente):
  max_value = 0
  tepcof_FO =0
  clave_maxima, maximo_valor = max(tiempo_final_paciente.items(), key=lambda x: x[1])
  for index, row in df_pacientes.iterrows():
    t_llegada = row['t_llegada']
    tepcof = row['TEPCOF_real']
    epcof = row['TEPCOF']
    prioridad = row['Prioridad']
    if t_llegada != None:
      if t_llegada >= max_value :
        max_value = t_llegada
      if tepcof > epcof:
        tepcof_FO += tepcof * (6-prioridad)

  FO = maximo_valor + max_value + tepcof_FO
  return FO
```

```
def calcula_TEPCOF(df_pacientes, df_datos_unido):
  # Itera sobre las filas del DataFrame
  for index1, row1 in df_pacientes.iterrows():
        tiempo_primera_actividad = 1000
        # Se calcula solo en aquellos pacientes que no han sido vistos
        if row1['visto'] == 0:
          a = row1['Paciente']
          b = row1['t_llegada']
          # Guardamos el tiempo de su primera actividad (el tiempo de inicio mas pequeño)
          for index2, row2 in df_datos_unido.iterrows():
            if row2['Paciente'] == a:
              t_inicio = df_datos_unido.loc[index2,'Tiempo_inicio']
              if t_inicio < tiempo_primera_actividad:
                tiempo_primera_actividad = t_inicio

          calculo = tiempo_primera_actividad + b
        else:
          calculo = None

        # Añadimos el dato en una nueva columna, en el indice correspondiente del primer
DataFrame
        df_pacientes.at[index1, 'TEPCOF_real'] = calculo

  return df_pacientes


def ejecuta_decoding(rep_solucion, array_prioridades, df_pacientes, df_recursos_final,
df_datos, array_pacientes):
  # Diccionarios (clave: valor) para mayor legibilidad de los resultados
  tiempo_inicio_actividad={}
  tiempos_finalizacion_actividad={}
  consultas_asignadas = {}
  recurso_asignado={}

  # Inicializamos las consultas asignadas a cada paciente
  for i in array_pacientes:
    consultas_asignadas[i] = 0 # paciente 1: CON1, paciente 2: CON2 ...

    # Seleccionar las columnas de interés
  columna_clave = 'Paciente'
  columna_valor = 't_disponible'

    # Crear el diccionario
  tiempo_final_paciente = df_pacientes.set_index(columna_clave)[columna_valor].to_dict()

    # Seleccionar las columnas de interés
  columna_clave2 = 'recurso'
  columna_valor2 = 't_disponible'
```

```
    # Crear el diccionario
  tiempo_final_recurso =
df_recursos_final.set_index(columna_clave2)[columna_valor2].to_dict()

  ## Bucle que organiza actividades en orden de la solucion
  for actividad in rep_solucion:
        # ¿Qué tipo de recurso necesita la actividad?
        indice = df_datos.loc[df_datos['Actividad'] == actividad].index.item()
        paciente, tipo, cantidad_recurso_actividad, tiempo_actividad =
df_datos.loc[indice,['Paciente','TR','Recursos_Necesarios','Tiempo']]
        tiempo_p = tiempo_final_paciente.get(paciente,0)

        asigna_recurso(TR, R, CR, tipo, actividad ,tiempo_p, tiempo_actividad, paciente,
cantidad_recurso_actividad, tiempo_final_recurso, tiempo_final_paciente,
consultas_asignadas,tiempos_finalizacion_actividad,tiempo_inicio_actividad,
recurso_asignado)

  data_list = []
  for key, value in tiempos_finalizacion_actividad.items():
          data_list.append({'Actividad': key, 'Recurso_asignado':
recurso_asignado[key],'Tiempo_inicio':tiempo_inicio_actividad[key], 'Tiempo_FIN': value})
  Data_pacientes = pd.DataFrame(data_list)
  df_datos_unido = pd.merge(df_datos, Data_pacientes, on='Actividad', how='inner')

  ######################################################################
  ### Calcula TEPCOF:
  df_pacientes = calcula_TEPCOF(df_pacientes, df_datos_unido)

  #### Calcula FO
  FO=FO_solucion(df_pacientes, tiempo_final_paciente)

  return FO, df_datos_unido, df_pacientes

def representa_GANTT(df_datos_unido):
  df_expanded = df_datos_unido.explode('Recurso_asignado')
  #Obtener la fecha actual
  fecha_actual = datetime.now()

  # Convertir los tiempos de inicio y fin en formato datetime para Plotly
  df_expanded['Inicio'] =  fecha_actual + pd.to_timedelta(df_expanded['Tiempo_inicio'],
unit='m')
  df_expanded['Fin'] =  fecha_actual + pd.to_timedelta(df_expanded['Tiempo_FIN'],
unit='m')


  # Generar colores aleatorios para los pacientes
  unique_patients = df_expanded['Paciente'].unique()
  colors = {patient: f'#{random.randint(0, 0xFFFFFF):06x}' for patient in
unique_patients}
```

```
  # Crear una nueva columna 'Leyenda' para el color de la leyenda
  df_expanded['Leyenda'] = df_expanded['Paciente'].apply(lambda x: f'Paciente {x}')


  # Generar el diagrama de Gantt
  fig = px.timeline(df_expanded, x_start='Inicio', x_end='Fin', y='Recurso_asignado',
text='Actividad', color='Leyenda', color_discrete_map={f'Paciente {k}': v for k, v in
colors.items()},
                    title='Diagrama de Gantt')

  # Mostrar el diagrama
  fig.show()
  return
```

a. Celda para la simulación de escenarios

```
I = 0 # Inicializamos pacientes
flag = 0

#franja = 0 # a partir de las 12 am
franja = 1 # a partir de las 6am
if franja==0:
    procesos_por_prioridad = {
            1: [['ENF', 2, 20, 30, 75],
                ['CON',1, 10, 20, 25],
                ['LAB', 1, 20,35,45],
                ['RAY',1, 3, 5, 15],
                ['ENF', 2,  20,30, 40],
                ['CON', 1, 20,30, 40]],
            2: [['ENF', 2, 18, 28, 45],
                ['CON',1, 10, 20, 25],
                ['LAB', 1, 3, 5, 15],
                ['RAY',1, 5, 10, 20],
                ['ENF', 2, 15, 20, 30],
                ['CON', 1, 15, 20, 25 ]],
            3: [['ENF', 1, 15, 22, 30],
                ['CON',1, 5, 15, 20],
                ['LAB', 1, 3, 5, 15],
                ['RAY',1, 5, 10, 20],
                ['ENF', 1, 10, 15, 20],
                ['CON', 1, 8, 15, 18]],
```

```
            4: [['ENF', 1, 5, 10, 20],
                ['ACON',1, 2, 3, 5],
                ['RAY',1, 2, 5, 10],
                ['ENF', 1, 5, 8, 12 ],
                ['ACON', 1, 5, 10, 15]],
            5: [['ENF', 1, 2, 4, 8],
                ['ACON',1, 2, 3, 5]]
      }
    TR={'CON':2,'ENF':9,'RAY':1,'LAB':1 ,'ACON':2}
    R=['CON1','CON2','ENF1','ENF2','ENF3','ENF4','ENF5','ENF6','ENF7','ENF8','ENF9','RAY'
,'LAB','ACON1','ACON2']

else:
    procesos_por_prioridad = {
          1: [['ENF', 2, 20, 30, 75],
              ['CON',1, 10, 20, 25],
              ['LAB', 1, 20,35,45],
              ['RAY',1, 3, 5, 15],
              ['ENF', 2,  20,30, 40],
              ['CON', 1, 20,30, 40]],
          2: [['ENF', 2, 18, 28, 45],
              ['CON',1, 10, 20, 25],
              ['LAB', 1, 3, 5, 15],
              ['RAY',1, 5, 10, 20],
              ['ENF', 2, 15, 20, 30],
              ['CON', 1, 15, 20, 25 ]],
          3: [['ENF', 1, 15, 22, 30],
              ['CON',1, 5, 15, 20],
              ['LAB', 1, 3, 5, 15],
              ['RAY',1, 5, 10, 20],
              ['ENF', 1, 10, 15, 20],
              ['CON', 1, 8, 15, 18]],
          4: [['ENF', 1, 5, 10, 20],
              ['CON',1, 2, 3, 5],
              ['RAY',1, 2, 5, 10],
              ['ENF', 1, 5, 8, 12 ],
              ['CON', 1, 5, 10, 15]],
          5: [['ENF', 1, 2, 4, 8],
              ['CON',1, 2, 3, 5]]
    }
    TR={'CON':1,'ENF':6,'RAY':1,'LAB':1}
    R=['CON1','ENF1','ENF2','ENF3','ENF4','ENF5','ENF6','RAY','LAB']

TEPCOF = {'1':0, '2': 15, '3':60, '4':120, '5': 240}
#Capacidad de los recursos
CR ={'CON':1,'ENF':1,'RAY':1,'LAB':float('inf') ,'ACON':1}
nivel_sat = 100 #50,75,100%
######################################################################
```

```
while flag ==0:
    I += 1
    array_prioridades = establece_prioridad(I, 5)
    df_pacientes, df_recursos_final, recursos, df_datos, df_recursos_ocupados,
orden_actividades_por_paciente = generar_pacientes_y_recursos(array_prioridades,
procesos_por_prioridad, TEPCOF, TR, R )
    flag = calcula_saturacion(df_datos, df_recursos_ocupados, nivel_sat,TR)
```

b. Aplicación heurística constructiva y mostrar GANTT

```
# Construcción de una solución en función de la prioridad y estado

array_pacientes = df_pacientes['Paciente'].values
# Agrupar actividades por paciente
matriz_actividades =
df_datos.groupby('Paciente')['Actividad'].apply(list).reindex(array_pacientes).tolist()

# Obtener la longitud máxima de las sublistas en Pu_ordenado
longitud_maxima = max(map(len, matriz_actividades))
# Rellenar con ceros las sublistas que no llegan al máximo para poder transponerla sin
perder actividades
Pu_transpuesto = [sublista + ['0'] * (longitud_maxima - len(sublista)) for sublista in
matriz_actividades]

# Transponer la lista resultante
Pu_transpuesto = list(map(list, zip(*Pu_transpuesto)))

# Obtenemos la matriz de Pu sin los ceros
rep_solucion_1 = []
for i in Pu_transpuesto:
  for j in i:
    if j!='0':
      rep_solucion_1.append(j)

print(rep_solucion_1)

######################################################################

FO_inicial_1, df_datos_unido, df_pacientes = ejecuta_decoding(rep_solucion_1,
array_prioridades, df_pacientes, df_recursos_final, df_datos, array_pacientes)

representa_GANTT(df_datos_unido)
```

B) Algoritmo de Colonia de Hormigas por actividades y con nodo de INICIO (ACO1)

a. Funciones principales:

```
import numpy as np
import time

# Utilizar la solución inicial para ajustar las feromonas
def ajustar_feromonas_iniciales(feromonas, solucion_inicial, calidad_inicial):
    feromonas[0][solucion_inicial[0]] += calidad_inicial
    for i in range(len(solucion_inicial) - 1):
        feromonas[solucion_inicial[i]][solucion_inicial[i+1]] += calidad_inicial  #
Ajustar según la calidad de tu solución inicial
    return feromonas

# Función para generar la matriz de costos, teniendo en cuenta las restricciones
# actividades : Actividad | Paciente  | Prioridad | TR  | Recursos_Necesarios | Tiempo
def generar_matriz_costos(actividades, orden_actividades_por_paciente):
    num_actividades = len(actividades)
    matriz_costos = np.full((num_actividades + 1, num_actividades + 1), np.inf)  #
Incluye nodo de inicio

    # Identificar las primeras actividades de cada paciente
    primeras_actividades = [actividades[0] for actividades in
orden_actividades_por_paciente.values()]

    # Asignar costos desde el nodo de inicio a las primeras actividades
    for actividad in primeras_actividades:
        tiempo = actividades[actividad - 1][5]  # Tiempo de la actividad
        prioridad = actividades[actividad - 1][2]  # Prioridad del paciente de la
actividad
        matriz_costos[0][actividad] = calcular_costo(tiempo, prioridad)

    # Asignar costos entre actividades normales
    for i in range(1, num_actividades + 1):
        for j in range(1, num_actividades + 1):
            if i != j:  # Asegurarse de que no se esté calculando el costo de una
actividad a sí misma
                if es_transicion_valida(actividades[i - 1][:2], actividades[j - 1][:2],
orden_actividades_por_paciente):
                    tiempo = actividades[j - 1][5]  # Tiempo de la actividad destino
                    prioridad = actividades[j - 1][2]  # Prioridad del paciente de la
actividad destino
                    matriz_costos[i][j] = calcular_costo(tiempo, prioridad)

    return matriz_costos

def es_transicion_valida(actividad_origen, actividad_destino,
orden_actividades_por_paciente):
    paciente_origen = actividad_origen[1]
    paciente_destino = actividad_destino[1]
    actividad_origen_id = actividad_origen[0]
    actividad_destino_id = actividad_destino[0]
```

```
    # Validación intra-paciente
    if paciente_origen == paciente_destino:
        indice_origen =
orden_actividades_por_paciente[paciente_origen].index(actividad_origen_id)
        indice_destino =
orden_actividades_por_paciente[paciente_destino].index(actividad_destino_id)
        return indice_destino == indice_origen + 1 # Devuelve true solo si es justo la
actividad que precede a la de origen

    # Validación inter-paciente
    else:
      return True

# Calcular coste de ir a una actividad
def calcular_costo(tiempo, prioridad):
    base_cost = 10  # Costo base para cualquier transición
    costo = base_cost + tiempo - (6-prioridad)
    return costo

def seleccionar_proxima_actividad(alpha, beta, q0, actividades_por_añadir, actividades,
feromonas, matriz_costos, orden_actividades_por_paciente,
actividades_seleccionadas_por_hormiga):

    if not actividades_por_añadir:
        return None, actividades_por_añadir

    actividad_actual = actividades_seleccionadas_por_hormiga[-1]
    transiciones_validas = []

    for actividad_destino in actividades_por_añadir:
        # Verificar que todas las actividades previas del paciente asociado a esta
actividad se hayan completado
        for a in actividades:
          if actividad_destino == a[0]:
            paciente_destino = a[1]
            break

        actividades_previas =
orden_actividades_por_paciente[paciente_destino][:orden_actividades_por_paciente[paciente
_destino].index(actividad_destino)]

        if all(actividad_previa in actividades_seleccionadas_por_hormiga for
actividad_previa in actividades_previas):
          transiciones_validas.append(actividad_destino)

    if not transiciones_validas:  # Si no hay transiciones válidas
        return None, actividades_por_añadir
```

```
    # Decision de explotación vs exploración
    if random.random() < q0:
        # Explotación: elegir el mejor basado en feromonas y costo (heurística)
        mejor_costo = 0
        seleccionado = None

        for actividad_destino in transiciones_validas:
            tau = feromonas[int(actividad_actual)][int(actividad_destino)]  # Cantidad de
feromona
            costo= matriz_costos[int(actividad_actual)][int(actividad_destino)] # Valor
costo
            eta = 1 / costo if costo != float('inf') else 0
            valor = (tau ** alpha) * (eta ** beta)
            if valor > mejor_costo:
                mejor_costo = valor
                seleccionado = actividad_destino

        actividades_por_añadir.remove(seleccionado)
        return seleccionado, actividades_por_añadir
    else:
        # Exploración: elegir basado en una distribución de probabilidades proporcional a
feromonas y heurística
        probabilidades = []
        for actividad_destino in transiciones_validas:
            tau = feromonas[int(actividad_actual)][int(actividad_destino)]  # Cantidad de
feromona
            costo= matriz_costos[int(actividad_actual)][int(actividad_destino)] # Valor
costo
            eta = 1 / costo if costo != float('inf') else 0

            probabilidades.append((tau ** alpha) * (eta ** beta))

        # Normalizar las probabilidades
        suma_probabilidades = sum(probabilidades)
        if suma_probabilidades == 0:
            return None, actividades_por_añadir  # En caso de que todas las
probabilidades sean 0

        probabilidades_normalizadas = [prob / suma_probabilidades for prob in
probabilidades]
        seleccionado = np.random.choice(transiciones_validas,
p=probabilidades_normalizadas)

        actividades_por_añadir.remove(seleccionado)

        return seleccionado, actividades_por_añadir

def generar_solucion_hormiga(best_sol_i, feromonas,
matriz_costos,orden_actividades_por_paciente, actividades,alpha, beta, q0):
```

```
  actividades_seleccionadas_por_hormiga = [0]
  actividades_por_añadir = copy.copy(best_sol_i)

  # Mientras que la solución no esté completa
  while len(actividades_seleccionadas_por_hormiga) < len(best_sol_i)+1:
            prox_actividad, actividades_por_añadir = seleccionar_proxima_actividad(alpha,
beta, q0,actividades_por_añadir, actividades, feromonas, matriz_costos,
orden_actividades_por_paciente, actividades_seleccionadas_por_hormiga)
            actividades_seleccionadas_por_hormiga.append(prox_actividad)

  return actividades_seleccionadas_por_hormiga[1:]  # Omitir el nodo de inicio en el
resultado final

# Actualización de feromonas
def actualizar_feromonas(matriz_feromonas, solucion_hormiga, calidad_solucion,
evaporacion):
    # Evaporación de feromonas
    matriz_feromonas *= (1 - evaporacion)

    # Aumentamos feromona en el arco de inicio
    matriz_feromonas[0][solucion_hormiga[0]] += calidad_solucion

    for i in range(len(solucion_hormiga)-1):
            # Aumentar las feromonas basado en la calidad de la solución
            matriz_feromonas[solucion_hormiga[i]][solucion_hormiga[i+1]] +=
calidad_solucion

    return matriz_feromonas

def calcular_calidad_solucion(FO_decoding):
    if FO_decoding > 0:
        return 100 / FO_decoding
    else:
        return float('inf')  # En caso de que tiempo_total sea 0, lo cual es improbable
en escenarios reales.
```

b. Función algoritmo principal:

```
# Algoritmo principal
def algoritmo_aco(tiempo_computacion,num_hormigas, parametros ,Q_0, FO_inicial,
solucion_inicial, actividades, orden_actividades_por_paciente, df_pacientes,
df_recursos_final, df_datos, array_pacientes):
    # Inicializar parametros
    num_actividades = len(actividades)  # Ajustar según tu caso

    alpha = parametros[0] # Influencia de la feromona
    beta = parametros [1] # Influencia de la información heurística
    evaporacion = parametros[2] # Tasa de evaporación de la feromona
```

```
    q0 = Q_0

    feromonas = np.ones((num_actividades+1, num_actividades+1))
    primeras_actividades = [actividades[0] for actividades in
orden_actividades_por_paciente.values()]
    coste_inicial= calcular_calidad_solucion(FO_inicial)
    feromonas = ajustar_feromonas_iniciales(feromonas, solucion_inicial, coste_inicial)

    matriz_costo = generar_matriz_costos(actividades,orden_actividades_por_paciente)

    best_hormiga = [solucion_inicial]
    best_hormiga_calidad = [coste_inicial]
    best_hormiga_FO = [FO_inicial]

    tiempo_inicial = time.time()
    best_value = float('inf')
    iteracion = 0


    while (time.time() - tiempo_inicial) < tiempo_computacion:
        iteracion += 1

        soluciones_hormigas = []
        calidad_soluciones = []
        FO_soluciones = []
        tiempo_cero = time.time()

        # Generar soluciones para cada hormiga
        for hormiga in range(num_hormigas):

            solucion_hormiga = generar_solucion_hormiga(best_sol_i, feromonas,
matriz_costo, orden_actividades_por_paciente, actividades,alpha, beta, q0)
            FO_hormiga, df_datos_unido, df_pacientes = ejecuta_decoding(solucion_hormiga,
array_prioridades, df_pacientes, df_recursos_final, df_datos,array_pacientes)  #
Implementar evaluación de la solución. Calcular su FO
            soluciones_hormigas.append(solucion_hormiga)
            calidad_soluciones.append(calcular_calidad_solucion(FO_hormiga))
            FO_soluciones.append(FO_hormiga)

        indice_best_hormiga = FO_soluciones.index(min(FO_soluciones))
        # Guardamos la mejor solucion de esta vuelta
        best_hormiga.append(soluciones_hormigas[indice_best_hormiga])
        best_hormiga_calidad.append(calidad_soluciones[indice_best_hormiga])
        best_hormiga_FO.append(FO_soluciones[indice_best_hormiga])
        feromonas = actualizar_feromonas(feromonas,
soluciones_hormigas[indice_best_hormiga], calidad_soluciones[indice_best_hormiga],
evaporacion)

    best_sol = best_hormiga[best_hormiga_FO.index(min(best_hormiga_FO))]
    FO_best = min(best_hormiga_FO)
```

```
    return best_sol, df, FO_best
```

c. Celda para ejecutar ACO1:

```
q01 = 0.4
parametros = [2,0.5, 0.2] # alpha, beta, evaporacion
tiempo_computacion = (((len(df_datos)*15) /2)*375)/1000 # a las 12h
#tiempo_computacion = (((len(df_datos)*9) /2)*375)/1000 # a las 6h


num_hormigas = 500 # a las 12h
#num_hormigas = 100 # a las 6h


best_sol1 , FO_ACO1 = algoritmo_aco(tiempo_computacion,num_hormigas, parametros,
q01, FO_inicial_1, rep_solucion_1, actividades, orden_actividades_por_paciente,
df_pacientes, df_recursos_final, df_datos,array_pacientes)
```

C) Algoritmo de Colonia de Hormigas por actividades y sin nodo de INICIO (ACO2 y ACO3):

a. Funciones generales

```
import numpy as np
import time

# Utilizar la solución inicial para ajustar las feromonas
def ajustar_feromonas_iniciales(feromonas, solucion_inicial, calidad_inicial):
    for i in range(len(solucion_inicial) - 1):
        # Los indices de las actividades son uno menos
        feromonas[solucion_inicial[i]-1][solucion_inicial[i+1]-1] +=
calidad_inicial
    return feromonas

# Función para generar la matriz de costos, teniendo en cuenta las restricciones
# actividades : Actividad | Paciente  | Prioridad | TR  | Recursos_Necesarios |
Tiempo
def generar_matriz_costos(actividades,orden_actividades_por_paciente):
    num_actividades = len(actividades)
    matriz_costos = np.full((num_actividades, num_actividades), np.inf)

    # Asignar costos en base a tiempo de ocupación y prioridad del paciente
    for i in range(num_actividades):
        for j in range(num_actividades):
            if i != j:  # Asegurarse de que no se esté calculando el costo de
una actividad a sí misma
                # Solo se consideran transiciones válidas
```

```
                if es_transicion_valida(actividades[i][:2], actividades[j][:2],
orden_actividades_por_paciente):
                    tiempo = actividades[j][5]  # Tiempo de la actividad destino
                    prioridad = actividades[j][2]  # Prioridad del paciente de
la actividad destino
                    matriz_costos[i][j] = calcular_costo(tiempo, prioridad)

    return matriz_costos

def es_transicion_valida(actividad_origen, actividad_destino,
orden_actividades_por_paciente):
    paciente_origen = actividad_origen[1]
    paciente_destino = actividad_destino[1]
    actividad_origen_id = actividad_origen[0]
    actividad_destino_id = actividad_destino[0]

    # Validación intra-paciente
    if paciente_origen == paciente_destino:
        indice_origen =
orden_actividades_por_paciente[paciente_origen].index(actividad_origen_id)
        indice_destino =
orden_actividades_por_paciente[paciente_destino].index(actividad_destino_id)
        # Devuelve true solo si es justo la actividad que precede a la de origen
        return indice_destino == indice_origen + 1

    # Validación inter-paciente
    else:
      return True

# Calcular coste de ir a una actividad
def calcular_costo(tiempo, prioridad):
    base_cost = 10  # Costo base para cualquier transición
    costo = base_cost + tiempo - (6-prioridad)
    return costo


def seleccionar_proxima_actividad(alpha, beta, q0, actividades_validas,
actividades, feromonas, matriz_costos, orden_actividades_por_paciente,
actividades_seleccionadas_por_hormiga):

    if not actividades_validas:
        return None, actividades_validas

    actividad_actual = actividades_seleccionadas_por_hormiga[-1]
    transiciones_validas = []

    for actividad_destino in actividades_validas:
        # Verificar que todas las actividades previas del paciente asociado a
esta actividad se hayan completado
        for a in actividades:
```

```
            if actividad_destino == a[0]:
                paciente_destino = a[1]
                break

        actividades_previas =
orden_actividades_por_paciente[paciente_destino][:orden_actividades_por_paciente
[paciente_destino].index(actividad_destino)]

        if all(actividad_previa in actividades_seleccionadas_por_hormiga for
actividad_previa in actividades_previas):
            transiciones_validas.append(actividad_destino)

    if not transiciones_validas:  # Si no hay transiciones válidas
        return None, actividades_validas

    # Decision de explotación vs exploración
    if random.random() < q0:
        # Explotación: elegir el mejor basado en feromonas y costo (heurística)
        mejor_costo = 0
        seleccionado = None

        for actividad_destino in transiciones_validas:
            tau = feromonas[int(actividad_actual)-1][int(actividad_destino)-
1]  # Cantidad de feromona
            costo= matriz_costos[int(actividad_actual)-
1][int(actividad_destino)-1] # Valor costo
            eta = 1 / costo if costo != float('inf') else 0
            valor = (tau ** alpha) * (eta ** beta)
            if valor > mejor_costo:
                mejor_costo = valor
                seleccionado = actividad_destino

        actividades_validas.remove(seleccionado)
        return seleccionado, actividades_validas
    else:
        # Exploración: elegir basado en una distribución de probabilidades
proporcional a feromonas y heurística
        probabilidades = []
        for actividad_destino in transiciones_validas:
            tau = feromonas[int(actividad_actual)-1][int(actividad_destino)-
1]  # Cantidad de feromona
            costo= matriz_costos[int(actividad_actual)-
1][int(actividad_destino)-1] # Valor costo
            eta = 1 / costo if costo != float('inf') else 0

            probabilidades.append((tau ** alpha) * (eta ** beta))

        # Normalizar las probabilidades
        suma_probabilidades = sum(probabilidades)
```

```
        if suma_probabilidades == 0:
            return None, actividades_validas  # En caso de que todas las
probabilidades sean 0

        probabilidades_normalizadas = [prob / suma_probabilidades for prob in
probabilidades]
        seleccionado = np.random.choice(transiciones_validas,
p=probabilidades_normalizadas)
        actividades_validas.remove(seleccionado)
        return seleccionado, actividades_validas

def generar_solucion_hormiga(best_sol_i, feromonas,
matriz_costos,orden_actividades_por_paciente, actividades,
actividad_inicial,alpha, beta, q0):
  # Elegir una actividad de inicio en función del algoritmo
  actividades_seleccionadas_por_hormiga = []
  actividades_seleccionadas_por_hormiga.append(actividad_inicial)
  actividades_validas = copy.copy(best_sol_i)
  actividades_validas.remove(actividad_inicial)

  # Mientras que la solución no esté completa
  while len(actividades_seleccionadas_por_hormiga) < len(best_sol_i):
            prox_actividad, actividades_validas =
seleccionar_proxima_actividad(alpha, beta, q0,actividades_validas, actividades,
feromonas, matriz_costos, orden_actividades_por_paciente,
actividades_seleccionadas_por_hormiga)
            actividades_seleccionadas_por_hormiga.append(prox_actividad)

  return actividades_seleccionadas_por_hormiga

def actualizar_feromonas(matriz_feromonas, soluciones_hormigas,
calidad_soluciones, evaporacion):
    # Evaporación de feromonas
    matriz_feromonas *= (1 - evaporacion)

    for i in range(len(soluciones_hormigas)-1):
            # Aumentar las feromonas basado en la calidad de la solución
            matriz_feromonas[soluciones_hormigas[i]-1][soluciones_hormigas[i+1]-
1] += calidad_soluciones

    return matriz_feromonas

def calcular_calidad_solucion(FO_decoding):
    if FO_decoding > 0:
        return 100 / FO_decoding
    else:
        return float('inf')  # En caso de que tiempo_total sea 0, lo cual es
improbable en escenarios reales.
```

b. Función algoritmo principal:

```
# ACO2 y ACO3
def algoritmo_aco_2_3(tiempo_computacion,num_hormigas, parametros ,Q_0,
FO_inicial, solucion_inicial, actividades, orden_actividades_por_paciente,
df_pacientes, df_recursos_final, df_datos,array_pacientes):
    # Inicializar parametros
    num_actividades = len(actividades)  # Ajustar según tu caso

    alpha = parametros[0] # Influencia de la feromona
    beta = parametros [1] # Influencia de la información heurística
    evaporacion = parametros[2] # Tasa de evaporación de la feromona
    q0 = Q_0
    feromonas = np.ones((num_actividades, num_actividades))
    best_hormiga = []
    best_hormiga_calidad = []
    best_hormiga_FO = []
    coste_inicial= calcular_calidad_solucion(FO_inicial)
    feromonas = ajustar_feromonas_iniciales(feromonas, solucion_inicial,
coste_inicial)
    matriz_costo =
generar_matriz_costos(actividades,orden_actividades_por_paciente)
    best_hormiga.append(solucion_inicial)
    best_hormiga_calidad.append(coste_inicial)
    best_sol_i = solucion_inicial # ACO 2 Y 3
    tiempo_inicial = time.time()
    iteracion = 0

    while (time.time() - tiempo_inicial) < tiempo_computacion:
        iteracion += 1
        soluciones_hormigas = []
        calidad_soluciones = []
        FO_soluciones = []
        # Generar soluciones para cada hormiga
        for hormiga in range(num_hormigas):
```

```
            paciente_inicial =
np.random.choice(range(1,len(orden_actividades_por_paciente))) # ACTIVAR EN ACO
2

            actividad_inicial =
orden_actividades_por_paciente[paciente_inicial][0] # ACTIVAR EN ACO 2




                                                                        acti
vidad_inicial = best_sol_i[0] # ACTIVAR EN ACO 3



            solucion_hormiga = generar_solucion_hormiga(best_sol_i, feromonas,
matriz_costo, orden_actividades_por_paciente, actividades,
actividad_inicial,alpha, beta, q0)

            FO_hormiga, df_datos_unido, df_pacientes =
ejecuta_decoding(solucion_hormiga, array_prioridades, df_pacientes,
df_recursos_final, df_datos,array_pacientes)

            soluciones_hormigas.append(solucion_hormiga)

            calidad_soluciones.append(calcular_calidad_solucion(FO_hormiga))

            FO_soluciones.append(FO_hormiga)


        # Actualizar feromonas

        tiempo_fin = time.time()

        indice_best_hormiga = FO_soluciones.index(min(FO_soluciones))

        best_hormiga.append(soluciones_hormigas[indice_best_hormiga]) #
Guardamos la mejor solucion de esta vuelta

        best_hormiga_calidad.append(calidad_soluciones[indice_best_hormiga])

        best_hormiga_FO.append(FO_soluciones[indice_best_hormiga])

        feromonas = actualizar_feromonas(feromonas,
soluciones_hormigas[indice_best_hormiga],
calidad_soluciones[indice_best_hormiga], evaporacion)


    best_sol = best_hormiga[best_hormiga_FO.index(min(best_hormiga_FO))]

    FO_best = min(best_hormiga_FO)

    return best_sol, FO_best
```

c. Celda para ejecutar ACO2 y ACO3:

```
parametros = [2,0.5, 0.2] # alpha, beta, evaporacion
```

```
q0 = 0.4
tiempo_computacion = (((len(df_datos)*15) /2)*375)/1000 # a las 12h
#tiempo_computacion = (((len(df_datos)*9) /2)*375)/1000 # a las 6h

num_hormigas = 500 # a las 12h
#num_hormigas = 100 # a las 6h

# Activar dentro del algoritmo los parámetros según ACO2 o ACO3
best_sol , FO_ACO = algoritmo_aco_2_3(tiempo_computacion,num_hormigas,
parametros, q0, FO_inicial_1, rep_solucion_1, actividades,
orden_actividades_por_paciente, df_pacientes, df_recursos_final,
df_datos,array_pacientes)
```

D) Algoritmo de Colonia de Hormigas por pacientes

a. Funciones generales

```
import time
import matplotlib.pyplot as plt

# Utilizar la solución inicial para ajustar las feromonas
def ajustar_feromonas_iniciales(feromonas, solucion_inicial, coste_inicial):
    feromonas[0][solucion_inicial[0]] += coste_inicial
    for i in range(len(solucion_inicial) - 1):
        feromonas[solucion_inicial[i]][solucion_inicial[i+1]] += coste_inicial
    return feromonas

# Función para generar la matriz de costos - EN ESTE CASO NO HAY RESTRICCIONES DE
PRECEDENCIA
# actividades : Actividad | Paciente  | Prioridad | TR  | Recursos_Necesarios | Tiempo
def generar_matriz_costos(df_pacientes):
    n_pacientes = len(df_pacientes)  # Número de pacientes
    matriz_costos = np.full((n_pacientes +1, n_pacientes+1), np.inf)

    for i in range(n_pacientes +1):
        for j in range(1, n_pacientes +1):
            if i != j :
                prioridad_j = df_pacientes.at[j-1,'Prioridad']
                visto_j = df_pacientes.at[j-1, 'visto']
                # Calcula el costo según la prioridad y si ha sido visto
                costo = (1 / (6 - prioridad_j)) * (1 - 0.2 * visto_j)
                matriz_costos[i][j] = costo

    return matriz_costos


def seleccionar_proximo_paciente( pacientes_seleccionados_por_hormiga,  matriz_feromonas,
matriz_costos, alpha, beta, q0):
    ultimo_paciente = pacientes_seleccionados_por_hormiga[-1]
```

```
    candidatos = [i for i in range(1,len(matriz_costos)) if i not in
pacientes_seleccionados_por_hormiga]
    probabilidades = []

    if random.random() < q0:  # Explotación: selecciona el mejor candidato con alta
probabilidad
        mejor_costo = 0
        seleccionado = None
        for candidato in candidatos:
            tau = matriz_feromonas[ultimo_paciente][candidato]
            costo = matriz_costos[ultimo_paciente][candidato]
            eta = 1 / costo if costo != float('inf') else 0
            valor = (tau ** alpha) * (eta ** beta)

            if valor > mejor_costo:
                mejor_costo = valor
                seleccionado = candidato

    else:  # Exploración: selecciona un candidato basado en una distribución de
probabilidad
      for candidato in candidatos:
        tau = 10*60
        matriz_feromonas[ultimo_paciente][candidato]
        eta = 1/matriz_costos[ultimo_paciente][candidato]
        probabilidades.append((tau ** alpha) * (eta ** beta))

      suma_probabilidades = sum(probabilidades)
      probabilidades_normalizadas = [p/suma_probabilidades for p in probabilidades]
      seleccionado = random.choices(candidatos, weights=probabilidades_normalizadas,
k=1)[0]

    return seleccionado

def actualizar_feromonas(matriz_feromonas, solucion, calidad, tasa_evaporacion):

    for i in range(len(matriz_feromonas)):
      for j in range(len(matriz_feromonas)):
        matriz_feromonas[i][j] *= (1 - tasa_evaporacion)

    # Actualizamos tambien el el arco del nodo inicio
    matriz_feromonas[0][solucion[0]] += calidad
    for i in range(len(solucion)-1):
           matriz_feromonas[solucion[i]][solucion[i+1]] += calidad

    return matriz_feromonas


def calcular_calidad_solucion(FO_decoding):
    if FO_decoding > 0:
        return 100 / FO_decoding
```

```
    else:
        return float('inf')
```

b. Función ACO principal:

```
# Algoritmo principal
def algoritmo_aco_pacientes(tiempo_computacion, FO_inicial,solucion_inicial,
num_hormigas, df_pacientes ,df_datos, alpha, beta, ro,q0,array_pacientes):
    num_pacientes = len(df_pacientes)

    # Inicialización de la matriz de feromonas
    matriz_feromonas = [[1 for _ in range(num_pacientes+1)] for _ in
range(num_pacientes+1)]

    # Inicializamos feromonas según la solución inicial
    calidad_inicial = calcular_calidad_solucion(FO_inicial)
    print(calidad_inicial)
    feromonas = ajustar_feromonas_iniciales(matriz_feromonas, solucion_inicial,
calidad_inicial)
    matriz_costos = generar_matriz_costos(df_pacientes)

    mejor_solucion = []
    mejor_solucion_calidad = []
    mejor_solucion_FO = []
    tiempo_inicial = time.time()
    iteracion = 0

    while (time.time() - tiempo_inicial) < tiempo_computacion:
        soluciones = []  # Lista para almacenar las soluciones de esta iteración
        calidad_soluciones = []
        FO_soluciones = []
        iteracion += 1
        for n in range(num_hormigas):
          pacientes_seleccionados_por_hormiga = [0]  # Inicia en nodo de INICIO
          while len(pacientes_seleccionados_por_hormiga) < num_pacientes+1:
              proximo_paciente =
seleccionar_proximo_paciente(pacientes_seleccionados_por_hormiga, matriz_feromonas,
matriz_costos, alpha, beta, q0)
              pacientes_seleccionados_por_hormiga.append(proximo_paciente)


          pacientes_seleccionados_por_hormiga = pacientes_seleccionados_por_hormiga[1:]


          ### CONVERTIMOS EN UNA SOLUCIÓN QUE PUEDA EJECUTAR NUESTRO DECODING
```

```
          # Organizamos las actividades en función del orden establecido por la hormiga
          matriz_actividades =
df_datos.groupby('Paciente')['Actividad'].apply(list).reindex(pacientes_seleccionados_por
_hormiga).tolist() # solucion = [2, 1, 3, 4]
          # Obtener la longitud máxima de las sublistas en Pu_ordenado
          longitud_maxima = max(map(len, matriz_actividades))
          # Rellenar con ceros las sublistas que no llegan al máximo para poder
transponerla sin perder actividades
          Pu_transpuesto = [sublista + ['0'] * (longitud_maxima - len(sublista)) for
sublista in matriz_actividades]
          Pu_transpuesto = list(map(list, zip(*Pu_transpuesto)))
          rep_solucion = []
          for i in Pu_transpuesto:
            for j in i:
              if j!='0':
                rep_solucion.append(j)

          FO_hormiga, df_datos_unido, df_pacientes = ejecuta_decoding(rep_solucion,
array_prioridades, df_pacientes, df_recursos_final, df_datos,array_pacientes)
          soluciones.append(pacientes_seleccionados_por_hormiga)
          calidad_soluciones.append(calcular_calidad_solucion(FO_hormiga))
          FO_soluciones.append(FO_hormiga)


        indice_mejor_hormiga = FO_soluciones.index(min(FO_soluciones))
        mejor_solucion.append(soluciones[indice_mejor_hormiga])
        mejor_solucion_calidad.append(calidad_soluciones[indice_mejor_hormiga])
        mejor_solucion_FO.append(FO_soluciones[indice_mejor_hormiga])

        # Actualiza la matriz de feromonas con la mejor solucion encontrada
        matriz_feromonas = actualizar_feromonas(matriz_feromonas,
soluciones[indice_mejor_hormiga], calidad_soluciones[indice_mejor_hormiga],ro)



    mejor_solucion_global =
mejor_solucion[mejor_solucion_FO.index(min(mejor_solucion_FO))]
    mejor_FO = min(mejor_solucion_FO)
    return mejor_solucion_global, mejor_FO
```

c. Celda para ejecutar ACO4:

```
solucion_inicial = array_pacientes
df_pacientes = df_pacientes.sort_values(by='Paciente')
q04 = 0.4
alpha= 2
beta = 0.5
```

```
tasa_evaporacion=0.2
# Parametros a calibrar
tiempo_computacion = (((len(df_datos)*15) /2)*375)/1000 # a las 12h
#tiempo_computacion = (((len(df_datos)*9) /2)*375)/1000 # a las 6h


num_hormigas = 500 # a las 12h
#num_hormigas = 100 # a las 6h
best_sol4, FO_ACO4 = algoritmo_aco_pacientes(tiempo_computacion,
FO_inicial_1,solucion_inicial, num_hormigas, df_pacientes ,df_datos, alpha,
beta, tasa_evaporacion,q04,array_pacientes)
```

E) Búsqueda Local

```
def select_two_patients(solution):
    indices = np.random.choice(len(solution), 2, replace=False)
    return [solution[i] for i in indices]

def busqueda_local(solucion_inicial, df_datos, df_pacientes, df_recursos_final):
    current_solution = solucion_inicial[:]

    pacientes_sacados = select_two_patients(current_solution)

    print('Sacados:', pacientes_sacados)

    df_pacientes_nuevo = df_pacientes
    pacientes_mantenidos = [p for p in current_solution if p not in pacientes_sacados]

    df_datos_mantenidos = df_datos[~df_datos['Paciente'].isin(pacientes_sacados)]
    df_pacientes_mantenidos =
df_pacientes[~df_pacientes_nuevo['Paciente'].isin(pacientes_sacados)]

    # Obtener las filas correspondientes a los pacientes sacados
    df_datos_sacados = df_datos[df_datos['Paciente'].isin(pacientes_sacados)]
    df_pacientes_sacados = df_pacientes[df_pacientes['Paciente'].isin(pacientes_sacados)]


    best_solution = None

    # Probar cada paciente sacado en todas las posiciones posibles del array restante
    for p in pacientes_sacados:

        best_score = float('inf')

        # Seleccionamos las filas del paciente que estudiamos
        df_datos_paciente = df_datos_sacados[df_datos_sacados['Paciente'] == p]
        df_pacientes_paciente = df_pacientes_sacados[df_pacientes_sacados['Paciente'] ==
```

```
p]

        # Las añadimos en el dataframe (da igual orden, luego se ordena según la
rep_solución)
        df_datos_mantenidos = pd.concat([df_datos_paciente, df_datos_mantenidos],
ignore_index=True)
        df_pacientes_mantenidos = pd.concat([df_pacientes_paciente,
df_pacientes_mantenidos], ignore_index=True)

        for i in range(len(pacientes_mantenidos) + 1):
            # Insertar el paciente en la posición i
            new_solution = pacientes_mantenidos[:i] + [p] + pacientes_mantenidos[i:]

            print(new_solution)
            # Organizamos las actividades en función del orden establecido por la hormiga
            matriz_actividades =
df_datos_mantenidos.groupby('Paciente')['Actividad'].apply(list).reindex(new_solution).to
list()

            # Obtener la longitud máxima de las sublistas en Pu_ordenado
            longitud_maxima = max(map(len, matriz_actividades))

            # Rellenar con ceros las sublistas que no llegan al máximo para poder
transponerla sin perder actividades
            Pu_transpuesto = [sublista + ['0'] * (longitud_maxima - len(sublista)) for
sublista in matriz_actividades]

            # Transponer la lista resultante
            Pu_transpuesto = list(map(list, zip(*Pu_transpuesto)))
            rep_solucion = []
            for i in Pu_transpuesto:
              for j in i:
                if j!='0':
                  rep_solucion.append(j)

            # Evaluar la nueva solución
            FO_busqueda , df_datos_unido, df_pacientes_decoding =
ejecuta_decoding(rep_solucion,df_pacientes_mantenidos, df_recursos_final,
df_datos_mantenidos,solucion_inicial)
            print(FO_busqueda)

            # Si la nueva solución es mejor, actualizar la mejor solución
            if FO_busqueda < best_score:
                best_solution = new_solution
                best_score = FO_busqueda

        # Actualizar pacientes mantenidos para la siguiente iteración
        pacientes_mantenidos = best_solution[:]
```

```
    return best_solution, best_score
```

a. ACO1 + BL

```
# Algoritmo principal
def algoritmo_aco(tiempo_computacion,num_hormigas, parametros ,Q_0, FO_inicial,
solucion_inicial, actividades, orden_actividades_por_paciente, df_pacientes,
df_recursos_final, df_datos, array_pacientes):
    # Inicializar parametros
    num_actividades = len(actividades)  # Ajustar según tu caso

    alpha = parametros[0] # Influencia de la feromona
    beta = parametros [1] # Influencia de la información heurística
    evaporacion = parametros[2] # Tasa de evaporación
    q0 = Q_0

    feromonas = np.ones((num_actividades+1, num_actividades+1))
    primeras_actividades = [actividades[0] for actividades in
orden_actividades_por_paciente.values()]
    coste_inicial= calcular_calidad_solucion(FO_inicial)
    feromonas = ajustar_feromonas_iniciales(feromonas, solucion_inicial, coste_inicial)


    matriz_costo = generar_matriz_costos(actividades,orden_actividades_por_paciente)

    best_hormiga = [solucion_inicial]
    best_hormiga_calidad = [coste_inicial]
    best_hormiga_FO = [FO_inicial]

    tiempo_inicial = time.time()
    iteracion = 0
    bl = 0
    orden_pacientes = array_pacientes

    while (time.time() - tiempo_inicial) < tiempo_computacion:
        iteracion += 1
        soluciones_hormigas = []
        calidad_soluciones = []
        FO_soluciones = []

        # Generar soluciones para cada hormiga
        for hormiga in range(num_hormigas):
            if bl == 0:
               solucion_hormiga = generar_solucion_hormiga(best_sol_i, feromonas,
matriz_costo, orden_actividades_por_paciente, actividades,alpha, beta, q0)
            else:

              # Organizamos las actividades en función del orden establecido por la
hormiga
```

```
            matriz_actividades =
df_datos.groupby('Paciente')['Actividad'].apply(list).reindex(best_sol_bl).tolist()
            # Obtener la longitud máxima de las sublistas en Pu_ordenado
            longitud_maxima = max(map(len, matriz_actividades))
            # Rellenar con ceros las sublistas que no llegan al máximo para poder
transponerla sin perder actividades
            Pu_transpuesto = [sublista + ['0'] * (longitud_maxima - len(sublista)) for
sublista in matriz_actividades]

            # Transponer la lista resultante
            Pu_transpuesto = list(map(list, zip(*Pu_transpuesto)))
            rep_solucion = []
            for i in Pu_transpuesto:
              for j in i:
                if j!='0':
                  rep_solucion.append(j)

            solucion_hormiga = rep_solucion
            bl = 0

          FO_hormiga, df_datos_unido, df_pacientes = ejecuta_decoding(solucion_hormiga,
df_pacientes, df_recursos_final, df_datos,array_pacientes)
          soluciones_hormigas.append(solucion_hormiga)
          calidad_soluciones.append(calcular_calidad_solucion(FO_hormiga))
          FO_soluciones.append(FO_hormiga)

      # Actualizar feromonas
      tiempo_fin = time.time()
      indice_best_hormiga = FO_soluciones.index(min(FO_soluciones))
      best_hormiga.append(soluciones_hormigas[indice_best_hormiga]) # Guardamos la
mejor solucion de esta vuelta
      best_hormiga_calidad.append(calidad_soluciones[indice_best_hormiga])
      best_hormiga_FO.append(FO_soluciones[indice_best_hormiga])
      feromonas = actualizar_feromonas(feromonas,
soluciones_hormigas[indice_best_hormiga], calidad_soluciones[indice_best_hormiga],
evaporacion)

      bl = 1
      best_sol_bl , best_FO_bl = busqueda_local(orden_pacientes, df_datos,
df_pacientes,df_recursos_final)
      orden_pacientes = best_sol_bl

  best_sol = best_hormiga[best_hormiga_FO.index(min(best_hormiga_FO))]
  FO_best = min(best_hormiga_FO)
  return best_sol, FO_best
```

b. ACO2 + BL // ACO3 + BL

```
def algoritmo_aco_2_3(tiempo_computacion, num_hormigas, parametros ,Q_0,
FO_inicial, solucion_inicial, actividades, orden_actividades_por_paciente,
df_pacientes, df_recursos_final, df_datos, array_pacientes):
    # Inicializar parametros
    num_actividades = len(actividades)  # Ajustar según tu caso

    alpha = parametros[0] # Influencia de la feromona
    beta = parametros [1] # Influencia de la información heurística
    evaporacion = parametros[2] # Tasa de evaporación
    q0 = Q_0
    feromonas = np.ones((num_actividades, num_actividades))
    best_hormiga = []
    best_hormiga_calidad = []
    best_hormiga_FO = []
    coste_inicial= calcular_calidad_solucion(FO_inicial)
    feromonas = ajustar_feromonas_iniciales(feromonas, solucion_inicial,
coste_inicial)
    matriz_costo =
generar_matriz_costos(actividades,orden_actividades_por_paciente)

    best_hormiga.append(solucion_inicial)
    best_hormiga_calidad.append(coste_inicial)
    best_sol_i = solucion_inicial # ACO 2 Y 3
    tiempo_inicial = time.time()
    iteracion = 0
    bl = 0
    orden_pacientes = array_pacientes

    while (time.time() - tiempo_inicial) < tiempo_computacion:
        iteracion += 1

        soluciones_hormigas = []
        calidad_soluciones = []
        FO_soluciones = []
        tiempo_cero = time.time()
        # Generar soluciones para cada hormiga
```

```
        for hormiga in range(num_hormigas):
            if bl == 0:

              actividad_inicial = best_sol_i[0] # ACO 3
            else:

              # Organizamos las actividades en función del orden establecido por
la hormiga
              matriz_actividades =
df_datos.groupby('Paciente')['Actividad'].apply(list).reindex(best_sol_bl).tolis
t() # solucion = [2, 1, 3, 4]
              # Obtener la longitud máxima de las sublistas en Pu_ordenado
              longitud_maxima = max(map(len, matriz_actividades))
              # Rellenar con ceros las sublistas que no llegan al máximo para
poder transponerla sin perder actividades
              Pu_transpuesto = [sublista + ['0'] * (longitud_maxima -
len(sublista)) for sublista in matriz_actividades]

              # Transponer la lista resultante
              Pu_transpuesto = list(map(list, zip(*Pu_transpuesto)))
              rep_solucion = []
              for i in Pu_transpuesto:
                for j in i:
                  if j!='0':
                    rep_solucion.append(j)

              solucion_hormiga = rep_solucion
              bl = 0

            solucion_hormiga = generar_solucion_hormiga(best_sol_i, feromonas,
matriz_costo, orden_actividades_por_paciente, actividades,
actividad_inicial,alpha, beta, q0)

            FO_hormiga, df_datos_unido, df_pacientes =
ejecuta_decoding(solucion_hormiga, array_prioridades, df_pacientes,
df_recursos_final, df_datos,array_pacientes)
```

```
            soluciones_hormigas.append(solucion_hormiga)
            calidad_soluciones.append(calcular_calidad_solucion(FO_hormiga))
            FO_soluciones.append(FO_hormiga)



        # Actualizar feromonas
        tiempo_fin = time.time()
        indice_best_hormiga = FO_soluciones.index(min(FO_soluciones))
        best_hormiga.append(soluciones_hormigas[indice_best_hormiga]) #
Guardamos la mejor solucion de esta vuelta
        best_hormiga_calidad.append(calidad_soluciones[indice_best_hormiga])
        best_hormiga_FO.append(FO_soluciones[indice_best_hormiga])
        feromonas = actualizar_feromonas(feromonas,
soluciones_hormigas[indice_best_hormiga],
calidad_soluciones[indice_best_hormiga], evaporacion)


        best_sol_i =
best_hormiga[best_hormiga_calidad.index(max(best_hormiga_calidad))] # Sirve solo
en caso de lanzar ACO_3 (si no solo vale para saber las actividades que hay)


        bl = 1
        best_sol_bl , best_FO_bl = busqueda_local(orden_pacientes, df_datos,
df_pacientes,df_recursos_final)
        orden_pacientes = best_sol_bl


    best_sol = best_hormiga[best_hormiga_FO.index(min(best_hormiga_FO))]
    FO_best = min(best_hormiga_FO)
    return best_sol, FO_best
```

c. ACO4 + BL

```
# Algoritmo principal
def algoritmo_aco_pacientes(tiempo_computacion, FO_inicial,solucion_inicial,
num_hormigas, df_pacientes ,df_datos, alpha, beta, ro,q0,array_pacientes):
    num_pacientes = len(df_pacientes)

    # Inicialización de la matriz de feromonas
    matriz_feromonas = [[1 for _ in range(num_pacientes+1)] for _ in
range(num_pacientes+1)]
```

```
    calidad_inicial = calcular_calidad_solucion(FO_inicial)
    feromonas = ajustar_feromonas_iniciales(matriz_feromonas, solucion_inicial,
calidad_inicial)
    matriz_costos = generar_matriz_costos(df_pacientes)

    mejor_solucion = []
    mejor_solucion_calidad = []
    mejor_solucion_FO = []
    tiempo_inicial = time.time()
    iteracion = 0
    bl = 0
    orden_pacientes = array_pacientes

    while (time.time() - tiempo_inicial) < tiempo_computacion:
        soluciones = []
        calidad_soluciones = []
        FO_soluciones = []
        iteracion += 1
        for n in range(num_hormigas):
          if bl == 0:
                                                                                #
Partimos desde el nodo INICIO
            pacientes_seleccionados_por_hormiga = [0]


            while len(pacientes_seleccionados_por_hormiga) < num_pacientes+1:
                proximo_paciente =
seleccionar_proximo_paciente_1(pacientes_seleccionados_por_hormiga, matriz_feromonas,
matriz_costos, alpha, beta, q0)
                pacientes_seleccionados_por_hormiga.append(proximo_paciente)


            pacientes_seleccionados_por_hormiga = pacientes_seleccionados_por_hormiga[1:]

          else:
            pacientes_seleccionados_por_hormiga = best_sol_bl


          ### CONVERTIMOS EN UNA SOLUCIÓN QUE PUEDA EJECUTAR NUESTRO DECODING

          # Organizamos las actividades en función del orden establecido por la hormiga
          matriz_actividades =
df_datos.groupby('Paciente')['Actividad'].apply(list).reindex(pacientes_seleccionados_por
_hormiga).tolist() # solucion = [2, 1, 3, 4]
          # Obtener la longitud máxima de las sublistas en Pu_ordenado
          longitud_maxima = max(map(len, matriz_actividades))
          # Rellenar con ceros las sublistas que no llegan al máximo para poder
transponerla sin perder actividades
          Pu_transpuesto = [sublista + ['0'] * (longitud_maxima - len(sublista)) for
```

```
sublista in matriz_actividades]

          # Transponer la lista resultante
          Pu_transpuesto = list(map(list, zip(*Pu_transpuesto)))
          rep_solucion = []
          for i in Pu_transpuesto:
            for j in i:
              if j!='0':
                rep_solucion.append(j)

          FO_hormiga, df_datos_unido, df_pacientes = ejecuta_decoding(rep_solucion,
df_pacientes, df_recursos_final, df_datos,array_pacientes)
          soluciones.append(pacientes_seleccionados_por_hormiga)
          calidad_soluciones.append(calcular_calidad_solucion(FO_hormiga))
          FO_soluciones.append(FO_hormiga)

        indice_mejor_hormiga = FO_soluciones.index(min(FO_soluciones))
        mejor_solucion.append(soluciones[indice_mejor_hormiga])
        mejor_solucion_calidad.append(calidad_soluciones[indice_mejor_hormiga])
        mejor_solucion_FO.append(FO_soluciones[indice_mejor_hormiga])

                                                                                ##
ACTIVAMOS BUSQUEDA LOCAL
        bl = 1
        best_sol_bl , best_FO_bl = busqueda_local(orden_pacientes, df_datos,
df_pacientes,df_recursos_final)
        orden_pacientes = best_sol_bl

        # Actualiza la matriz de feromonas con la mejor solucion encontrada
        matriz_feromonas = actualizar_feromonas(matriz_feromonas,
soluciones[indice_mejor_hormiga], calidad_soluciones[indice_mejor_hormiga],ro)

    # Devuelve el mejor camino y su costo después de todas las iteraciones
    mejor_solucion_global =
mejor_solucion[mejor_solucion_FO.index(min(mejor_solucion_FO))]
    mejor_FO = min(mejor_solucion_FO)
    return mejor_solucion_global, mejor_FO
```