Trabajo Fin de Grado
Grado en Ingeniería de Tecnologías Industriales

Acercamiento al problema de planificación de personal mediante técnicas de optimización no exactas

Autor: Fátima Rebollo Molina

Tutores: Jesús Muñuzuri Sanz
Antonio Lorenzo Espejo

Dpto. Organización Industrial y Gestión de Empresas II

Sevilla, 2023

Trabajo Fin de Grado
Grado en Ingeniería de Tecnologías Industriales

Acercamiento al problema de planificación de personal mediante técnicas de optimización no exactas

Autor:
Fátima Rebollo Molina

Tutores:
Jesús Muñuzuri Sanz
Antonio Lorenzo Espejo

Dpto. de Organización Industrial y Gestión de Empresas II
Sevilla, 2023

Autor: Fátima Rebollo Molina

Cotutor: Jesús Muñuzuri Sanz

Cotutor: Antonio Lorenzo Espejo

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

Presidente:

Vocales:

Secretario:

Acuerdan otorgarle la calificación de:

Sevilla, 2023

El Secretario del Tribunal

A mi familia

A mis profesores

Resumen

El problema de planificación de personal es un reto siempre de actualidad en la Organización de Empresas, ya que continuamente nacen nuevas regulaciones y metodologías de trabajo a las que hay que adaptar la planificación. El caso del personal de practicaje de un puerto es especialmente delicado por las condiciones que lo rodean, y por tanto debe afrontarse mediante metodologías desarrolladas *ad hoc*.

Este trabajo se basa en un enfoque innovador para la optimización de dicha planificación: la satisfacción de los empleados y la igualdad de los horarios definidos. Mediante la utilización de un algoritmo genético, este trabajo pretende resolver un caso real de los prácticos de un puerto, de forma que pueda reducirse el tiempo de computación y obtenerse resultados que mejoren las condiciones laborales de los empleados.

Abstract

The personnel scheduling problem poses an ever-challenging scenario for optimization in the business management framework throughout history, having to adapt to the multiple regulations and work methodologies that continually arise. The particular case of maritime pilots is critical to provide quality service to the transportation system and, thus, it should be faced by using methodologies which are specifically developed *ad hoc*.

This paper aims to provide meaningful results to the aforementioned case by applying an innovative approach: taking pilots satisfaction and equality of the resulting schedules into consideration; and using a Genetic Algorithm to try and solve a real case study, thus minimizing the amount of computation needed and reaching optimal solutions which help provide better working conditions for the personnel in a faster way.

Índice

1 INTRODUCCIÓN

El problema de planificación del personal es uno de los mayores retos de la optimización moderna. Se da en todas las industrias, incluyendo salud, industria y comercio, y su objetivo es asignar empleados (recursos) a turnos o tareas de forma que se cumplan tanto los requisitos operativos como las restricciones impuestas por las leyes laborales, preferencias de los empleados y objetivos empresariales. Por ello, el gran número de variables y la naturaleza combinatoria del problema ha aumentado la dificultad para encontrar una solución óptima.

Edie (1954) y Dantzig (1954) fueron los primeros en estudiar esta cuestión, analizando las mejores formas de organizar al personal en los puestos de peaje de los túneles de Nueva York pero desde entonces su complejidad y envergadura ha ido aumentando a la par que lo hacía su importancia: el foco de las empresas está puesto desde hace años en la optimización y la eliminación de las ineficiencias; y se ha demostrado que la administración del personal es uno de los factores de mayor importancia en la reducción de estas ineficiencias (Alfares, 2004).

En realidad, el problema de planificación de personal no es único, sino que se trata de aglutinar numerosos escenarios que se dan a diario en la industria, y que han sido clasificados y estudiados a lo largo de la historia, como veremos más adelante en estas páginas. Entre ellos, se pueden destacar la asignación de turnos en la jornada laboral; la asignación de tareas a empleados en función de sus habilidades y disponibilidad, que se da sobre todo en la gestión de proyectos; la planificación de horarios, que resulta especialmente relevante en industrias como la atención médica o la hostelería; y por último, la planificación de descansos o *days-off scheduling.*

Este último tipo de problema se enfoca en la asignación de días de descanso a la plantilla a lo largo de un horizonte temporal determinado, que puede variar en función de la industria o las necesidades. El objetivo siempre es encontrar un equilibrio entre proporcionar los descansos adecuados a los trabajadores y mantener la operatividad de la organización.

Cuando el problema de planificación de descansos se une a tipos de trabajo que añaden peligrosidad, fatiga, turnos rotativos u horarios irregulares, nos encontramos con modelos operativos que incluyen un gran número de restricciones y consideraciones a tener en cuenta. Normalmente, es esencial garantizar que los empleados tengan suficientes descansos no solo para cumplir con las restricciones que suelen estar vigentes, sino también para mantener al personal activo, motivado y en las mejores condiciones operativas posibles; ya que no en vano son recursos humanos. Estas limitaciones pueden incluir la necesidad de proporcionar descansos consecutivos, mantener un equilibrio entre los descansos largos y cortos y respetar las preferencias individuales de los empleados en la medida de lo posible.

Este es el caso que se pretende resolver a lo largo de este trabajo: la planificación de descansos para el personal de prácticos de un puerto. Por las estrictas regulaciones a las que está sometido el transporte marítimo de grandes buques en cuanto a tiempos de entrega, así como la especial situación de estrés y peligrosidad a la que se enfrentan los prácticos que han de guiarlos hacia el puerto; este es uno de los colectivos con mayor regulación en cuanto a sus jornadas laborales y los descansos que deben cumplir.

En general, desde un punto de vista académico, los métodos de resolución preferidos para abordar la planificación de personal han sido la programación matemática y la metaheurística ; y, aunque como veremos más adelante existen numerosas aplicaciones y clasificaciones para esta tipología de problemas, en estas páginas abordaremos el caso de la planificación de los días de descanso de los prácticos de un puerto español.

Este trabajo busca hacer una comparación entre los resultados arrojados por metaheurísticas en comparación con los que devuelven los modelos de programación matemática creados por Lorenzo-Espejo et al. (2021, 2023). que utilizan distintos enfoques para fijar o maximizar los períodos de descanso que tiene cada empleado y que se aplicó de forma específica a la plantilla de un puerto español teniendo en cuenta sus sugerencias de asignarles un único descanso largo o dos cortos a lo largo de cada horizonte de planificación.

Teniendo en cuenta la ya mencionada naturaleza del problema y su modelado, se ha escogido como método de resolución el algoritmo genético, que ya ha sido aplicado con éxito a casos similares que veremos a lo largo del desarrollo de este documento.

Durante el resto del trabajo se explorará esta temática en el siguiente orden: en primer lugar, se presenta una revisión de la literatura, seguida del planteamiento del problema. En la sección 4 presentaremos los modelos utilizados por los autores previamente mencionados y el modelo de resolución con metaheurísticas elaborado para este trabajo. Por último, se presentarán los resultados y su comparación con los dados por los modelos exactos.

2 REVISIÓN DE LA LITERATURA

2.1 Definición y clasificaciones del problema

Se hace ahora necesario ampliar la definición y profundidad del problema de asignación de personal que aportamos al comienzo de este trabajo. En realidad, bajo este título se aglutinan una serie de problemas con una estructura similar, que difieren en cuanto a restricciones y objetivos. Todo esto añade a la dificultad de la cuestión que nos atañe, ya que además de encontrar una solución óptima, se han de definir y establecer un dominio general para la resolución del problema, un objetivo a alcanzar y unas restricciones- que pueden ser duras o blandas (Courtois et al., 2010).

Así, el denominado problema de asignación de personal ha sido objeto de numerosos estudios a lo largo de los años, motivados no solo por el interés académico, sino también por la relevancia que éste tiene en su aplicación a un entorno empresarial que cada vez cuenta con más regulaciones de tipo laboral, de salud, etc.

A todas ellas hay que sumar el hecho de que la evolución se está produciendo, como hemos mencionado anteriormente, hacia una cultura de máxima optimización o eficiencia. Metodologías como el Lean, el Six-Sigma o el Agile -que no solo se aplican a la producción, sino a toda la organización de una empresa- están a la orden del día, y todas ellas comparten el común propósito de eliminar cualquier actividad residual. De forma evidente, eso atañe más claramente a la correcta planificación del personal, uno de los factores que más influyen en esta filosofía de mejora continua. No se trata solo de que el coste del personal supone la mayoría del coste total en que incurre una empresa- y, por tanto, este ha de tener el mayor rendimiento posible- sino de la importancia que tiene en los resultados de una compañía la buena disposición de los trabajadores a la hora de obtenerlos.

Es por ello que, en los últimos años, otras variables y restricciones han tomado peso en esta serie de problemas; que hacen referencia a la gestión de la carga de trabajo que se le asigna a cada empleado y la satisfacción de estos (Van den Bergh et al., 2013). Todo ello son circunstancias que influyen en la escala y el grado en que un problema es resoluble.

Por todo esto, muchas de las investigaciones realizadas a este respecto comienzan haciendo una clasificación de los tipos de problemas con los que podemos encontrarnos según diferentes criterios de división, que veremos a lo largo de esta sección.

2.1.1 Clasificaciones

De toda la investigación que se ha hecho sobre la temática, la primera clasificación del problema fue propuesta por Baker (1976), dividiéndolos en 3 grupos distintos:

- *Shift scheduling problems*, donde hay que asignar una serie de turnos a los empleados durante la jornada laboral, en un horizonte de planificación diario. Esto implica que la asignación de cada turno se puede realizar de forma independiente y que la solución es fácil de encontrar e implementar. Sin embargo, esto solo respondería a aquellas situaciones en las que la demanda se comporta de forma lineal durante la duración de cada turno, es decir, no hay picos de demanda de duración menor que la de cada una de las posibles asignaciones. Si esto ocurriera, se produciría un solapamiento, y por tanto sería necesaria la creación de un modelo que contemplara turnos solapados.

- *Days-off scheduling problems*, donde hay que cuadrar las semanas operativas de la empresa con respecto a las semanas de descanso de los empleados, garantizando que la demanda de plantilla se cumpla. Aquí es donde más restricciones encontraremos, contando con periodos de alta necesidad de producción o preferencias de los empleados, que suelen pedir los días libres consecutivos y semanas de un máximo de 5 días laborables en los casos en los que la compañía se mantiene operativa de forma continua.
- *Tour scheduling problems*, que combinan aspectos de los anteriores: divergencias entre las semanas operativas de la empresa y los trabajadores y asignación de varios turnos por día, como ocurre por ejemplo en hospitales, aerolíneas, hoteles…Esto tiene como resultado horarios en los que los empleados tienen tanto días como turnos libres, y cuya complejidad viene determinada en gran parte por la duración del mínimo intervalo de planificación (del turno). Según Baker, los resultados del problema de asignación serían dados como entrada a los modelos de resolución de la programación de días libres, integrándolos de esa manera.

Baker creó estas divisiones basándose en la naturaleza cíclica de la planificación, organizándolos de forma que cada uno de ellos responde a resolución de un ciclo (un turno, un día…). En su investigación, hace hincapié en el hecho de que los métodos de resolución tradicionales utilizan un sistema con un enfoque descendente del problema, solucionando los ciclos más cortos primero; y en que la solución óptima pasaría por integrar ambas resoluciones en una.

Sin embargo, esta no es la única forma de entender los diferentes tipos de planificación. De Causmaecker et al. (2004) propusieron una división alternativa basándose en encuestas realizadas a diversas compañías de la siguiente forma:

- Planificación centrada en la permanencia, donde la demanda de mano de obra viene determinada de antemano y se requiere un servicio continuo.
- Planificación centrada en la movilidad, donde los empleados han de desarrollar sus tareas en distintas localizaciones. Según esta investigación, las reorganizaciones del trabajo planeado son frecuentes en este tipo de escenarios, así como la necesidad de planificar también los medios de transporte que los trabajadores utilizarán para desplazarse a estos distintos lugares.
- Planificación centrada en la fluctuación, donde la demanda de mano de obra está en constante cambio y que se basa en los datos históricos para predecir futuros picos.
- Planificación centrada en proyectos, donde las compañías se organizan en porfolios de proyectos. Así, han de asignar grupos de trabajadores a cada uno de ellos y después redistribuirlos conforme éstos avanzan en cronología y necesidad de mano de obra. Como en el caso anterior, el método más utilizado es el de predicción basada en los modelos históricos, siendo aquí más ineficaces por las diferencias que se dan entre proyectos. Esto, unido a la asignación en grupo del personal, son las mayores diferencias entre la planificación centrada en proyectos y centrada en la fluctuación.

Otro popular método de clasificación es el propuesto por Betchtold et al. (1991), que clasifican este tipo de problemas según el método de resolución aplicado en dos categorías: aquellos que se resuelven mediante programación lineal y los que lo hacen basados en su construcción. Posteriormente, otros autores han añadido más divisiones según este criterio. Alfares (2004), propone diez categorías: (1) soluciones manuales, (2) Programación entera, (3) Modelado implícito, (4) Descomposición, (5) Programación por objetivos, (6) Generación de conjuntos de trabajo, (7) Solución basada en programación lineal, (8) Construcción/mejora, (9) Metaheurísticas y (10) Otros métodos.

2.1.2 Métodos de resolución propuestos

Por supuesto, al existir tanta diversidad en la formulación del problema, el número de soluciones que se han ido aportando durante su investigación es exponencial. Cada autor propone una variación u otra en función de la complejidad, las restricciones, la función objetivo y/o los recursos disponibles.

Para empezar, Tien y Kamiyama (1982) proponen un marco de resolución del problema que lo subdivide en cinco pasos que son los más habituales en este tipo de soluciones:

1. Cálculo de los requerimientos temporales de mano de obra en cada uno de los periodos.
2. Cálculo de los requerimientos de la mano de obra totales para el horizonte de planificación.
3. Diseño de periodos ociosos a lo largo del horizonte de planificación.
4. Desarrollo de los horarios de tiempo ocioso y de trabajo de cada trabajador.
5. Asignación de un horario de turnos para cada empleado.

Estos pasos no tienen por qué seguirse en orden, sino que conforman etapas interdependientes que normalmente se cierran en bucles de retroalimentación cuya ordenación depende sobre todo en el método usado para resolver el problema. Además, los autores reconocen entre ellas dos grupos distintos: las primeras dos responden al problema de asignación, a los requerimientos de la empresa; y las tres últimas son dependientes de las preferencias de los empleados y las regulaciones estatales o locales.

Otros autores también han contribuido con sus propias estructuras de resolución, como es el caso de Ernst et al. (2004), que se enfocaron en el concepto de línea de trabajo definido por la secuencia de días de trabajo y ociosos de cada trabajador a lo largo del horizonte de planificación; y defendieron una división no en cinco sino en seis etapas: (1) modelado de demanda, (2) programación de días libres, (3) programación de turnos, (4) construcción de línea de trabajo, (5) asignación de tareas y (6) asignación de personal.

Asimismo, los autores revisan en su trabajo las distintas técnicas que han sido utilizadas en la creación de los cuadros de turnos de los empleados, dividiéndolas en 4 grupos: (1) Modelado de la demanda, (2) Enfoques de inteligencia artificial (teoría de conjuntos difusos, sistemas de búsqueda y expertos), (3) Programación de restricciones, (4) Metaheurísticas y (5) Enfoques de programación matemática.

La más completa de todas las clasificaciones aportadas en esta revisión de la literatura es la propuesta por Van den Bergh et al. (2013), que recogen en su artículo una serie de temáticas alrededor de las cuales enmarcar la mayoría de las investigaciones publicadas desde el año 2004. En su trabajo, los autores hacen un análisis de más de 300 problemas de planificación de personal, identificando una amplia variedad de métodos de resolución, de entre los cuales los más repetidos son la programación matemática y la metaheurística.

2.1.2.1 Programación matemática

Para poder resolver el problema que nos atañe mediante programación matemática, este ha de ser formulado de forma lineal, entera, o entera mixta. Esta metodología se basa en la formulación de Dantzig (1954), que permite agregar restricciones específicas del problema al modelo. En este caso, el autor modela los descansos de los trabajadores- fuera de los periodos de comida- para considerarlos dentro de la jornada laboral, lo que aumenta el número de incógnitas del modelo original. Así, estas modificaciones impuestas pueden dar lugar a la aparición de problemas de gran escala, con un número de variables y restricciones que alarga enormemente el tiempo de computación.

A pesar de las ventajas que introduce esta formulación, existen otros enfoques que pueden ser utilizados para modelar este tipo de cuestiones, como el propuesto por Aykin (1996, 2000). Éste defendía un modelado implícito para la programación de turnos como alternativa a la formulación de Dantzig, basándose en el trabajo de Bechtold y Jacobs (1990). Por otro lado, destacan Topaloglu y Ozkarahan (2003) que presentaron un modelo general implícito de programación entera para el *tour scheduling problem*. Rong (2010) también se inspira en estos autores en sus dos propuestas: una formulación de una programación entera implícita y otra de

programación binaria implícita con el horizonte de planificación de un mes y la restricción añadida de los fines de semana libres para los empleados.

Otro enfoque interesante es el enfoque *multicommodity* propuesto por Cappanera y Gallo (2004) para el problema de programación de la tripulación de un vuelo. En este enfoque, cada empleado se representa como una mercancía, lo que permite que determinar el horario mensual de cada empleado sea el resultado de procesar una ruta en un grafo modelado convenientemente para ajustarse además a las restricciones laborales y de sindicatos. Así, el problema puede resolverse con un solver entero, ya que en realidad todos los resultados serán binarios

Además, Mesquita et al. (2015) proponen una formulación de flujo *multicommodity* distinta del problema de programación de tripulaciones, donde cada mercancía corresponde a un patrón de días libres. Este enfoque permite una mayor flexibilidad en la programación de personal y puede adaptarse mejor a las necesidades de cada empresa, reduciendo los costes operativos a la par que se balancean las horas de trabajo de cada trabajador.

Una buena compilación de otros trabajos y tipos de resolución la dan Lin y Ying (2014) en su revisión de la literatura, donde ordenan algunos de los principales artículos de la literatura en función de estos métodos exactos: programación lineal (Fowler et al., 2008; Hochbaum y Levin, 2006; Hojati y Patil, 2011); programación de restricciones (Laporte y Pesant, 2004; Qu y He, 2009); programación de objetivos (Azaiez y Al Sharif, 2005; Lin et al., 2012; Topaloglu y Ozkarahan, 2003), programación entera (Eiselt y Marianov, 2008; Eitzen et al., 2004; Seckiner et al., 2007); programación entera mixta (Firat y Hurkens, 2012; Hertz et al., 2010; Yilmaz, 2012); generación de columnas (Al-Yakoob y Sherali, 2008; He y Qu, 2012; Restrepo et al., 2012); programación dinámica (Beliën y Demeulemeester, 2007; Elshafei y Alfares, 2008); y relajación de Lagrange (Bard y Purnomo, 2007; Pot et al., 2008). Aunque hay muchos métodos exactos para resolver el problema de la programación de personal, estos pueden ser muy costosos en términos de recursos computacionales, especialmente para proyectos de tamaño moderado o grande. En algunos casos, estos métodos no pueden encontrar soluciones viables en un tiempo razonable.

2.1.2.2 Metaheurísticas

Las metaheurísticas también se utilizan ampliamente para abordar problemas de planificación de personal. Estas técnicas son generalmente fáciles de aplicar a cada problema específico y devuelven soluciones factibles, a veces muy cercanas al óptimo, en un tiempo limitado. Sin embargo, las metaheurísticas presentan una clara desventaja al no poder demostrar la optimalidad de las soluciones encontradas.

En general, estos algoritmos pueden ser clasificados en dos grupos mayoritarios tal y como describen Katoch et al. (2021):

1. Algoritmos basados en una sola solución, que utilizan una sola solución candidata y hacen una búsqueda local del optimo
2. Algoritmos basados en una población de soluciones, que utilizan varios candidatos durante el proceso de búsqueda. Así, se mantiene la diversidad en la población evitando los óptimos locales. Dentro de estos, pueden distinguirse a su vez dos grupos diferenciados
    a. Algoritmos evolutivos
    b. Algoritmos de inteligencia de enjambre

Se han empleado muchas heurísticas y la mayoría de las metaheurísticas comunes para problemas de planificación de personal, como la búsqueda tabú, la optimización de enjambre de partículas, el algoritmo genético y el recocido simulado. Estas técnicas ofrecen una solución rápida y eficiente para problemas de planificación de personal, aunque no garantizan la solución óptima.

Por un lado, a búsqueda tabú ha sido útil para el caso del *Nurse Rostering Problem* (NRP) como el anteriormente descrito por diversos autores. Burke et al. (1999) presentaron un sistema híbrido desarrollado para un sistema de calendarios de enfermeros a nivel comercial, y también descubrieron que la búsqueda tabú es muy eficiente cuando se trata de resolver el problema a pequeña escala. Dias et al. (2003) construyeron, por su parte, un horario completo que resolvía el NRP mediante la búsqueda tabú en combinación con la planificación genética, y autores como Bellanti et al. (2004) hicieron lo propio.

Por otro, existen numerosos casos de éxito en la aplicación del recocido simulado en la literatura. Lin et al. (2015) lo utilizaron para la construcción del cuadro de horarios de una planta de urgencias con 12 tipos de turno y más de 100 enfermeros, con restricciones duras como la legislación y la política del propio hospital, que se sumaban a las restricciones blandas de preferencias del personal. El resultado obtenido las cumplía todas, incluso en los periodos de alta demanda.

A pesar de todos estos ejemplos, la metaheurística más popular para la resolución de problemas de programación de personal es sin duda el algoritmo genético (GA). Por ello, y a pesar de pertenecer al grupo de las metaheurísticas aquí descrito, dedicaremos el apartado siguiente a explicar su origen, funcionamiento y su utilidad en este tipo de problemas, dando ejemplos de textos clave en los que el GA ha obtenido resultados destacables por su rapidez y su precisión en la búsqueda de un óptimo.

2.1.2.3 Algoritmo Genético (GA)

Las primeras nociones de algoritmos genéticos probablemente procedan del trabajo realizado por Holland en 1975, donde se parte de la premisa de que toda adaptación, aplicada a cualquier concepto- Holland menciona no solo la genética, sino la planificación económica, la teoría de juegos y la inteligencia artificial- necesita de una modificación de alguna o algunas estructuras. Estas modificaciones pueden ser estudiadas hasta aislar los parámetros básicos que conforman las secuencias de adaptación. En este contexto, la adaptación en se define mediante una serie de medidas de rendimiento- aptitud, utilidad, payoff o eficiencia relativa, para cada uno de los campos anteriormente mencionados.

A Holland no dejaba de sorprenderle como, a pesar de nuestros esfuerzos en aumentar el poder de la computación en la resolución de problemas, nada parecía acercarse a la exactitud con que la naturaleza daba respuesta a estos mismos, a través de un mecanismo supuestamente no orquestado de evolución y selección natural. Así que emuló este poder de evolución para eliminar una de las barreras más importantes en el diseño de software: la necesidad de especificar de forma anticipada al algoritmo las características del problema, así como los pasos a seguir para resolverlo (Holland, 1992)

Tal y como se define en Anwar et al. (2022), el algoritmo genético es una metaheurística que consta de un procedimiento iterativo inspirado en la evolución biológica de los organismos para replicarlos en la obtención de soluciones para los problemas de optimización. A pesar de que los detalles de esta evolución no son totalmente conocidos, como explican en su trabajo Sivanandam y Deepa (2008), hay ciertos principios que se han demostrado experimentalmente y sobre los que el GA se basa:

1. La evolución es un proceso que no opera sobre organismos enteros, sino sobre cromosomas individuales
2. La selección natural es un mecanismo que relaciona a cada cromosoma con la eficiencia de la entidad a la que representan, permitiendo que los más eficientes en la adaptación al medio tengan más oportunidades de reproducirse.
3. El proceso evolutivo tiene lugar en la fase de reproducción, que cuenta con varios mecanismos en la naturaleza. Los más comunes son la mutación (los cromosomas de la descendencia son distintos a los de los padres) y la recombinación (que combina los cromosomas de los padres para producir descendencia)

Este tipo de procesos se engloba dentro de la categoría de algoritmos evolutivos, utilizados para resolver problemas que no tienen aún una solución eficiente bien definida, como en problemas de optimización o modelados y simulaciones en los que se usan funciones aleatorias (Alam et al., 2020).

Las soluciones generadas a través de este método se conocen como individuos, y el conjunto de todas ellas, como población. Todas las soluciones candidatas tienen una serie de características- genes o fenotipos- que pueden alterarse y evolucionar. El método general que sigue el algoritmo para esta fabricación de individuos crea una población inicial de forma aleatoria o con ayuda de alguna heurística de construcción. La medida de la calidad de las soluciones al problema considerado se hace a través de una función de evaluación, que luego el algoritmo utiliza como información cuantitativa en la búsqueda de soluciones.

Dentro de las técnicas evolutivas, el GA representa el grupo más extendido en su aplicación, dado que se apoyan en el uso de selección, recombinación y mutación, y el reemplazo se hace en las nuevas generaciones de individuos. Así, el procedimiento crea sucesivas generaciones de individuos cada vez más adecuados mediante

la aplicación de operaciones simples y guiado solo por el valor de la función de evaluación asociado a cada individuo de la población. Esta medida los ordena dependiendo de su idoneidad relativa para el problema que este siendo resuelto, que coincide a su vez con la función de evaluación que asigna estos valores (Sivanandam y Deepa, 2008).

Otra gran contribución de Holland al campo de la GA fue la adaptación del concepto de inversión en genética- cada posición especifica de un cromosoma tiene dos posibles alelos, 0 o 1. El funcionamiento de cada uno de los operadores en el caso de este algoritmo es el siguiente: la selección escoge los mejores individuos en base a su valor en la función de evaluación. La recombinación escoge una posición aleatoria e intercambia las subsecuencias entre cromosomas para crear nuevas generaciones. Por último, la mutación modifica genes específicos de cromosomas en base a una medida de probabilidad (Katoch et al., 2021).

Así, el funcionamiento del GA clásico es, esquemáticamente, el siguiente:

1. Se inicializa una población de n cromosomas de forma aleatoria
2. Se utiliza la función de evaluación para evaluar la calidad de cada uno de ellos y se registra.
3. Se emplean los operadores anteriormente descritos para evolucionar a la población: dos o más individuos son seleccionados en función de su valor en la función de evaluación. Se aplica la recombinación de uno o varios puntos de cruce entre ellos. Por último, se aplica el operador de mutación con su correspondiente probabilidad al descendiente surgido de las anteriores operaciones.
4. Los nuevos descendientes se guardan en la nueva población y se inicia el reemplazo, que puede hacerse de forma elitista, generacional, estacionario…
5. Se aplica el algoritmo durante un numero de iteraciones, tiempo de computación máximo o hasta alcanzar una solución satisfactoria.

Como el algoritmo genético varía el proceso de búsqueda de forma dinámica mediante las probabilidades de recombinación y mutación, puede modificar los genes codificados y evaluar múltiples individuos dando varias soluciones óptimas, se dice que tiene una mejor capacidad de búsqueda global (Katoch et Al, 2021).

Así, según define el propio Holland, los algoritmos genéticos son perfectos para problemas complejos, y con gran abanico de soluciones posibles, debido a que actúan como una red que se tiende sobre el espacio de estas soluciones posibles, evaluando múltiples puntos de este espacio a la vez, pero centrándose siempre en aquellos que tienen mayor potencial (Holland, 1992). Esto es posible gracias a los operadores anteriormente descritos: la selección y a recombinación aseguran que el algoritmo efectivamente se mantenga en las zonas de mayor calidad del espacio de soluciones, que tendrán más posibilidades de reproducirse y generar descendencia; y a su vez la mutación, aunque no es significativa para el avance hacia el óptimo, sí que previene la uniformidad de la población que provocaría un estancamiento del mecanismo de evolución. Así, con una población significativamente menor al dominio del problema, se puede evaluar de forma efectiva el mismo a través del GA.

Sin embargo, podría parecer que, a través de la recombinación, la descendencia de dos individuos en una región de alta calidad pudiera pasar a otra de menor, provocando un atraso en el desempeño del algoritmo. La probabilidad de que esto ocurra depende de la distancia entre los 0 y 1 que definan esa región concreta- la descendencia de un individuo del tipo [1,0, *, *, *, *] podría estar fuera de la región de su progenitor solo si la recombinación ocurre a partir del segundo gen, lo que solo supone una probabilidad de 1/5 en un individuo con 6 genes. A la vez, la recombinación da respuesta al clásico problema en sistemas adaptativos sobre el coste de explotación frente al de exploración: al encontrar una buena estrategia de evaluación, el algoritmo podría "acomodarse" en ella, reduciendo así las probabilidades de encontrar una todavía mejor. La exploración supone una degradación del rendimiento, penalizado por la función de evaluación, confirmando o desestimando las anteriores hipótesis del algoritmo sobre la calidad de una región (Holland, 1992).

Desde su concepción en 1975, el algoritmo genético ha ido evolucionando y dando lugar a algunas variantes. Estudiaremos ahora las clasificaciones que han derivado de estas tal y como describen Sivanandam y Deepa (2008).

Clasificación del algoritmo genético

a. Algoritmo genético simple (SGA)

Se trata del algoritmo base explicado anteriormente, cuyo uso es opimo y eficiente para problemas donde el espacio de búsqueda es amplio, complejo o difícil de entender, no hay análisis matemático disponible y los métodos de búsqueda tradicionales no arrojan soluciones

b. Algoritmo genético paralelo (PGA)

Se trata de varios SGA ejecutándose de forma paralela, como en el caso de problemas de *job shop scheduling,* donde se reduce el tiempo de ejecución para problemas con grandes espacios de búsqueda. Este procedimiento se puede hacer de forma paralela en función de los siguientes elementos: la forma de aplicación de la función de evaluación y de la mutación; y la forma de aplicación local o global de la selección. La forma más sencilla de paralelizar el SGA es con varias copias del GA que inicialicen y evolucionen distintas poblaciones de forma independiente, de forma que se evita que todos los SGAS converjan a una solución de calidad pobre. Así, se habla de paralelización maestro-esclavo, si existe un módulo maestro que almacena la población y varios módulos esclavos que evalúan la calidad de los individuos; paralelización de grano fino si la evaluación de la calidad de hace de forma simultánea para cada individuo, pero el resto de las fases ocurre de forma local; o GA distribuido, si los SGA intercambian individuos de forma ocasional.

c. Algoritmo genético hibrido (HGA)

Este tipo de algoritmos combinan esta metaheurística con algún otro método de optimización, mejorando su rendimiento global. Esto bien puede ser mediante el uso de la heurística para inicializar la población o sobre los descendientes (Sivanandam y Deepa, 2008) , como mediante la aplicación de la búsqueda local (Gharsalli, 2022; Oh et al., 2004).

Así, la evolución de esta técnica permite seguir mejorando su eficiencia, favoreciendo que pueda ser utilizado en más áreas de optimización. Si ya se ha estudiado la generalidad del tipo de casuísticas en las que el GA es un buen método de resolución de problemas, en adelante se presentan algunos escenarios particulares en los que este algoritmo ha sido aplicado con éxito, dando lugar a soluciones de alta calidad de forma eficaz.

Aplicaciones del GA

Basándonos en la revisión que proporcionan en su trabajo Lambora et al. (2019), podemos hacer un barrido por los ejemplos más significativos donde el GA es relevante en la resolución de problemas.

El primero de ellos es el problema de ruta mínima, donde cada gen de un cromosoma puede ser codificado como un nodo, y que obtienen resultados idénticos a los del algoritmo de Dijkstra.

Por otro lado, y fuera de los clásicos problemas de optimización, el GA ha supuesto un enorme avance en el campo del procesado de imágenes, donde es la técnica de mayor éxito en la búsqueda de soluciones al ser este un problema con un amplio dominio de soluciones posibles. Se resuelven múltiples variables como el contraste, el realzado de detalles o filtros avanzados y parámetros deformables de forma eficaz mediante este algoritmo. Los resultados obtenidos dependen de un buen patrón de codificación de los cromosomas y la función de evaluación.

Esto tiene también su fundamento en el trabajo de Cavicchio (1970), donde se aplica un GA al diseño de una serie de detectores para una máquina de reconocimiento de patrones. Esta, que funcionaba de forma que, de una malla de 25x25 – 625 pixeles binarios, que podían ser claros u oscuros-, se escogían detectores que eran subconjuntos de la malla. Por tanto, la elección de estos detectores resultaba clave para el correcto funcionamiento de la máquina de reconocimiento, ya que debía de ser un conjunto lo suficientemente significativo, de entre un espacio de búsqueda bastante extenso. El GA resultaba perfecto para este caso. Así, en los actuales problemas en los que se requiere un método efectivo para el reconocimiento de patrones-cualesquiera que sea, puede utilizarse una variación del GA adaptada a las necesidades de este.

Además de en las anteriores, el algoritmo genético ha sido utilizado en campos como el *data mining*- Tiwari et al (2019), Xin Li et al. (2009), Gopalan et al. (2006), Tang & Lu (2007) o la cadena de suministros – Naso et al. (2007), Lau et Al (2009), Radhakrishnan et al. (2009).

Por supuesto, también hay numerosos ejemplos de cómo el GA se ha utilizado en la resolución de problemas de

planificación de personal, como el DGA de Easton y Mansour (1999) y el HGA de Nordström y Tufekci (1994), centrado en minimizar los costes de mantenimiento en proyectos con actividades independientes cuando estos se encuentran fuera del lugar en que se desarrolla el proyecto (como en una película, no se trata únicamente de periodos de descanso) y que arrojo resultados reseñables frente a los óptimos conocidos.

Otros ejemplos pueden encontrarse en Cowling et al. (2002), donde un hiper- GA es aplicado al problema de crear un horario para una serie de cursos distribuidos geográficamente en el periodo de varias semanas para un conjunto de entrenadores geográficamente distribuidos; o en Tsai y Li (2009), que aplica el GA en un modelo en dos fases para el problema del horario de las enfermeras de un hospital. En la primera fase, el GA busca discordancias con normativas gubernamentales, políticas de la empresa o injusticias en los horarios. Una vez creado un primer modelo, el GA vuelve a actuar sobre él, optimizándolo.

Así, podemos concluir que, como se introducía al principio de esta sección, el Algoritmo genético tiene gran relevancia en el campo de estudio de la resolución de problemas de optimización en general, puesto que es capaz de aportar soluciones de alta calidad utilizando un procedimiento iterativo sencillo que evalúa un gran número de candidatas a la vez. Asimismo, se demuestra a través de la literatura que la hipótesis de que este tipo de problemas pueda ser resulto a través de este procedimiento es verídica, y es por tanto trivial que sea el método elegido en la resolución del modelo que se presentará más tarde.

2.1.2.4 Otros métodos de resolución

Por último, existen enfoques híbridos que combinan diferentes métodos para abordar problemas de planificación de personal. Por ejemplo, Turhan y Bilgen (2020) proponen un método híbrido que combina heurísticas basadas en programación entera mixta con recocido simulado para resolver el problema de planificación de enfermeros. Este enfoque combina las ventajas de dos técnicas diferentes para obtener una solución más eficiente y precisa.

En conclusión, existen diferentes enfoques para abordar problemas de planificación de personal, desde la programación matemática hasta las metaheurísticas y los enfoques híbridos. Cada enfoque tiene sus ventajas y desventajas, y la elección del enfoque adecuado dependerá de las necesidades específicas de cada empresa y del problema en cuestión.

2.1.3 Áreas de aplicación

Toda esta variedad de metodologías de resolución se ha dado a lo largo de los años por el enorme número de aplicaciones que este conjunto de problemas tiene dentro y fuera del marco teórico. Desde su origen, que puede ser atribuido al trabajo de Edie (1954) acerca de las ya mencionadas cabinas de peaje, los distintos métodos de planificación han sido empleados en sistemas de transporte, servicios de emergencia- policía, bomberos- y otras industrias relacionadas con el servicio como hoteles o restaurantes.

Tal y como apuntan Ernst et al. (2004), Aggarwal (1982) hace una revisión enfocada directamente a las aplicaciones de problemas de planificación de personal y de vehículos, detallando los objetivos, restricciones y metodologías para cada una de las áreas de aplicación. En esta subsección exploraremos cada una de estas áreas de aplicación, detallando los problemas e investigaciones clave en la literatura existente que intentar proveer soluciones.

2.1.3.1 Sistemas de transporte

La planificación de personal dentro del mundo del transporte se conoce como planificación de la tripulación- de aviones, ferrocarriles, autobuses y puertos. Los factores comunes a estos problemas son los siguientes:

1. Coexisten características espaciales y temporales, es decir: cada tarea se caracteriza por un momento y un lugar de inicio y final específicos.
2. Todas las tareas a realizar por los empleados se determinan mediante un horario. Las tareas son los bloques más pequeños, que se descomponen de otros más grandes- vuelos, viajes de tren o de bus. Una tarea puede ser una o varias paradas consecutivas en una línea de tren o bus.

Debido a su escala e impacto económico, la planificación y asignación de tripulaciones aéreas es probablemente la aplicación más grande de la planificación y asignación de personal. Se han dedicado más artículos a metodologías y aplicaciones en esta área que en cualquier otra, ya que además es en ella donde se dan las mayores diferencias entre problemas- los de tripulaciones de aerolíneas, por ejemplo, varían enormemente en función de si son europeas o norteamericanas- dando lugar a un sinfín de dominios en los que puede desarrollarse.

Una técnica popular para resolverlo es la técnica de descomposición, que se divide en tres etapas principales: (*1) emparejamiento de tripulantes con tareas, (2) optimización de emparejamientos de tripulantes* con sus tareas y (3) asignación de horarios a tripulantes. El proceso de generación de estas parejas tripulante-tarea consiste en construir al menos un gran número de las combinaciones posibles para un horario dado. Algunas de ellas serán seleccionadas en el paso dos por tener el menor "coste", y por último estas se secuenciarán para en los cuadros que se asignarán a la tripulación.

Existen numerosos artículos que tratan sobre la resolución de este tipo de problema mediante el método de descomposición, incluyendo algunos como Anbil et al. (1991), Baker (1979), Bodin (1983) o Crainic y Rosseau (1987). También se han desarrollado sistemas comerciales de planificación y asignación de tripulaciones aéreas.

La programación y asignación de tripulaciones también se ha estudiado en sistemas de transporte público, como autobuses y trenes. Se han desarrollado sistemas de apoyo a la toma de decisiones y software para la programación de tripulaciones en estos sistemas. A diferencia de lo que ocurre con los vuelos, estos sistemas de transporte no necesitan de tiempos de rotación y descanso tan largos, y el inicio y el fin de una tarea no tiene por qué ocurrir en el mismo lugar. Algunos ejemplos en la literatura de sistemas empleados para la resolución de estos problemas son: (1) Ball y Roberts (1985) o Carraresi et al. (1995) para líneas de autobús; (2) Anbari (1987) o Caprara et al. (1999) para ferrocarriles; (3) Chu y Chan (1998) para trenes rápidos en Hong Kong; (4) Glen (1984) para astilleros y (5) Sarin y Aggarwal (2001) para camiones.

Otra aplicación muy interesante fue el modelado realizado por Lorenzo-Espejo et al. (2021), en el que estudiaban de forma particular la planificación del personal de un puerto español. En este caso se vuelve de vital importancia que los prácticos del puerto se aseguren de que la entrada y salida se haga de forma puntual, atendiendo además a las exigentes regulaciones relacionadas con la tripulación de barcos y a las posibles penalizaciones que se contemplan por retrasos. Cierto es que este acercamiento podría aplicarse a más de un área- transporte, comercio, industria. De hecho, aunque se trata de un sistema de transporte como tal, las características del problema implican que la ubicación de inicio y final del transporte no sea relevante, dada la larga duración de los movimientos y el relativamente pequeño espacio recorrido.

2.1.3.2 Teleoperadores

A diferencia de los anteriores, este tipo de problemas no presenta una dimensión espacial. En cierto sentido, esto hace que la planificación y la elaboración de horarios se simplifiquen. Sin embargo, a diferencia de las aplicaciones de planificación de tripulaciones en el transporte, no se conoce de antemano la naturaleza exacta y la cantidad de tareas que deben realizarse. Sólo se conoce el patrón de requerimientos de personal para todo el horizonte de planificación, lo que introduce una nueva complicación en su resolución

La variabilidad de los requerimientos de mano de obra en los centros de teleoperadores es tan grande en cuestión de semanas o días, que se hace muy difícil hacer una planificación de la misma de forma que se minimicen los costes y se cubran adecuadamente las necesidades de demanda.

Los trabajos de Mehrotra (1997) y Grossman et al. (1999) proporcionan una revisión de este tipo de problemas y sus resoluciones, que pasan por predecir las necesidades óptimas de personal- para lo cual el modelo preferido es el de puesta en cola de Erlang-C- así como la planificación de personal que minimice el coste, pero que también tenga en cuenta las características y habilidades personales de cada empleado.

En general, la asignación de personal en centros de llamada suele usar simplificaciones de la realidad para encontrar estos óptimos, y se han utilizado enfoques matemáticos y metaheurísticos y soluciones integradas que consideren todos los requerimientos necesarios.

2.1.3.3 Sistemas de salud

El principal énfasis en la planificación de personal en sistemas de salud se pone en la programación de los turnos de enfermería, generalmente en unidades de cuidados intensivos de hospitales. Existen imperativos clínicos y económicos asociados con proporcionar niveles adecuados de personal en las diferentes unidades médicas de un hospital, que deben garantizar la presencia de enfermeros cualificadas para cubrir la demanda derivada del número de pacientes en las unidades, al tiempo que se cumplen las regulaciones laborales. Esto incluye distinguir entre personal permanente y temporal, garantizar una distribución justa de los turnos nocturnos y de fin de semana, permitir días libres, e intentar incluir las preferencias individuales de los empleados. En la mayoría de los casos, estos problemas de programación de horarios están sujetos a múltiples restricciones.(Ernst et al., 2004)

En las décadas de 1970 y 1980, se abordaron diferentes formulaciones y técnicas de solución para estos problemas. El objetivo de muchos estudios era proporcionar herramientas de apoyo que redujeran la necesidad de construir manualmente los horarios de enfermeros. Algunos estudios (Maier-Roth y Wolfe, 1973; Norby et al., 1977; Ryan et al., 1975) abordaron el problema de determinar los niveles de personal y habilidades en función del número de pacientes y sus necesidades médicas. Otros adoptaron enfoques de (1) programación matemática (Warner, 1976; Warner y Prawda, 1972), (2) *branch and bound* (Trivedi, 1974) o (3) programación por objetivos (Ozkarahan, 1991).

En la década de 1990, se lograron avances adicionales (Jaumard et al., 1998; Millar et al. 1998) mediante la aplicación de programación lineal y/o mixta, y técnicas de optimización de redes para desarrollar horarios de enfermeros. También se utilizaron métodos de programación de restricciones (Cheng et al., 1997 y Weil et al., 1995). Estos métodos se aplicaron a problemas que involucraban horarios cíclicos y no cíclicos, y generalmente incluían reglas de horarios específicas de un hospital en particular, lo que podría requerir una reformulación sustancial para su uso en otro hospital.

También se han propuesto enfoques que combinan técnicas heurísticas para abordar los problemas más complejos, como el modelo de simulación complementado con métodos de inteligencia artificial propuesto por Nooriafshar (1995); el recocido simulado de Isken y Hancock (1991) o los modelos estocásticos de Siferd y Benton (1994).

Además, se han estudiado otros aspectos de los sistemas de programación de horarios en el ámbito de la salud. Se han desarrollado modelos para generar horarios de enfermeros que brindan atención domiciliaria y en clínicas regionales, donde los desplazamientos entre diferentes ubicaciones son un factor importante. En estos casos, vuelve a tener relevancia el modelo de puesta en cola para gestionar las llegadas de llamadas de citas en ambulatorios y hospitales (Agnihothri y Taylor, 1991).

En resumen, la programación de horarios en los sistemas de salud, particularmente en la planificación de horarios de enfermeros en unidades hospitalarias, implica desafíos clínicos y económicos. Los horarios deben cumplir con estrictas restricciones debido a las regulaciones laborales y la necesidad imperativa de suplir la demanda en sectores de especial necesidad. A lo largo de los años, se han aplicado diversas técnicas y enfoques, como la programación matemática, la búsqueda tabú, la simulación y los algoritmos genéticos, para abordar estos problemas de programación de horarios. Además, se han explorado aspectos específicos, como la incorporación de la capacitación del personal, la consideración de la gravedad de los pacientes y la gestión eficiente de los recursos en entornos de atención médica complejos. Estas investigaciones continúan buscando soluciones efectivas y eficientes para la elaboración de horarios en el ámbito de la salud.(Ernst et al., 2004)

2.1.3.4 Servicios de protección y emergencia

Un aspecto importante en la asignación de personal en servicios de policía, ambulancia, bomberos y seguridad es la necesidad de cumplir con los estándares de servicio esperados. Estos estándares pueden estar especificados en términos de tiempos de respuesta para atender incidentes o a capacidad de desplegar un número específico de profesionales debidamente capacitados para diferentes tipos de incidentes, entre otros. Además, debido a la naturaleza de las tareas, la mayoría de los servicios de emergencia tienen regulaciones muy estrictas que especifican patrones aceptables de turnos de trabajo.

La frecuencia de los incidentes que requieren la intervención de los servicios de emergencia suele variar en diferentes momentos del día, semana e incluso estación del año. Por ejemplo, puede haber una mayor demanda de ambulancias y oficiales de policía en ciertas áreas los viernes y sábados por la noche, o en áreas turísticas durante períodos vacacionales. Dado que es necesario cumplir con los estándares de servicio, los cambios en la

frecuencia de los incidentes conllevarán cambios en el número de personal necesario para proporcionar los niveles de cobertura requeridos.

En este caso, hay estudios que abordan el problema de manera que se minimice la falta de cobertura (Taylor y Huxley, 1989). Los turnos asignados pueden tener cualquier hora de entrada y salida, pero los patrones de estos sí que están específicamente definidos (4 días/10 horas, 5 días/8 horas, con 3 días consecutivos de descanso). Para obtener las asignaciones, los autores proponen una heurística de búsqueda local con restricciones tabú, comenzando con el horario actual como solución inicial, pero admitiendo que serán necesarios algunos ajustes manuales y optimizaciones adicionales en las soluciones intermedias y finales.

La asignación de horarios para oficiales de ambulancia, de acuerdo con una demanda de mano de obra de diferentes clasificaciones para turnos específicos, se aborda en Ernst et al. (1999). Se construyen horarios a lo largo de un año a partir de patrones permitidos de turnos diurnos, nocturnos y de descanso, junto con períodos de vacaciones anuales. Estos patrones de turno se definen como *stints* y se presentan ejemplos como DDNN y OOO, donde D representa un turno diurno de 10 horas, N un turno nocturno de 14 horas y O un turno de descanso. Se utiliza una matriz de transición para especificar qué *stints* pueden seguir a un *stint* dado y para establecer las preferencias relativas para diferentes sucesiones de *stints*. El método de solución utiliza un algoritmo ruta mínima para generar horarios que se clasifican según el grado en que cubren demanda y en que distribuyen la carga de trabajo, entre otros criterios.

El enfoque de Sinuany-Stern y Teomi (1986) se centra en el problema de desarrollar horarios para guardias de seguridad, que se formula como una programación entera multiobjetivo. Los diferentes objetivos implican minimizar la falta de cobertura de la demanda y satisfacer las preferencias de los trabajadores. Hay una combinación de guardias a tiempo completo y a tiempo parcial. Las restricciones especifican los patrones permitidos de turnos de trabajo y libres para los guardias. Dado que el problema resultante es grande, el artículo discute una combinación de una técnica de ramificación y acotación y una heurística para resolver el problema.

En resumen, la asignación de personal en servicios de emergencia como policía, ambulancia, bomberos y seguridad requiere cumplir con los estándares de servicio y responder a cambios en la frecuencia de los incidentes. Se utilizan enfoques heurísticos, técnicas de optimización y programas enteros para diseñar horarios que minimicen la falta de cobertura, satisfagan las preferencias de los empleados y se ajusten a los patrones de turnos y regulaciones específicas de cada servicio.

2.1.3.5 Administración pública y otros servicios

Las administraciones gubernamentales a todos los niveles (local, estatal y nacional) manejan una gran cantidad de servicios intensivos en cuanto a mano de obra. La optimización de la programación del personal para estos trabajos brinda una oportunidad significativa para mejorar el servicio al ciudadano y, al mismo tiempo, reducir los costes. Sin embargo, estas áreas de aplicación también presentan desafíos únicos, ya que las condiciones laborales de este sector tienden a ser más generosas que las del sector privado y pueden restringir severamente la flexibilidad de los horarios. A continuación, se muestra una indicación de la amplitud de las aplicaciones en esta área.

El problema de cabinas de peaje de Edie (1954) que ya se ha discutido en esta revisión es el ejemplo más conocido dentro de esta área.

En Mould (1996) se presenta un estudio de caso que trata sobre el procesado de reclamaciones que pueden ocurrir en la administración (como beneficios de seguridad social, reclamación de multas…) o en compañías privadas que gestionan reclamaciones de seguros. Se describe un sistema de modelado de demanda que permite a la administración lidiar mejor con las grandes fluctuaciones en la carga de trabajo.

Otro caso relevante es el de la planificación de empleados del servicio postal, que se estudia en Ritzman et al. (1976). Los principales problemas surgen aquí de la gran escala de estos problemas y la variedad de habilidades personales (Feiring, 1993) que han de ajustarse a un gran número de puestos distintos. (Liang y Buclatin, 1988)

También se conocen aplicaciones en el contexto de dotación de personal de las universidades, tanto de profesorado como de secretaría, biblioteca (Ashley, 1995) o turnos de vigilancia de exámenes (Awad y Chinneck, 1998). Ambos cuentan con complicaciones derivadas de restricciones de disponibilidad, tanto del alumnado como de los empleados.

La implementación eficiente de recursos en esta área de aplicación es fundamental para mejorar el rendimiento

financiero dadas las tendencias a la privatización del sector de servicios públicos de agua, gas y electricidad por falta de eficiencia. Así, aunque de momento existe poca literatura que trate específicamente este problema, es un área de investigación interesante del que cabe tener en cuenta por su relevancia.

2.1.3.6 Gestión de espacios

En esta área se engloban todo tipo de operaciones que implican completar tareas realizadas en el mismo lugar, con ejemplos como aeropuertos, terminales de carga o recintos deportivos.

El mayor número de aplicaciones publicadas en esta categoría se refiere a la programación del personal relacionado con aeropuertos, incluyendo al personal de aduanas (Mason et al., 1998), de repostaje de aviones (Alvarez-Valdes et al., 1999) y al personal general en tierra, incluyendo manipuladores de equipaje (Dowling et al., 1997). Todos estos problemas se caracterizan por el hecho de que la demanda de servicios es relativamente conocida, ya que viene dada por los horarios de las aerolíneas. Un tipo de problema de programación de personal algo diferente surge en la programación del personal de mantenimiento de aeronaves, donde la principal dificultad radica en asignar una variedad de tareas que dependen en gran medida de habilidades especiales del personal.

Dos aplicaciones relacionadas con eventos deportivos incluyen la asignación de árbitros en la Liga de Béisbol Estadounidense (Evans, 1988) y para partidos de cricket en Inglaterra (Wright, 1991). Sin embargo, estos problemas están más estrechamente relacionados con la creación de horarios que con otros problemas de programación de personal descritos aquí.

2.1.3.7 Hostelería y turismo

El personal representa una proporción significativa de los costes totales en hoteles, complejos turísticos y restaurantes. Los costos del personal y su formación a menudo superan el 30% de los costes de operar un hotel. Una reducción incluso del 1% de estos costes representa un ahorro considerable que se manifiesta en el resultado financiero de las organizaciones.

Normalmente, los hoteles emplean personal con diferentes habilidades como camareros, limpiadores, recepcionistas, contables, encargados de reservas y mantenimiento. Algunas de estas tareas deben ser programadas para realizarse las 24 horas del día, sin conocerse a ciencia cierta la demanda, cuya predicción se basa en datos históricos y, en tiempos recientes, sistemas online de reservas y análisis de afluencia. Además, el contrato de estos empleados suele incluir periodos de formación obligatorios y desarrollo de habilidades múltiples.

Estudios específicos en el área de hostelería son Rocha et al. (2012) y Taghizadehalvandi y Ozturk (2019).

2.1.3.8 Industria

En entornos de producción dinámicos (Aardal, 1987), se deben tomar decisiones para establecer los niveles de producción de muchos elementos diferentes a fin de satisfacer la demanda durante un período determinado y mantener los inventarios en niveles admisibles. Un problema clave asociado con el equilibrio entre la demanda y la oferta es programar los requisitos de mano de obra necesarios para cada período de producción.

La programación de mano de obra y materiales se considera en Faaland (1993) para lograr objetivos estratégicos y tácticos en una empresa de electrónica. Estos son, entre otros, minimizar los costes asociados con la mano de obra, los materiales, el inventario y las penalizaciones por envíos tardíos.

Se desarrolla un sistema de apoyo a la toma de decisiones programar los trabajadores a tiempo parcial en una empresa de periódicos en el trabajo de Gopalakrishnan et al. (1993). Los trabajadores a tiempo parcial realizan la carga y descarga de materiales, el empaquetado y la distribución de paquetes. El sistema aborda dos aspectos: generar el número de trabajadores a tiempo parcial requeridos para cada turno y programarlos según ciertos criterios que optimizar.

En las empresas de construcción ocurre lo mismo que en la hostelería: una gran porción de los costes se emplea en el personal. Se estudia la integración de la programación de tareas del proyecto y la programación del personal en Al-Zubaidi y Christer (1997). Aquí las restricciones temporales vienen en forma de ventanas de tiempo, y los tiempos de inicio de las tareas determinan los requisitos diarios de personal. El objetivo es determinar el personal

necesario en cada día, en función de las tareas a realizar, de manera que se minimicen los costes.

También existe el caso en que estos trabajadores hayan de ser asignados para llevar a cabo varios trabajos en distintas ubicaciones (Azarmi y Abdulhameed, 1995). De nuevo, cada tarea va caracterizada por una ventana de tiempo de inicio y fin o duración. Pueden existir tareas en las que haya restricciones en cuanto a orden de ejecución de las tareas. Los objetivos de esta optimización suelen ser maximizar la cantidad de trabajo realizado y minimizar el tiempo empleado.

3 DESCRIPCIÓN DEL PROBLEMA

A lo largo de esta sección, se aportara una descripción del problema a resolver, que emula aquel que Lorenzo-Espejo et al. (2021) resolvieran en su artículo.

El problema presentado se refiere a la planificación de los descanso de los prácticos del puerto, que son trabajadores especializados y experimentados en las condiciones de este. Su rol consiste en guiar a los barcos a puerto, lo cual puede resultar una tarea difícil debido al tráfico que suele congestionar estas entradas y salidas. Por estas razones, sumadas al hecho de que la disponibilidad de estos profesionales debe estar asegurada en todo momento por las duras penalizaciones económicas que se derivan de retrasos de algún tipo; el crear una planificación que cumpla con la normativa vigente y los convenios de los trabajadores, y que a la vez asegure la satisfacción de todos ellos, es una tarea ardua y complicada si no se cuenta con los medios adecuados.

El problema presentado es similar también al descrito por Wermus and Pope (1994), en que los pilotos requieren que los horarios resultantes cumplan dos objetivos: ser justos y equitativos y contar con periodos largos de descanso para cada trabajador. No consideraremos aquí, como si hacen ellos, el estado de *stand-by*, debido a que no es necesario asumir que haya pilotos disponibles que no estén trabajando.

Sin embargo, a diferencia de en este último caso, el modelo aquí descrito pretende poner el foco en la satisfacción y las preferencias del trabajador, en contraposición a los objetivos tradicionales de reducir la plantilla o el coste total de la misma, al mismo tiempo que se optimiza la satisfacción del cliente. La mayor contribución que esto presenta a la literatura actual es que, de forma similar a lo que lograron Lorenzo-Espejo et al. (2021), permite a los encargados escoger la duración mínima de cada uno de los dos periodos de descanso a los que tendrán derecho los trabajadores según las restricciones del modelo; y la forma de resolverlo mediante metaheurísticas asegura una obtención de resultados más rápida, pero, preferiblemente, igual de fiable.

Así, nuestro problema comprende una población de e prácticos que pueden realizar cualquier tipo de trabajo, para los que están igualmente cualificados, y el horizonte de planificación es de t días. Los horarios resultantes han de cumplir con varias restricciones: legislación vigente, preferencias de los pilotos y demanda de personal.

3.1 Restricciones

Así, nuestro problema comprende una población de *e* prácticos que pueden realizar cualquier tipo de trabajo, para los que están igualmente cualificados, y el horizonte de planificación es de *t* días. Los horarios resultantes han de cumplir con varias restricciones: legislación vigente, preferencias de los pilotos y demanda de personal.

3.1.1 Legislación vigente

Dentro de este apartado, consideraremos dos restricciones: por un lado, no se permite que los pilotos trabajen durante más de – días consecutivos, ya que su labor se considera exigente y puede provocar *burnouts* por la constante exposición al peligro y las enormes consecuencias ambientales o económicas que puede acarrear un fallo durante la jornada laboral. Por tanto, los prácticos no deben ser sobrecargados con trabajo en periodos cortos de tiempo.

Por otro lado, los pilotos están obligados a tener un mínimo de días libres dentro de cada horizonte de planificación, posibilitando que solo puedan trabajar durante MW días en este periodo.

3.1.2 Preferencias de los pilotos

Como se menciona anteriormente, el cuadro resultante ha de estar guiado por las preferencias de los trabajadores. Así, uno de los mayores desafíos es el de asignar estos periodos largos de descanso, para lo que se utilizaran dos políticas diferentes: asignar dos periodos de descanso en cada horizonte de planificación, o incluir uno solo de mayor duración.

Para asegurar que esta asignación es justa, podemos utilizar dos medidas según la política utilizada. Para la primera de ellas, el equilibrio viene dado en la diferencia entre la máxima y mínima carga de trabajo asignada a los pilotos. Además, al ser necesario asignar dos descansos de duración mínima determinada, estos pueden ser modificados según los requerimientos de los prácticos para no coincidir.

Con respecto a la segunda política de asignación, el equilibrio entre trabajadores se logrará reduciendo la diferencia entre el máximo y el mínimo tiempo de descanso acumulado de los trabajadores. Asimismo, la diferencia entre la máxima y mínima carga de trabajo entre dos trabajadores cualesquiera ha de ser menor que un valor determinado, para garantizar la igualdad en términos de carga de trabajo.

3.1.3 Demanda de personal

Además de las anteriores, el problema descrito ha de cumplir con restricciones relacionadas con el cumplimiento de la demanda diaria especificada para el puerto durante cada periodo, de forma que se completen los servicios en tiempo y forma. Como ya se ha comentado anteriormente, esto es crucial para las autoridades portuarias, que han de asumir altos costes por los retrasos que ocasione la espera de los prácticos.

Sin embargo, para reducir los costes de personal, se asigna el número exacto de pilotos requeridos a cada día, de forma que esta demanda no puede ser más que una previsión realizada al principio de cada horizonte de planificación, cuando todos los prácticos han de ser informados de sus horarios.

Los patrones sobre los que se realiza esta planificación vienen dados por el número de trabajadores diario requerido (PD_j), ya que se asume que cada una de las tareas diarias puede ser realizada por el trabajador de guardia. Esto no se repite en otros puertos, donde otras características tienen que ser tomadas en cuenta para determinar la duración de cada servicio a realizar.

3.2 Modelado

3.2.1 Modelado solución exacta propuesta

Como hemos ido indicando a lo largo del documento, la intención de este trabajo es hacer una comparación entre los resultados obtenidos por los ya citados autores Lorenzo-Espejo et al. (2021) para el problema de planificación del personal del puerto que ellos presentan en su artículo.

Así, cabe destacar que ellos utilizan un acercamiento al problema a través de la programación matemática, buscando por lo tanto soluciones siempre factibles y optimas. De esta manera, sus resultados, aunque muy buenos, necesitan de un largo tiempo de computación. El objetivo de este trabajo es proveer métricas que puedan acortar este tiempo de computación al mismo tiempo que ofrecen buenas soluciones al problema entre manos.

Para ello, se parte de los modelos elaborados por los autores, que se presentan a continuación.

3.2.1.1 Modelo 1

Para este primer problema, el modelo buscaba equilibrar la carga de trabajo de los empleados, de forma que tuvieran dos descansos largos de duración mínima D_1^{min} y D_2^{min} para cada tipo de descanso, respectivamente. En la creación del modelo se adaptaron las restricciones creadas por Brunner et al. (2013), con el que la diferencia principal es el objetivo que se busca, que en el caso de estos últimos está enfocado a la reducción de la plantilla- con la intención de minimizar los costes de esta. Sin embargo, y como se comenta a lo largo de este trabajo, la satisfacción del personal está comenzando a tomarse en cuenta en la elaboración de las planificaciones de la plantilla como un factor clave para la producción por todos los beneficios derivados de la flexibilidad frente a las necesidades del personal.

Por ello, este modelo escoge una función objetivo que mide el fitness de las soluciones del problema en función de su justicia para con los trabajadores, donde todos ellos tengan cargas de trabajo similares, si no iguales.

La siguientes entradas son necesarias para la formulación:

- D_1^{min} and D_2^{min} (que deben ser mayores o iguales a 2) son las mínimas duraciones de los descansos de tipos 1 y 2, respectivamente.
- PD_j es la demanda de personal para el día j, que se mide en número de trabajadores.
- MW es el máximo número de días asignados a cada trabajador en un horizonte de planificación (*maximum workload allowed*).
- C_b^{min} es la duración mínima de cualquier descanso planeado. así, si un trabajador tiene asignado descanso en el día j, no pueden serle asignados días de servicio hasta j+C_b^{min}.
- C_w^{max} es el número máximo de días consecutives en los que un trabajador puede estar de servicio.
- t es el número de días del horizonte de planificación.
- T es el vector de días del horizonte de planificación. ($T = \{1, \dots, t\}$).
- e es el número de empleados que componen la plantilla.
- E es el vector de empleados que componen la plantilla.

Las variables utilizadas en este modelo son las siguientes:

$$x_{i,j} = \begin{cases} 1 \text{ si el trabajador } i \text{ esta de servicio en el dia } j \\ 0 \text{ c. c.} \end{cases}$$

$$b_{i,j}^{1} = \begin{cases} 1 \text{ si el trabajador } i \text{ inicia descanso tipo 1 en el dia } j \\ 0 \text{ otherwise} \end{cases}$$

$$b_{i,j}^{2} = \begin{cases} 1 \text{ si el trabajador } i \text{ inicia descanso tipo 2 en el dia } j \\ 0 \text{ otherwise} \end{cases}$$

$$w^{max} = \text{máxima carga de trabajo de los empleados}$$

$$w^{min} = \text{mínima carga de trabajo de los empleados}$$

$$\text{Min } w^{max} - w^{min} \tag{1}$$

s.t.:

$$\sum_{i \in E} x_{i,j} = PD_j;\ \forall j \in T \tag{2}$$

$$\sum_{j \in T} x_{i,j} \leq MW;\ \forall i \in E \tag{3}$$

$$\sum_{n=0}^{D_1^{min}-1} (1 - x_{i,j+n}) \geq D_1^{min} * b_{i,j}^{1};\ \forall i \in E, j \in \{1, \dots, t - D_1^{min} + 1\} \tag{4}$$

$$\sum_{j=1}^{t-D_1^{min}+1} b_{i,j}^{1} = 1;\ \forall i \in E \tag{5}$$

$$\sum_{j=t-D_1^{min}+2}^{t} b_{i,j}^{1} = 0;\ \forall i \in E \tag{6}$$

$$\sum_{n=0}^{D_2^{min}-1} (1 - x_{i,j+n}) \geq D_2^{min} * b_{i,j}^{2};\ \forall i \in E, j \in \{1, \dots, t - D_2^{min} + 1\} \tag{7}$$

$$\sum_{j=1}^{t-D_2^{min}+1} b_{i,j}^2 = 1;\ \forall i \in E \tag{8}$$

$$\sum_{j=t-D_2^{min}+2}^{t} b_{i,j}^2 = 0;\ \forall i \in E \tag{9}$$

$$\sum_{n=0}^{D_2^{min}-1} b_{i,j+n}^1 \leq (1 - b_{i,j}^2);\ \forall i \in E, j \in \{1, \dots, t - D_2^{min} + 1\} \tag{10}$$

$$\sum_{n=0}^{D_1^{min}-1} b_{i,j+n}^2 \leq (1 - b_{i,j}^1);\ \forall i \in E, j \in \{1, \dots, t - D_1^{min} + 1\} \tag{11}$$

$$\sum_{n=0}^{C_w^{max}} x_{i,j+n} \leq C_w^{max};\ \forall i \in E, j \in \{1, \dots, t - C_w^{max}\} \tag{12}$$

$$1 - x_{i,j} - x_{i,j+k+1} + \sum_{n=j+1}^{j+k} x_{i,n} \geq 0;\ \forall i \in E, j \in \{1, \dots, t - C_b^{min}\}, k \in \{1, \dots, C_b^{min} - 1\} \tag{13}$$

$$\sum_{j \in T} x_{i,j} \leq w^{max};\ \forall i \in E \tag{14}$$

$$\sum_{j \in T} x_{i,j} \geq w^{min};\ \forall i \in E \tag{15}$$

$$x_{i,j}, b_{i,j}^1, b_{i,j}^2 \in \{0,1\};\ \forall i \in E, j \in T \tag{16}$$

$$w^{max}, w^{min} \geq 0 \tag{17}$$

Una explicación completa de la formulación de las restricciones puede encontrarse en Lorenzo-Espejo et al. (2021)

3.2.1.2 Modelo 2

El segundo modelo planteado por los autores pretende conseguir un descanso de la máxima duración posible en el periodo de planificación, en lugar de dos de duración determinada, como en el caso anterior. Para ello, se aplica una formulación diferente, usando las siguientes variables:

$$x_{i,j} = \begin{cases} 1 \text{ si el trabajador } i \text{ está de servicio el dia } j \\ 0 \text{ caso contrario} \end{cases}$$

$$r_{i,j,n} = \begin{cases} 1 \text{ si el trabajador } i \text{ ha descansado los } n \text{ dias previos al dia } j \\ 0 \text{ caso contrario} \end{cases}$$

$$b^{max} = \text{maximo descanso acumulado de los trabajadores}$$

$$b^{min} = \text{minimo descanso acumulado de los trabajadores}$$

Además, este modelo necesita entradas adicionales:

- β_1 es el peso asignado al primer término de la función objetivo.
- β_2 es el peso asignado al segundo término de la función objetivo.
- α es la máxima diferencia permitida entre las cargas de trabajo de dos trabajadores cualesquiera (α debe ser mayor que 0 y menor que 1).

$$\text{Max } \beta_1 * \left[\sum_{i \in E} \sum_{j=2}^{t+1} \sum_{n=2}^{j} \left[\left(r_{i,j,n-1} - r_{i,j,n} \right) * (n-1) \right] \right] - \beta_2 * \left(b^{max} - b^{min} \right) \quad (19)$$

s.t.:

$$\sum_{i \in E} x_{i,j} = PD_j;\ \forall j \in T \quad (20)$$

$$\sum_{j \in T} x_{i,j} \leq MW;\ \forall i \in E \quad (21)$$

$$\sum_{n=0}^{C_w^{max}} x_{i,j+n} \leq C_w^{max};\ \forall i \in E, j \in \{1, \dots, t - C_w^{max}\} \quad (22)$$

$$2 * r_{i,j,n} \leq r_{i,j,n-1} + (1 - x_{i,j-n});\ \forall i \in E, j \in \{3, \dots, t+1\}, n \in \{2, \dots, j-1\} \quad (23)$$

$$r_{i,j,n} \geq r_{i,j,n-1} - x_{i,j-n};\ \forall i \in E, j \in \{3, \dots, t+1\}, n \in \{2, \dots, j-1\} \quad (24)$$

$$r_{i,j,1} = 1 - x_{i,j-1};\ \forall i \in E, j \in \{2, \dots, t+1\} \quad (25)$$

$$\sum_{j\in T} x_{i,j} \geq (1-\alpha) * \sum_{j\in T} x_{k,j}\ ;\ \forall i \in E, k \in \{i+1, \dots, e\} \tag{26}$$

$$\sum_{j\in T} x_{i,j} \leq \frac{1}{1-\alpha} * \sum_{j\in T} x_{k,j}\ ;\ \forall i \in E, k \in \{i+1, \dots, e\} \tag{27}$$

$$\sum_{j=2}^{t+1} \sum_{n=2}^{j} \left[\left(r_{i,j,n-1} - r_{i,j,n}\right) * (n-1)\right] \leq b^{max};\ \forall i \in E \tag{28}$$

$$\sum_{j=2}^{t+1} \sum_{n=2}^{j} \left[\left(r_{i,j,n-1} - r_{i,j,n}\right) * (n-1)\right] \geq b^{min};\ \forall i \in E \tag{29}$$

$$1 - x_{i,j} - x_{i,j+k+1} + \sum_{n=j+1}^{j+k} x_{i,n} \geq 0;\ \forall i \in E, j \in \left\{1, \dots, t - C_b^{min}\right\}, k \in \{1, \dots, C_b^{min} - 1\} \tag{30}$$

$$r_{i,j,j} = 0;\ \forall i \in E, j \in \{2, \dots, t+1\} \tag{31}$$

$$x_{i,j} \in \{0,1\};\ \forall i \in E, j \in T \tag{32}$$

$$r_{i,j,n} \in \{0,1\};\ \forall i \in E, j \in \{2, \dots, t+1\}, n \in \{1, \dots, j\} \tag{33}$$

$$b^{max}, b^{min} \geq 0 \tag{34}$$

En este modelo se siguen cumpliendo todas las restricciones exigidas por la legislación, al tiempo que se da énfasis a la satisfacción del trabajador en tanto en cuanto se busca maximizar su descanso al mismo tiempo que se equilibra la carga de trabajo entre los trabajadores. Así, además de no poder superar un número de días de trabajo en el horizonte de planificación (MW), la función objetivo tiene en cuenta que la diferencia entre el descanso más largo y corto entre los trabajadores sea mínima.

Este es, precisamente, el modelo que trataremos de resolver utilizando un algoritmo genético.

3.2.2 Modelado solución no exacta mediante algoritmo genético

Los algoritmos genéticos son, como hemos visto, perfectos para modelar de forma menos compleja este tipo de problemas con muchas restricciones y cuyas soluciones exactas necesiten gran tiempo de computación.

Las soluciones que arroja el algoritmo genético no tienen por qué ser factibles, pero se aproximan de forma rápida al óptimo y, por tanto, si se programa de manera correcta, pueden llegar a proporcionar soluciones optimas en menor tiempo que la programación matemática.

De forma similar a lo que se ha explicado de forma general en los pasos previos, nuestro algoritmo genético seguirá el esquema evolutivo siguiente:

1. Inicialización del tiempo

   $t = 0$

2. Generación de la población inicial

$(P(t))$

3. Evaluación de la población inicial

   $eval(P(t))$

4. Selección del número de iteraciones (n)

   $iter = n$

5. Inicializar un bucle para la evolución, hasta llegar al número de iteraciones fijado.

   $for\ num_{iter} < n$

6. Selección

   $Q = Seleccion\ (P(t))$

7. Crossover

   $Q' = Recombinacion\ (Q)$

8. Mutación

   $Q'' = Recombinacion\ (Q')$

9. Evaluación de población final

   $eval\ (Q'')$

10. Nueva población

    $P(\ t+1) = sustituir\ (P(t), Q'')$

11. Añadir 1 al número de iteraciones

    $n = n + 1$

12. Fin del bucle

Una vez implementado este esqueleto, sin embargo, han de hacerse consideraciones propias del problema para asegurar una buena evolución ya que, si bien la estructura del algoritmo es común, hay parámetros que son únicos de cada solución deseada. Así, como venimos diciendo, el algoritmo es un marco que puede ser modificado para mejorar la eficiencia y la eficacia de cada caso concreto.

A lo largo del capítulo siguiente, describiremos en detalle la adaptación de la función objetivo del modelo anteriormente mencionado, así como la construcción de una población inicial para el modelo que nos atañe.

4 IMPLEMENTACIÓN Y RESULTADOS

Como se ha destacado a lo largo de todo este trabajo, el caso particular de la planificación del personal del puerto resulta especialmente complicado debido a la rigidez de las restricciones que deben imponerse, así como a lo limitado del dominio de soluciones factibles en el que podemos movernos.

En cualquier caso, y como ha quedado plasmado en el capítulo anterior, la formulación de unas ecuaciones que sean capaces de modelar con exactitud la magnitud del problema es una tarea ardua y nada gratificante, ya que el modelo de Lorenzo-Espejo et al. (2021), no llega a alcanzar una solución óptima.

Por todas las consideraciones realizadas con anterioridad, este trabajo busca contribuir a la literatura existente mediante la creación e implementación de un algoritmo genético que resuelva el problema entre manos. Este algoritmo será implementado en Python 3.10, y los códigos utilizados estarán disponibles en el ***Anexo 2***.

4.1 Parámetros de entrada

El problema que se está resolviendo es un caso real de planificación para un puerto español y su plantilla de trabajadores, que para el caso de estudio se compone de 14 trabajadores a organizar durante 14 semanas, o lo que es lo mismo, 98 días. Estas consideraciones son idénticas a las que tuvieran en cuenta Lorenzo-Espejo et al. en la ejecución de sus modelos.

Para cada periodo de planificación, cada piloto no debe superar los 65 días trabajados en total, ni 5 de ellos de forma consecutiva. Además, la duración mínima de cada descanso debe ser de al menos 2 días, lo que implica que no deben programarse descansos de un solo día.

Habiendo estudiado el patrón de la demanda durante un año, se ha generado una pauta para la demanda semanal compuesta de los máximos registrados durante dicho periodo para cada día de la semana, quedando un gráfico como el de la figura 1. Si se asignara ese número de trabajadores durante cada día, la demanda no quedaría insatisfecha durante ninguna jornada del año registrado.

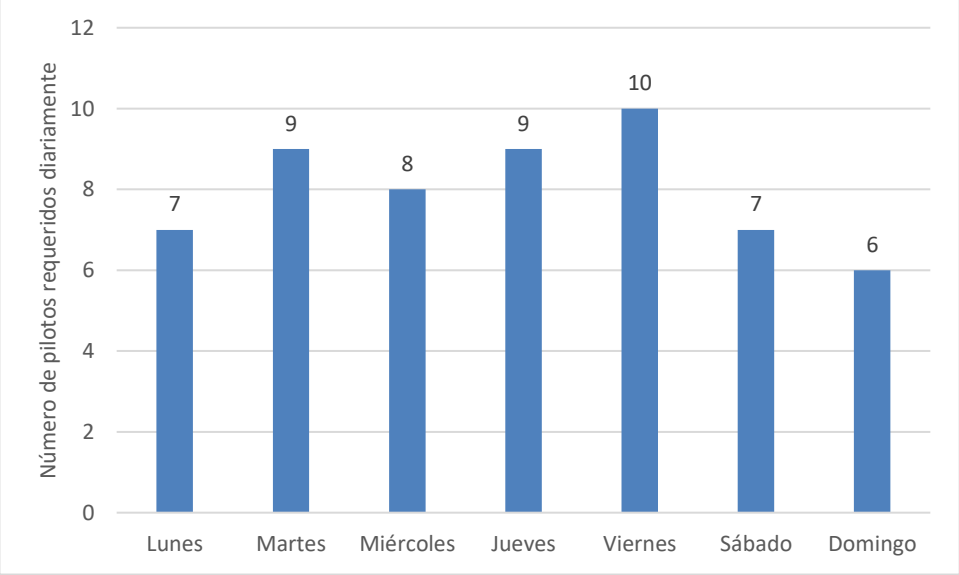


Figura 1: Patrón de demanda semanal. Fuente: Elaboración propia

Es importante reseñar que la demanda viene dada como número de trabajadores por día ya que se considera para este caso de estudio que los prácticos solo dan servicio a un buque por día.

A continuación, se resumen los parámetros necesarios que se aplican a este modelo:

- $t = 98\ días$
- $T = \{1,\ 2,\ \cdots,\ 98\}$
- $E = \{1,\ 2,\ \cdots,\ 14\}$
- $PD_j = \{\ 7,\ 9,\ 8,\ 9,\ 10,\ 7,\ 6\ \}$
- $MW = 63\ días$
- $C_b^{\min} = 2$ días
- $C_w^{\max} = 5\ días$

Con estos parámetros de entrada hemos de construir nuestro algoritmo genético dentro del marco anteriormente mencionado, y lo veremos a lo largo de la siguiente sección.

4.2 Construcción del algoritmo

4.2.1 Definición del individuo solución

El primer paso para la construcción de nuestro algoritmo es diseñar los individuos que van a componer cada población evolucionada, ya que uno de ellos va a acabar siendo la solución al problema planteado.

En nuestro caso, al tratarse de un problema de planificación de personal, nuestro algoritmo ha de devolver precisamente esta planificación a lo largo de todo el horizonte temporal. Según los parámetros que hemos resumido con anterioridad, contamos con una plantilla de 14 trabajadores que deben distribuir descansos y días de trabajo a lo largo de un horizonte de 98 días – 14 semanas.

Por tanto, nuestra solución tendrá la forma de una matriz binaria (M) de 14×98:

- Cada una de las filas de la matriz representa el vector de cada trabajador i durante cada día de trabajo.
- Cada una de las columnas recoge el número de empleados que trabajan en un día j.
- Cada elemento M_{ij} nos da información sobre si el práctico i está trabajando (1) o descansando (0) en el día j.

4.2.2 Inicialización de la población

Por tanto, la población inicial debe estar compuesta por un numero `size_population` de individuos con las características definidas anteriormente.

Para los algoritmos genéticos, esta inicialización puede ser aleatoria o responder a ciertos criterios de adecuación con la solución deseada; y de ello dependerá en alguna medida la rapidez con la que el algoritmo evolucione hacia una solución factible o incluso si el algoritmo es capaz de llegar al óptimo deseado. Por ello, este paso es de gran importancia de cara a la eficacia de nuestro algoritmo, dándose que una buena población inicial puede acelerar la convergencia hacia soluciones de alta calidad, mientras que una mala inicialización puede estancar

el algoritmo o llevarlo por caminos no óptimos.

Tampoco es de menor importancia el tamaño de población elegido, ya que este debe partir de un compromiso entre la exploración y la eficiencia del algoritmo: si bien una población mayor puede evaluar más individuos por cada iteración, lo hará en detrimento del tiempo de computación.

En este apartado exploraremos más en detalle la construcción de nuestra primera población inicial de forma aleatoria, con un tamaño de población predefinido. Esto es solo una base sobre la que iremos construyendo más tarde, de forma que podamos obtener una comparación entre los resultados obtenidos de esta forma y los que pueden llegar a alcanzarse mediante una buena construcción de la población inicial y una exploración del tamaño de esta. Todo ello será explicado en la sección 4.3 de este trabajo.

De momento, como decimos, procedemos a inicializar una población de tamaño estándar de manera aleatoria:

```
#Generacion de una poblacion aleatoria
def generate_population(population_size, num_rows, num_cols):
    population = []
    random.seed(64)
    for _ in range(population_size):
        # Genera una matriz binaria aleatoria
        individual = np.random.randint(2, size=(num_rows, num_cols))
        population.append(individual)
    return population
```

Código 1: construcción de una población inicial aleatoria

Como vemos, la función de generación se inicializa con el tamaño de población, numero de filas y columnas que le demos. Las matrices son combinaciones del tamaño especificado de valores binarios completamente aleatorios, si bien se inicializan con la misma semilla para poder evaluar la evolución del resto de parámetros.

4.2.3 Definición de la función evaluación

La construcción de la función de evaluación resulta asimismo fundamental para el algoritmo genético por muchas razones:

1. Funcionalidad: El buen o mal funcionamiento y evolución de nuestro algoritmo depende de que la evaluación de los individuos se haga conforme a los criterios de optimización elegidos en nuestro modelo. Si la función de evaluación no asigna un fitness adecuado a los individuos, no obtendremos nunca soluciones factibles según nuestros criterios. La correcta implementación de la función objetivo es por tanto crucial para el funcionamiento del algoritmo genético.
2. Eficiencia: Además, nuestra F.O. debe evaluar de la manera mas rápida posible cada individuo, de forma que sea posible realizar un gran número de iteraciones sin que el coste computacional sea demasiado elevado. Esto se consigue mediante la optimización de los bucles, la reducción del número de veces que la función debe leer los datos y la utilización de librerías y funciones específicas dentro de Python, creadas para este fin.

En nuestro caso, la función de evaluación debía ser una adaptación del modelo 2 propuesto en el capítulo anterior por Lorenzo- Espejo et al. (2021), de forma que pueda ser utilizado en nuestro algoritmo y no en programación matemática. Esto fundamentalmente implica que el modelo no tiene restricciones- es decir, que las soluciones generadas pueden ser no factibles en el contexto de nuestro problema. De hecho, para nuestro modelo concreto y como ya hemos comentado con anterioridad, el dominio de las soluciones factibles resulta muy limitado; y por ello observaremos a lo largo del análisis de sensibilidad que un gran número de las soluciones que arroja el algoritmo son, en realidad, no factibles.

La forma en la que conseguimos que nuestra solución evolucione hacia el óptimo es mediante la penalización de las soluciones poco deseables- no factibles. Estas son incluidas en la función de evaluación con un determinado peso, que puede ser variado para orientar las soluciones de las iteraciones hacia valores más interesantes.

Aquí podemos ver el detalle de la construcción de la función de evaluación para el algoritmo genético:

```
def evalua(individuo):
    num_workers, num_days = np.shape (individuo)
    PDj = [7, 9, 8, 9, 10, 7, 6]  # Demanda para cada día de la semana
    from itertools import cycle
    penalties = {
        "descansos_insuficientes": 0,
        "trabajos_consecutivos": 0,
        "limite_dias_trabajados": 0,
        "demanda_diaria": 0
    }
    longest_breaks = []
    dias_trabajados = 0
    workforce_diario = 0
    #Cálculo de dias totales trabajados
    dias_trabajados = np.sum(individuo, axis = 1)
    #Cálculo de los trabajadores diarios
    workforce_diario = np.sum(individuo, axis = 0)
    #Calculo de descansos
    for i in range(num_workers):
        breaks = []
        count = 0
        consecutive_workdays = 0
        for j in range(num_days):
            if individuo[i][j] == 0:
                count += 1
                if consecutive_workdays > 5:
                    penalties["trabajos_consecutivos"] += 1
                consecutive_workdays = 0
            else:
                if count > 0:
                    breaks.append(count)
                count = 0
                consecutive_workdays += 1
        if count > 0:
            breaks.append(count)
        if breaks:
            longest_breaks.append(max(breaks))
            penalties["descansos_insuficientes"] += sum(1 for x in breaks if
x < 2)
        else:
            longest_breaks.append(0)

        if consecutive_workdays > 5:
            penalties["trabajos_consecutivos"] += 1

    if any(x>63 for x in dias_trabajados):
        penalties["limite_dias_trabajados"] +=1

    PDj_cycle = cycle(PDj)
    for i in range(num_days):
        demanda_diaria = next(PDj_cycle)
        if workforce_diario[i] != demanda_diaria:
            penalties["demanda_diaria"] += 2


    objective1 = sum (longest_breaks)
    objective2 = (max (longest_breaks) - min(longest_breaks))
    total_penalty = sum(penalties.values())
    fitness = objective1 - objective2 - total_penalty
    return (fitness)
```

Código 2: Construcción de la función de evaluación.

Como vemos, nuestra función de evaluación sigue los siguientes pasos:

1. Inicialización de parámetros: Se definen todas las variables y parámetros necesarios para el funcionamiento de nuestro algoritmo, como un diccionario de penalizaciones (`penalties`), un vector que almacene los descansos mas largos de cada trabajador (`longest_breaks)` y los valores de los días trabajados (`dias_trabajados`) y la plantilla diaria (`workforce_diario`)
2. El bucle evalúa cada elemento de la matriz individuo y almacena en un vector todos los descansos de cada trabajador (implementando el contador `count)` `y` el número de días consecutivos trabajados en `consecutive_workdays`
3. Dentro del bucle, se implementan las penalizaciones de `descansos_insuficientes` y `trabajos consecutivos`, según unos valores de peso que pueden variar, por cada vez que se incumple una restricción dentro de la matriz.
4. Para cada individuo, el algoritmo escoge el descanso más largo de entre todos los que se hayan registrado en el vector `breaks y` lo guarda en `longest_breaks`.
5. Se comparan los valores de `días_trabajados` con el máximo establecido; así como el `workforce_diario` con la demanda semanal, convertida en ciclo para su comparación
6. Por último, se establecen los objetivos ya descritos por el modelo:

```
objective1 = sum (longest_breaks)
objective2 = (max (longest_breaks) - min(longest_breaks))
total_penalty = sum(penalties.values())
fitness = int(objective1) - int(objective2) - int(total_penalty)
```

Código 3: definición de la función de evaluación

El primer objetivo es la suma de los descansos largos de cada trabajador, el segundo la minimización de las diferencias entre los descansos máximo y mínimo de los trabajadores; y por último se añaden las penalizaciones que nos ayudaran a identificar las soluciones factibles a la hora de realizar las iteraciones.

Estas penalizaciones responden a las restricciones que ya se han comentado en el modelo 2, resumidas en lo siguiente:

- Ningún trabajador puede trabajar más de 63 días durante el horizonte de planificación. $(MW = 63\ días)$
- Ningún trabajador puede trabajar durante más de 5 días consecutivos. $(C_w^{\max} = 5\ días)$
- Ninguno de los descansos de los trabajadores debe tener una duración menor a 2 días. $(C_b^{\min} = 2\ \text{días})$
- La demanda de trabajadores para cada día de la semana debe ser igual al número de trabajadores de guardia ese día PD_j

El resto del código implementado estará incluido en el ***Anexo 2.***

Es necesario señalar también que los pesos que tiene cada uno de estos términos en la función de evaluación también son definitorios para la evolución de las soluciones, ya que es la forma que tenemos de codificar la importancia que tiene cada uno de los objetivos para la solución óptima: si nuestro algoritmo devuelve demasiadas soluciones no factibles, debemos corregir el peso de las penalizaciones en la FO de modo que el algoritmo intente compensarlas en sus evoluciones.

Ese también es un resultado que podremos comparar a lo largo de análisis de sensibilidad.

4.3 Evolución del diseño del algoritmo y resultados obtenidos

Como hemos ido explicando, existen numerosos hiperparámetros abiertos a elección en la construcción de un algoritmo genético, y todos ellos pueden influir en mayor o menos medida en la eficacia y la eficiencia del mismo.

A lo largo de esta sección se pretende ir construyendo un algoritmo genético desde la base del seudocódigo presentado, implementando operadores genéticos genéricos; y evolucionándolo hacia una versión que tenga en cuenta las particularidades del problema concreto a tratar; al mismo tiempo que se recogen los resultados para su comparación.

Hablamos de un análisis de sensibilidad cualitativo porque aquí se recogen los resultados fruto de la experimentación llevada a cabo con la intención de ir mejorando el algoritmo hasta dar con una solución óptima; y por ello las conclusiones serán fruto de la observación y la comparación fácilmente observable, pero no medida. Sin embargo, esto puede servir a modo de guía sobre el impacto que cada uno de los parámetros a variar tenga en los resultados observables. Para recogerlos, haremos uso de las siguientes tablas:

Solución inicial	**Tamaño población**	**Nº de iteraciones**	**Peso penalizaciones**	**Selección inicial**	**Probabilidad *crossover***	**Probabilidad mutación**	**Selección final**	**Elitismo**
Aleatoria	300	1000	1	Torneo n=3	0.8	0.2	Torneo n=3	N

Tabla 1: Ejemplo de parámetros elegidos para la muestra de resultados.

Así, en cada uno de los experimentos iremos modificando los operadores genéticos junto con todos estos hiperparámetros, de forma que el código final represente un algoritmo lo más optimizado posible para las especificaciones del problema, y por tanto proporcione soluciones factibles- e incluso optimas. Hay que tener en cuenta que la combinación perfecta de todos ellos es la que permite que nuestra evolución se haga de manera correcta, eficiente y eficaz. Intentaremos, en los próximos apartados, explicar el impacto de cada uno de los hiperparámetros del algoritmo genético en base a los resultados observados en los repetidos experimentos que se han realizado a este efecto.

Con la intención de no ser redundantes, las próximas secciones agruparán los experimentos realizados en bloques según los operadores genéticos implementados y, dentro de los mismos, se evaluarán los resultados de variar ciertos aspectos como el tamaño de la población, el número de iteraciones o las probabilidades de mutación y entrecruzamiento. En todos ellos se mantienen los torneos de selección inicial y final, que son de 3 individuos. Las combinaciones de parámetros elegidos están recogidas en la siguiente tabla:

Configuración	Solución inicial	Tamaño población	Nº de iteraciones	Peso penalizaciones	Probabilidad crossover	Tipo Crossover	Probabilidad mutación	Tipo de mutación	Elitismo
1	Aleatoria	300	100000	1	0.8	Un solo punto, por filas	0.3	Mantiene demanda	N
2	Aleatoria	300	100000	1	0.9	Un solo punto, por filas	0.3	Mantiene demanda	N
3	Aleatoria	300	100000	1	0.9	Un solo punto, por filas	0.5	Mantiene demanda	N
4	Aleatoria	300	100000	1	0.5	Un solo punto, por filas	0.5	Mantiene demanda	N
5	Aleatoria	300	100000	1	0.5	Un solo punto,	0.5	Mantiene demanda	S

						por filas			
6	Aleatoria	300	100000	1	0.8	Un solo punto, por filas	0.3	Mantiene demanda	S
7	Aleatoria	300	100000	1	0.9	Un solo punto, por filas	0.3	Mantiene demanda	S
8	57% de 1s	300	100000	1	0.9	Un solo punto, por filas	0.3	Mantiene demanda	S
9	Nº de 1s	300	100000	1	0.9	Un solo punto, por filas	0.3	Mantiene demanda	S
10	Nº de 1s	300	100000	1	0.9	Varios puntos, por filas	0.3	Mantiene demanda	S
11	Nº de 1s	300	100000	1	0.9	Varios puntos, por filas	0.3	Mantiene demanda	S
12	Nº de 1s	300	100000	1	0.9	Varios puntos, por filas	0.5	Mantiene demanda	S
13	Nº de 1s	300	300000	1	0.9	Varios puntos, por filas	0.5	Mantiene demanda	S
14	Nº de 1s	100	500000	1	0.9	Varios puntos, por filas	0.5	Mantiene demanda	S
15	Nº de 1s	100	500000	[2,1,1,2]	0.9	Varios puntos, por filas	0.5	Mantiene demanda	S
16	Nº de 1s	100	500000	[2,1,1,2]	0.8	Varios puntos, por filas	0.5	Mantiene demanda	S
17	Nº de 1s	300	100000	[2,1,1,2]	0.8	Varios puntos, por filas	0.3	Mantiene demanda	S
18	Nº de 1s	300	300000	[2,1,1,2]	0.8	Varios puntos, por filas	0.3	Mantiene demanda	S
19	Nº de 1s	300	300000	[2,1,1,3]	0.8	Varios puntos, por filas	0.5	Mantiene demanda	S
20	Nº de 1s	300	300000	[2,1,1,3]	0.8	Varios puntos, por filas	0.3	Mantiene demanda	S
21	Nº de 1s	300	300000	[1,1,1,2]	0.8	Varios puntos, por filas	0.3	Mantiene demanda	S
22	Nº de 1s	300	300000	[1,1,1,2]	0.8	Varios puntos, por filas	0.5	Mantiene demanda	S
23	Nº de 1s	300	300000	[1,1,1,2]	0.8	Varios puntos, por filas	Variable: [0.7-0.1]	Mantiene demanda, prob. Variable	S
24	Nº de 1s	300	300000	[1,1,1,2]	0.8	combinado, filas+columnas	Variable: [0.7-0.1]	Mantiene demanda, prob. Variable	S
25	Nº de 1s	300	300000	[1,1,1,2]	0.8	combinado, filas+columnas	0.5	Mantiene demanda, prob. Variable	S
26	Nº de 1s	300	300000	[1,1,1,2]	0.9	combinado,	Variable:	Mantiene demanda,	S

						filas+columnas	[0.7-0.1]	prob. Variable	
27	Nº de 1s	3000	30000	[1,1,1,2]	0.9	combinado, filas+columnas	Variable: [0.7-0.1]	Mantiene demanda, prob. Variable	S
28	Nº de 1s	3000	300000	[1,1,1,2]	0.9	combinado, filas+columnas	Variable: [0.7-0.1]	Mantiene demanda, prob. Variable	S
29	Nº de 1s	100	300000	[1,1,1,2]	0.9	combinado, filas+columnas	Variable: [0.7-0.1]	Mantiene demanda, prob. Variable	S
30	Nº de 1s	100	300000	[1,1,1,2]	0.6	combinado, filas+columnas	Variable: [0.7-0.1]	Mantiene demanda, prob. Variable	S
31	Aleatoria	100	100000	[1,1,1,1]	0.8	Un solo punto, por filas	0.3	aleatoria	N
32	Nº de 1s	100	500000	[1,1,1,2]	0.6	combinado, filas+columnas	Variable: [0.7-0.1]	Mantiene demanda, prob. Variable	S
33	Nº de 1s	100	300000	[1,1,1,2]	0.6	combinado, filas+columnas	0.5	Mantiene demanda	S
34	Nº de 1s	100	300000	[1,1,1,2]	0.6	combinado, filas+columnas	0.3	Mantiene demanda	S
35	Aleatoria	100	500000	[1,1,1,2]	0.8	Un solo punto, por filas	0.3	aleatoria	N
36	Aleatoria	100	500000	[1,1,1,4]	0.8	Un solo punto, por filas	0.3	aleatoria	N
37	Aleatoria	100	500000	[1,1,1,8]	0.8	Un solo punto, por filas	0.3	aleatoria	S
38	Num 1s	100	5000000	[1,1,1,2]	0.8	combinado, filas+columnas	0.3	Mantiene demanda	S
39	Demanda diaria	100	500000	[1,1,1,2]	0.8	combinado, filas+columnas	Variable: [0.7-0.1]	Mantiene demanda	S
40	Demanda diaria	300	10000	[1,1,1,1]	0.9	Un solo punto, filas	0.3	Mantiene demanda	S
41	Demanda diaria	100	1000000	[1,1,1,2]	0.9	Un solo punto, filas	0.5	Mantiene demanda	S
42	Demanda diaria	100	1000000	[1,1,1,2]	0.9	Un solo punto, filas	Variable: [0.9-0.2]	Mantiene demanda	S
43	Demanda diaria	100	1000000	[2,1,1,1]	0.9	Un solo punto, filas	Variable: [0.9-0.2]	Mantiene demanda	S
44	Demanda diaria	300	500000	[2,2,1,1]	0.9	Un solo punto, filas	Variable: [0.9-0.2]	Mantiene demanda	S

Tabla 2: Resumen de hiperparametros usados en la experimentación

4.3.1 Experimento 1: Crossover de un solo punto por filas, mutación aleatoria

El primer experimento se realiza, como ya hemos dicho, con un esqueleto básico del código, para comprobar la función de evaluación, el tiempo de computación y la evolución de la solución. Este experimento se realiza con los operadores más sencillos que pueden implementarse, y sin mecanismos de elitismo u otras modificaciones en la selección, que de momento se realiza como un torneo simple entre 3 individuos.

Los operadores genéticos utilizados en este caso serán:

- **Selección simple por torneo de 3 individuos elegidos al azar.** El algoritmo escoge 3 individuos cualesquiera dentro de la población y los enfrenta entre sí, quedando elegido el de mayor fitness de cada uno de los 2 torneos como candidatos para la evolución.

```
# Selection - Tournament
        size_tournament = 3
        selec1 =
np.random.permutation(size_population)[0:size_tournament]
        selec2 =
np.random.permutation(size_population)[0:size_tournament]

        value1, ind1 = max((population_evaluated[selec1[i]][0],
selec1[i]) for i in range(size_tournament))
        value2, ind2 = max((population_evaluated[selec2[i]][0],
selec2[i]) for i in range(size_tournament))
```

Código 4: Operadore de selección por torneo a 3

- **Crossover de un solo punto de corte, corte por filas**. Este crossover es el cruce más sencillo de los que existe, tomando los individuos padre y cruzándolos entre sí en un solo punto para generar dos descendientes. Este corte se hace en función de los trabajadores, es decir: cada descendiente tendrá la configuración de la mitad de los trabajadores de cada uno de sus predecesores, fila a fila.

```
if np.random.rand() <= p_cross:
    # Generar un único punto de corte en la mitad de la matriz
    parent1 = pop[ind1]
    parent2 = pop[ind2]
    cross_point = len(parent1) // 2

    descendent1 = np.concatenate((parent1[:, :cross_point], parent2[:,
cross_point:]), axis=1)
    descendent2 = np.concatenate((parent2[:, :cross_point], parent1[:,
cross_point:]), axis=1)
else:
    descendent1 = pop[ind1]
    descendent2 = pop[ind2]
```

Código 5: operador de crossover de un solo punto de corte, por filas

- **Mutación aleatoria:** escoge 3 elementos de la matriz al azar y los muta- si el valor es un 0, lo transforma en un 1, y viceversa.

```
if np.random.rand() <= p_mutation:
    # Elegir un índice de fila y columna al azar para la mutación
    mutation_indices = np.random.randint(0, descendent1.size, size=(3,
2))
    contador_mutacion +=1
    for row_idx, col_idx in mutation_indices:
        row = row_idx % descendent1.shape[0]
        col = col_idx % descendent1.shape[1]
```

```
        if descendent1[row, col] == 1:
            # Cambiar de 1 a 0 en la posición seleccionada
            descendent1[row, col] = 0
```

Código 6: operador de mutación aleatorio

El código completo utilizado en este caso puede consultarse en el ***Anexo 2***.

Con estos parámetros, los resultados obtenidos son los siguientes:

F.O	Tiempo transcurrido(s)	Descansos insuficientes	Trabajos consecutivos	Límite días trabajados	Demanda diaria
-67	-67	105	101	1	0

Tabla 3: Resultados obtenidos con la configuración 31 (Anexo 1)

- **F.O:** Valor de la función de evaluación o función objetivo. Se trata del valor asignado por nuestra función de evaluación al individuo solución final. Se calcula como se ha descrito anteriormente (con los parámetros de los pesos a variar en función del experimento)
- **Tiempo transcurrido:** Lo utilizamos como medida de la eficiencia de nuestro algoritmo. En este caso, 100000 iteraciones de 300 individuos en cada población, proporciona una solución final en 104s
- **Descansos insuficientes:** Es el nombre asignado a la penalización descrita por $(C_b^{\min} = 2 \text{ días})$, por la que ningún trabajador debe tener descansos de menos de 2 días de duración. La función de evaluación asigna un peso (variable) a esta restricción, de modo que cada vez que se incumpla, se añadirá dicho peso al valor de la penalización.
- **Trabajos consecutivos:** De forma similar, esta penalización hace referencia a la restricción definida por $(C_w^{\max} = 5\ días)$ en el modelo anterior. Aquí, se añade su peso cada vez que se incumple en toda la matriz solución.
- **Limite días trabajados:** Modifica la función de evaluación para que penalice cada vez que un trabajador supere el máximo número de días trabajados por horizonte de planificación. $(MW = 63\ días)$
- **Demanda diaria:** este valor aumenta cada vez que la demanda diaria no se corresponde con la dada por el patrón de demanda PD_j.

De esta forma vamos a ir variando los diferentes parámetros de la función objetivo y reservando los resultados para poder establecer comparaciones en función de estos parámetros de solución. Por brevedad, los parámetros escogidos se resumirán en el Anexo 1 y, a partir de ahora, nos referiremos a las distintas configuraciones especificadas en el mismo, del siguiente modo:

Configuración parámetros	F. O	Tiempo transcurrido(s)	Descansos insuficientes	Trabajos consecutivos	Límite días trabajados	Demanda diaria
31	-67	105	101	1	0	94
35	141	506	17	0	0	98
36	56	692	12	0	0	98

Tabla 4: Ejemplo para la visualización, resultados del experimento 1

Los resultados arrojados por estas configuraciones son buenos en cuanto al valor de la FO, ya que tiende a maximizar los descansos largos y por tanto añade un gran número de 0s a las matrices solución generadas. Las iteraciones realizadas en este experimento iban sobre todo enfocadas a la modificación de la función de evaluación para otorgar más peso a la penalización de la demanda, que no ha dado el resultado esperado, ya que, como decimos, el algoritmo no está aún programado específicamente para responder a las necesidades del problema.

Es muy importante anotar, a estas alturas del trabajo, que es necesario entender los datos arrojados en función de los parámetros establecidos, ya que los valores de referencia que podamos tomar dependerán en gran medida de la variación de los hiperparámetos escogidos. Por ejemplo, un número mayor de iteraciones máximas significará un mayor tiempo de computación; así como la variación de los pesos afecta negativamente al valor de la FO, y positivamente al de las penalizaciones. Por claridad a la hora de la presentación de los datos, el número de penalizaciones mostradas será siempre normalizado- es decir, si hemos asignado un peso 2 a la penalización, en la tabla se muestra únicamente el numero individual de ocurrencias de cada una de esas penalizaciones, en lugar de mostrarlas duplicadas.

Para el caso concreto del tiempo de computación, se establecen unos valores de referencia en función de la población y el número de iteraciones:

N iter	Tiempo transcurrido(s)
100000	105-130
300000	300-400
500000	420-700

Tabla 5: Valores de referencia de los tiempos de computación en función del número de iteraciones

El siguiente paso lógico es introducir el elitismo en la selección final para las evoluciones, que se codifica de la siguiente forma:

```
# Elitismo: reemplazar los peores individuos por los mejores de la generación
anterior
    population_evaluated.sort(key=lambda x: x[0], reverse=True)
    population_evaluated[-2:] = [(best_solution_value, best_solution)] * 2
```

Código 7: Implementación de elitismo

Para este experimento se realizó una única iteración, debido a que por interpolación puede suponerse un comportamiento similar al de las anteriores. Los resultados obtenidos:

Configuración parámetros	F. O	Tiempo transcurrido(s)	Descansos insuficientes	Trabajos consecutivos	Límite días trabajados	Demanda diaria
37	-325	552	9	0	0	98

Tabla 6: Resultados del experimento 1, elitismo

4.3.2 Experimento 2: Crossover de un solo punto por filas, mutación que mantiene la demanda diaria

4.3.2.1 Solución inicial aleatoria

Esta vez, con la intención de evitar obtener los resultados que teníamos anteriormente, que optimizaban a costa de ignorar los objetivos de demanda diaria, introduciremos modificaciones en el operador de mutación, de forma que funcione de la siguiente forma.

- **Mutación que mantiene los trabajadores de la jornada:** tres de los elementos M_{ij} serán mutados de forma aleatoria. Si la mutación ha sido de 0 a 1, otro de los elementos de su misma columna pasará de 1 a 0- y viceversa: si el elemento mutado era un 1, se buscará un 0 de la misma columna para mutarlo a un 1. Así, se mantiene el número de trabajadores en cada jornada a la vez que se introduce cierta diversidad en las soluciones.

```
if np.random.rand() <= p_mutation:
            # Elegir un índice de fila y columna al azar para la
mutación
            mutation_indices = np.random.randint(0, descendent1.size,
size=(3, 2))
            contador_mutacion +=1
            for row_idx, col_idx in mutation_indices:
                row = row_idx % descendent1.shape[0]
                col = col_idx % descendent1.shape[1]

                if descendent1[row, col] == 1:
                    # Cambiar de 1 a 0 en la posición seleccionada
                    descendent1[row, col] = 0

                    # Encontrar los índices de elementos 0 en la misma
columna
                    zero_indices = np.where(descendent1[:, col] ==
0)[0]

                    # Elegir al azar un índice de elemento 0 para
cambiar a 1
                    if zero_indices.size > 0:
                        idx_to_change = np.random.choice(zero_indices)
                        descendent1[idx_to_change, col] = 1
```

Código 8: Operador de mutación que mantiene la demanda diaria estable

Después de esta mutación, de forma evidente, se evalúa el fitness de los individuos mutados y, en caso de ser mejores que el mejor valor actual, este se sustituye por el fitness del individuo mutado.

En la siguiente tabla podemos comparar las soluciones que devuelve este experimento, en función de los hiperparametros escogidos y especificados en la tabla del ***Anexo 1:***

Configuración párametros	F.O	Tiempo transcurrido(s)	Descansos insuficientes	Trabajos consecutivos	Límite días trabajados	Demanda diaria
1	-128	104	133	12	0	84
2	-140	108	146	5	0	78
3	-126	110	126	9	0	82
4	-127	111	125	7	0	85
5	-130	122	129	11	0	79
6	-139	123	140	7	0	82
7	-132	131	137	10	0	83

Tabla 7: Resultados del experimento 2

La mayoría de estas configuraciones respondían a la variación de las probabilidades de mutación y crossover y, como podemos observar, en cuanto a resultados finales no existe gran diferencia entre la eficacia de unos u otros. Tampoco este tipo de mutación parece especialmente eficaz a la hora de reducir las penalizaciones por demanda.

Se observa que la probabilidad de mutación tiene gran impacto en la primera evolución del algoritmo pero que, sin embargo, en las iteraciones finales queda estancado. En apartados siguientes podremos ver la construcción de un operador de mutación cuya probabilidad evoluciona con el tiempo, para tratar de paliar este fallo.

Por otro lado, todas estas configuraciones implican una solución inicial no especializada para nuestro problema, ya que los individuos se inicializan de forma aleatoria mediante un `randint` con una semilla especifica.

Probemos ahora qué ocurriría en caso de que esta solución inicial estuviera adaptada a las necesidades de nuestro problema.

4.3.2.2 Solución inicial modificada

Como planteamiento para crear una primera solución desde la que el algoritmo pueda iterar, se propone aquel que procure acercarla lo más posible a las restricciones de demanda, sin ser especialmente especifica de forma que el algoritmo pueda explorar convenientemente el espacio de búsqueda. Así, se conviene crear una primera solución en la que la demanda total del periodo de planificación está cubierta, si bien no distribuida correctamente.

La idea es obtener la demanda semanal, mediante la suma del patrón de la demanda diaria PD_j, y multiplicarla por el número de semanas del horizonte temporal (14). Esto nos da una demanda total para el periodo de 784, de modo que, la inicialización de nuestra matriz debe tener, al menos 784 1s:

```
def generate_population(population_size, num_rows, num_cols):
    population = []
    num_ones_per_individual = 784  # Número deseado de 1s en cada individuo

    for _ in range(population_size):
        individual = np.zeros((num_rows, num_cols), dtype=int)

        # Establecer aleatoriamente los 1s en el individuo
        indices = np.random.choice(num_rows * num_cols,
num_ones_per_individual, replace=False)
        row_indices, col_indices = np.unravel_index(indices, (num_rows,
num_cols))
        individual[row_indices, col_indices] = 1

        population.append(individual)

    return population
```

Código 9: Inicialización de una matriz que cumple el requisito global de demanda

La población inicial está compuesta, pues, de individuos que tienen 784 1s en total. Comprobemos ahora si esta mejora del algoritmo arroja resultados diferentes a los ya analizados:

Configuración parámetros	F. O	Tiempo transcurrido(s)	Descansos insuficientes	Trabajos consecutivos	Límite días trabajados	Demanda diaria
8	-179	120	151	19	0	77
9	-185	125	163	16	0	79
10	-178	115	161	21	1	74

Tabla 8: Resultados experimento 2.1, con solución inicial modificada.

Como puede observarse, las soluciones arrojadas no parecen demasiado prometedoras al introducir esta nueva

modificación. Sin embargo, quizás se deba a que los operadores no están tampoco modificados para favorecer la evolución de forma que se cumpla con las penalizaciones al mismo tiempo que se introduce diversidad en la población…

4.3.3 Experimento 3: Crossover de varios puntos por filas, mutación que mantiene la demanda diaria

Ahora, se procede a modificar el operador de crossover de manera que actúe de forma un poco menos sencilla, de forma que las soluciones evolucionadas cuenten con mayor diversidad y, a la vez, haya más probabilidad de alcanzar el óptimo.

Para formular este operador genético, se idea una función que tome dos individuos cualesquiera de la población y los cruce en `num_points` puntos escogidos al azar. En este caso, el operador copia las filas completas de trabajadores de una matriz padre a otra hijo. El detalle del código es el siguiente:

```
def multi_point_crossover(parent1, parent2, num_points):
    # Asegúrate de que num_points no sea mayor que la longitud de los
cromosomas
    num_points = min(num_points, len(parent1) - 1)

    # Genera num_points puntos de corte distintos
    cut_points = sorted(random.sample(range(1, len(parent1)), num_points))

    child1 = np.copy(parent1)
    child2 = np.copy(parent2)
    last_cut = 0

    for cut in cut_points:
        if (cut - last_cut) % 2 == 0:
            # Si la distancia entre cortes es par, intercambia los segmentos
entre los cortes
            child1[last_cut:cut], child2[last_cut:cut] =
child2[last_cut:cut], child1[last_cut:cut]
        else:
            # Si la distancia entre cortes es impar, intercambia los
segmentos después del corte
            child1[cut], child2[cut] = child2[cut], child1[cut]

        last_cut = cut

    return child1, child2
```

Código 10: Operador de crossover con varios puntos de corte, por filas.

El algoritmo comprueba si los cortes generan conjuntos pares o impares de secciones a copiar, de modo que el primer descendiente reciba los segmentos entre cortes o después de los cortes. En resumen, para un número de cortes igual a 3, los cruces entre dos matrices cualesquiera se producen del siguiente modo:

```
Las matrices iniciales son:
    0     0     1     0     0     0     1     0
    0     1     1     0     1     1     0     1
    0     1     1     1     0     1     1     1
    0     1     0     0     1     1     1     1
    1     0     0     1     1     1     1     1
    1     1     0     0     1     0     0     1

    1     0     1     0     0     0     0     0
    0     0     1     0     1     0     1     1
    1     1     1     1     1     1     0     1
    1     1     0     1     0     1     1     1
    1     0     0     1     0     1     1     0
    1     1     0     1     0     1     1     0

Las matrices cruzadas por filas son:
    0     0     1     0     0     0     1     0
    0     0     1     0     1     0     1     1
    1     1     1     1     1     1     0     1
    0     1     0     0     1     1     1     1
    1     0     0     1     0     1     1     0
    1     1     0     0     1     0     0     1

    1     0     1     0     0     0     0     0
    0     0     1     0     1     0     1     1
    1     1     1     1     1     1     0     1
    1     1     0     1     0     1     1     1
    1     0     0     1     0     1     1     0
    1     1     0     1     0     1     1     0
```

Código 11: Ejemplo de crossover de 3 puntos en matrices

Reflexionemos, ahora, sobre los resultados que arroja el algoritmo incorporando este tipo de crossover más complejo:

Configuración parámetros	F. O	Tiempo transcurrido(s)	Descansos insuficientes	Trabajos consecutivos	Límite días trabajados	Demanda diaria
11	-135	112	182	13	0	85
12	-123	110	189	14	0	90
13	-111	409	160	6	0	88
14	-98	629	174	15	0	91

Tabla 9: Resultados del experimento 3

Como se puede observar, en un principio parecía que la evolución funcionaba de manera eficaz, sin estancarse demasiado. Por eso, decidimos ir aumentando el tiempo de ejecución, con la intención de obtener una solución satisfactoria. Sin embargo, no conseguimos el objetivo ya que, aunque el valor de la función de evaluación sí que mejoraba con el número de iteraciones, al comparar los valores de las penalizaciones- que es, finalmente, lo que hace a una solución factible o no- encontramos que no había diferencias entre introducirlo o no.

Además, aquí empiezan a hacerse modificaciones en el tamaño de la población y las probabilidades de mutación, descubriéndose que un tamaño de 100 resulta muy efectivo para no quedar estancado y para alcanzar una convergencia de forma más rápida. No obstante, para casos concretos seguiremos utilizando el valor de 300 o poblaciones mayores, que han demostrado arrojar resultados significativos para nuestro estudio. Por su parte el operador de mutación juega de nuevo un papel fundamental en las primeras iteraciones, no así durante las últimas, cuando supone más una desaceleración del proceso.

Los resultados que han arrojado este experimento invitan a pensar que quizás la función de evaluación no concede toda la importancia que debería a las penalizaciones. Estudiemos, en el siguiente apartado, la evolución que puede llegar a alcanzarse mediante una variación en los pesos de estas.

4.3.4 Experimento 4: Crossover de varios puntos por filas, mutación que mantiene la demanda diaria, cambios en los pesos.

En función de lo visto en los resultados anteriores, la función de evaluación necesita conceder una mayor importancia a las restricciones de demanda y descansos insuficientes. Por ello, hemos ido asignando pesos diferentes a las penalizaciones a lo largo de este experimento, con la intención de que la evolución hacia FO mayores nos proporcionara también un descenso en el número de penalizaciones en las que se incurre:

Configuración parámetros	F. O	Tiempo transcurrido(s)	Descansos insuficientes	Trabajos consecutivos	Límite días trabajados	Demanda diaria
15	-316	725	178	15	0	87
16	-317	423	187	10	0	93
17	-354	117	131	4	0	85
18	-313	381	197	13	0	90
19	-411	396	119	13	0	85
20	-425	333	127	15	0	82
21	-193	351	125	6	0	85

Tabla 10: Resultados del experimento 4

Para casos en los que hemos aplicado pesos de 2 para la restricción de demanda diaria y la de descansos insuficientes (configuraciones 15,16,17,18), los resultados no muestran una clara mejora en la evolución de las mismas. Al tratar de dar mayor importancia a la penalización diaria (19, 20) solo se observa un descenso en el valor de la FO, sin el esperado descenso en el número de ocurrencias de las penalizaciones.

De nuevo miramos a los operadores genéticos para tratar de solventar estas cuestiones, ya que quizás la mutación, como hemos comentado anteriormente, no este favoreciendo al algoritmo en un estado avanzado de iteraciones.

Por otro lado, además, también podría suponer una mejora para el problema de la penalización de la demanda que, además de intercambiarse los individuos por sus trabajadores, lo hicieran entre días, de forma que se pudieran recoger las estructuras de demanda que sean favorables.

4.3.5 Experimento 5: Crossover combinado de varios puntos, mutación que mantiene la demanda diaria, variable en el tiempo, cambios en los pesos.

Para las cuestiones anteriormente mencionadas, se hace por tanto imperativo implementar dos cambios en nuestro algoritmo:

1. **Operador de mutación variable en el tiempo:** La mutación, al principio del algoritmo genético, es tremendamente útil para favorecer la exploración del espacio de soluciones. Sin embargo, conforme las iteraciones avanzan, quizás demasiada tasa de mutación pueda resultar más perjudicial que beneficiosa. Por ello, implementaremos una lógica que permita al algoritmo ir disminuyendo la probabilidad de mutación en función de la iteración en la que nos encontremos.

```
def prob_mutacion(iteration, max_iterations):
    # Define la probabilidad de mutación inicial y final
```

```
    initial_mutation_prob = 0.7  # Probabilidad de mutación alta al
principio
    final_mutation_prob = 0.1    # Probabilidad de mutación baja al final

    # Calcula la probabilidad de mutación actual en función de la
iteración actual
    mutation_prob = initial_mutation_prob - (initial_mutation_prob -
final_mutation_prob) * (iteration / max_iterations)
    return mutation_prob
```

Código 11: Función de cálculo de la probabilidad de mutación variable con el número de iteraciones

Esta probabilidad de mutación se aplica posteriormente al código dentro del `main`, de modo que:

```
if np.random.rand() <= prob_mutacion(i,n_iteration):
        contador_mutacion += 1
        # Elegir un índice de individuo al azar para la mutación
        mutation_idx = ind1
        descendent1 = mutate_individual(pop[mutation_idx])
        pop[mutation_idx] = descendent1
```

Código 12: Implementación de la mutación con probabilidad variable

Así somos capaces de ser muy eficaces en cuanto al balance entre exploración y explotación de las soluciones, puesto que tenemos la capacidad de modificar esa balanza en distintos momentos del algoritmo.

2. **Operador de crossover que combine el cruce entre filas y el de columnas:** se establecerá que, con una probabilidad dada, si se produce crossover, este será entre filas o entre columnas. Para ello, obtenemos una función que defina el crossover combinado.

```
def combine_crossovers(parent1, parent2, num_points, p3):
    if random.random() < p3:
        # Aplicar el crossover por filas
        return crossover_rows(parent1, parent2, num_points)
    else:
        # Aplicar el crossover por columnas
        return crossover_columns(parent1, parent2, num_points)
```

Código 13: Definición de la función de crossover combinado en función de una probabilidad p3

Y, por otro lado, establecemos el crossover por columnas, de forma idéntica a como hacíamos para filas:

```
def crossover_columns(matrix1, matrix2, cut_points):
    # Calculo del numero de columnas de la matriz
    num_columns = matrix1.shape[1]

    # Generacion de puntos de corte aleatorios
    cut_indices = np.random.choice(num_columns, size=cut_points,
replace=False)

    # Ordenacion de los puntos de corte de forma descendiente
    cut_indices.sort()

    # Inicializar matrices de descendendientes
    child1 = np.zeros_like(matrix1)
    child2 = np.zeros_like(matrix2)

    # Crossover por columnas
    for i in range(num_columns):
        if i in cut_indices:
```

```
            # Crossover en los puntos de corte
            child1[:, i] = matrix2[:, i]
            child2[:, i] = matrix1[:, i]
        else:
            # Copiar las columnas de los padres
            child1[:, i] = matrix1[:, i]
            child2[:, i] = matrix2[:, i]

    return child1, child2
```

Código 14: Operador de crossover por columnas.

Así, podemos refinar aún más la construcción de nuestro algoritmo, con la intención de ir obteniendo resultados cada vez mejores. Revisemos las soluciones arrojadas:

Configuración parámetros	F. O	Tiempo transcurrido(s)	Descansos insuficientes	Trabajos consecutivos	Límite días trabajados	Demanda diaria
22	-189	343	128	8	0	86
23	-192	333	130	5	0	83
24	-200	702	125	8	0	75
25	-191	493	188	12	0	90
26	-198	725	118	6	0	91
27	-248	93	148	18	0	81
28	-230	1877	137	9	0	83
29	-198	1133	202	10	0	86
30	-192	499	114	10	0	88
38	-174	12633	111	4	0	87

Tabla 11: Resultados del experimento 5

Llegamos al final de la experimentación y los resultados obtenidos no son los deseados, ya que, a pesar de haber realizado numerosas variaciones, e introducir operadores y soluciones específicas para nuestro problema, no hemos conseguido que se nos devuelva una solución factible.

Cabe destacar que, en la configuración 38 se ha ejecutado el algoritmo durante un largo periodo de tiempo (unas 3,5h) con la intención de comprobar la convergencia de este en periodos prolongados de tiempo. Se observa que, de manera similar a ejecuciones de órdenes menores realizadas, los resultados convergen a cierto valor y quedan estancados durante muchas iteraciones.

A lo largo de este capítulo, hemos presentado los resultados de la aplicación del algoritmo generado a la planificación real de un caso de personal de puerto. A pesar de las numerosas iteraciones que se han seguido para tratar de encontrar la combinación de hiperparametros que resulte más efectiva a nuestro algoritmo, los resultados obtenidos no han alcanzado nuestras expectativas.

4.3.6 Experimento 6: Solución inicial igual a demanda diaria.

Como llevamos observando resultados bastante alejados de lo deseable para la demanda diaria, partamos de una solución inicial que satisfaga esta demanda desde el principio.

Para ello, obtenemos una función de generación de población que inicializa los individuos de forma aleatoria, pero asegurando que cada día se cumpla la demanda, dada por la suma total de valores de las columnas. La implementación de este operador se detalla en el código siguiente:

```
def generate_population(population_size, num_rows, num_cols, demand_vector):
    population = []

    for _ in range(population_size):
        individual = np.zeros((num_rows, num_cols), dtype=int)

        for col in range(num_cols):
            demand = demand_vector[col % len(demand_vector)]
            indices = random.sample(range(num_rows), demand)
            individual[indices, col] = 1

        population.append(individual)

    return population
```

Código 16: Inicialización de la poblacion con columnas iguales a demanda inicial

Este código, como puede verse, recibe como entradas el tamaño de la población y los individuos deseados, y el vector de demanda al que ha de compararse. Este valor se introducirá en nuestra `main`, del siguiente modo:

```
    PDj_cycle = [7, 9, 8, 9, 10, 7, 6]
    pop = generate_population(size_population, 14, 98, PDj_cycle)
```

Código 17: Implementación de la función generate_population de demanda diaria.

Ahora, analicemos los resultados obtenidos por las siguientes configuraciones:

Configuración párametros	F.O	Tiempo transcurrido(s)	Descansos insuficientes	Trabajos consecutivos	Límite días trabajados	Demanda diaria
39	-119	642	175	9	0	0
40	-117	369	165	15	0	0
41	-100	105	156	20	0	0
42	-243	1479	139	26	0	0
43	-257	114	150	22	0	0
44	-242	328	138	23	0	0

Tabla 12: Resultados del experimento 6

De nuevo y a pesar de nuestros esfuerzos, no conseguimos que la solución converja hacia valores menores de descansos insuficientes, pero sí logramos que las soluciones cumplan con la demanda exigida. Cambiando los valores de los pesos en la FO conseguimos que estos valores, en algunos casos, disminuyan. Sin embargo, de nuevo existe estancamiento en la evolución de las soluciones, aun existiendo nulas restricciones al crossover o la mutación.

4.4 Relación de soluciones obtenidas

Configuración parámetros	F. O	Tiempo transcurrido(s)	Descansos insuficientes	Trabajos consecutivos	Límite días trabajados	Demanda diaria
1	-128	104	133	12	0	84
2	-140	108	146	5	0	78
3	-126	110	126	9	0	82
4	-127	111	125	7	0	85
5	-130	122	129	11	0	79
6	-139	123	140	7	0	82
7	-132	131	137	10	0	83
8	-179	120	151	19	0	77
9	-185	125	163	16	0	79

10	-178	115	161	21	1	74
11	-135	112	182	13	0	85
12	-123	110	189	14	0	90
13	-111	409	160	6	0	88
14	-98	629	174	15	0	91
15	-316	725	178	15	0	87
16	-317	423	187	10	0	93
17	-354	117	131	4	0	85
18	-313	381	197	13	0	90
19	-411	396	119	13	0	85
20	-425	333	127	15	0	82
21	-193	351	125	6	0	85
22	-189	343	128	8	0	86
23	-192	333	130	5	0	83
24	-200	702	125	8	0	75
25	-191	493	188	12	0	90
26	-198	725	118	6	0	91
27	-248	93	148	18	0	81
28	-230	1877	137	9	0	83
29	-198	1133	202	10	0	86
30	-192	499	114	10	0	88
31	-67	105	101	1	0	94
35	141	506	17	0	0	98
36	56	692	12	0	0	98
37	-325	552	9	0	0	98
38	-174	12633	111	4	0	87
39	-119	642	175	9	0	0
40	-117	369	165	15	0	0
41	-100	105	156	20	0	0
42	-243	1479	139	26	0	0
43	-257	114	150	22	0	0
44	-242	328	138	23	0	0

Tabla 13: Soluciones de todos los experimentos

Si quisiéramos hacer un análisis de las soluciones obtenidas, habría que comparar múltiples parámetros, ya que hemos visto que la FO da información incompleta cuando hablamos de la factibilidad de una solución.

Si bien es cierto que, las soluciones que parten de matrices aleatorias, con operadores genéticos aleatorios y no modificados a las necesidades del problema, consiguen unos valores de FO muy altos, no consiguen ajustar la solución a la demanda diaria, que es restricción clave para la viabilidad de la solución.

Por otro lado, la configuración número 10 – inicialización de matriz solución con un cierto número de unos, crossover por filas en varios puntos y mutación aleatoria- arroja un mejor resultado en cuanto al ajuste a la demanda.

Si tenemos en cuenta los valores de la ejecución más larga realizada, en la que la evolución se realiza de forma más controlada, podríamos tender a pensar que un algoritmo menos especifico, como el del experimento 1, obtiene mejores resultados de la función objetivo. Sin embargo, como se ha visto reflejado en esta relación de resultados, esto solo ocurre a costa de incumplir las penalizaciones de demanda, que resultan a estos efectos cruciales para la consecución de una solución factible.

Sin duda, y como venimos diciendo a lo largo de todo el trabajo, el caso de estudio de los prácticos de puerto resulta perfecto para la utilización de técnicas de optimización porque es demasiado complejo para abordar de otro modo. Sin embargo, esto también ha propiciado que un estudio del caso mediante el algoritmo genético se haga casi imposible, disponiendo de tiempo y recursos limitados. Las múltiples restricciones y el uso de una función objetivo poco convencional contribuyen a que esto suceda.

El estancamiento observado en la evolución para ejecuciones muy largas solo puede deberse a que estas restricciones necesitan de operadores más específicos o parámetros muy curados para el problema entre manos, ya que de otro modo se hace imposible que el algoritmo salga de ese óptimo local en el que se encuentra para seguir buscando soluciones mejores que sean capaces de satisfacer las restricciones del espacio de factibilidad del problema.

Además, la solución que se presentaba a través de la formulación matemática ha resultado ser factible, aunque no óptima, si bien es cierto que esta obtención no iba exenta de altos costes computacionales. El algoritmo genético innegablemente evoluciona de manera rápida, pero ha de ser meticulosamente construido para no quedarse estancado en óptimos locales o descartar soluciones prometedoras de la población inicial.

Si utilizamos el gráfico de la mejor solución obtenida mediante programación matemática (Lorenzo-Espejo, 2023):

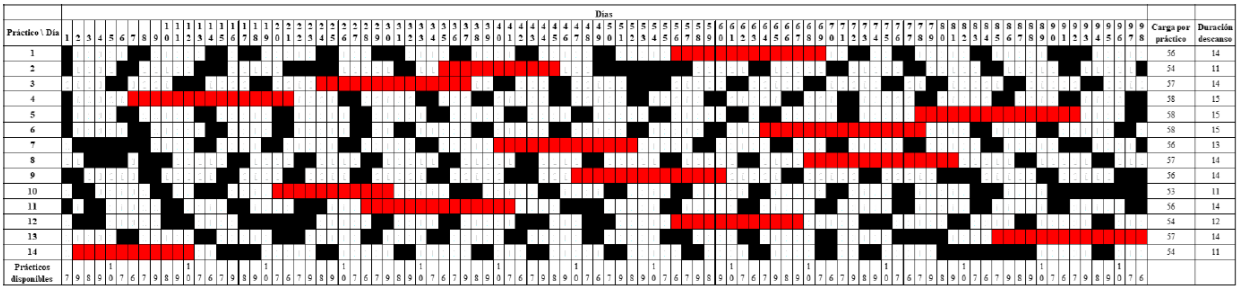


Tabla 14: Solución propuesta para el modelo de programación exacta. Fuente: Lorenzo-Espejo, 2023

Y lo comparamos con nuestra mejor solución, la dada por la configuración 44, resaltada en verde en la ***tabla 12,*** observamos que, si bien la demanda se cumple para ambas soluciones, los descansos que proporciona el método exacto son mucho mas largos para los empleados, y además la carga media de trabajo por empleado es un poco menor a la de nuestra solución. Por su parte, la diferencia entre descansos máximos y mínimos no cambia en absoluto entre ambas soluciones, lo que puede ser un punto a destacar.

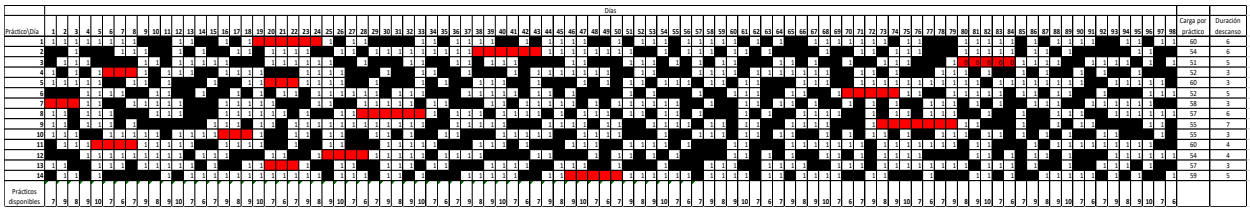
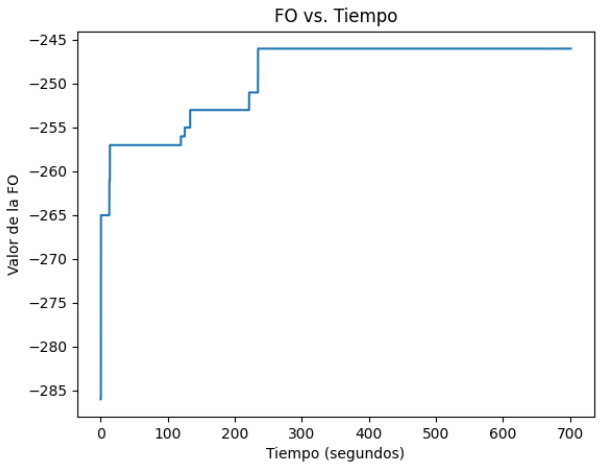
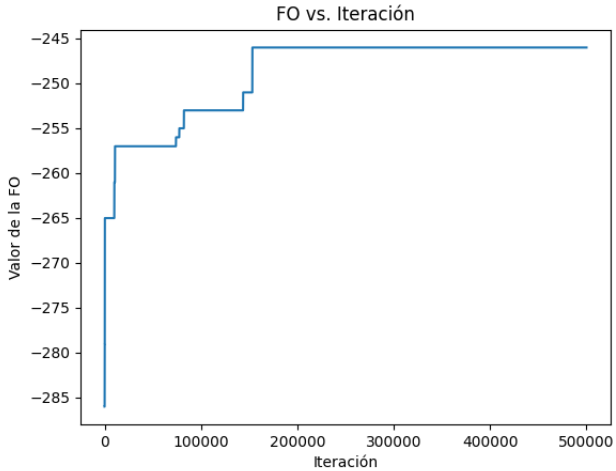


Tabla 15: Mejor solución obtenida por el algoritmo genético

Para esta solución también hemos representado la evolución de la FO con respecto al tiempo y al número de iteraciones, resultando las siguientes gráficas:





Previsiblemente, ambas gráficas son muy parecidas entre sí, dejando ver la evolución inicial del algoritmo y su posterior estancamiento. Se ha escogido como representativo ese numero de iteraciones debido al conocimiento de que, con un número mayor, la gráfica tampoco habría tenido evolución, como quedo demostrado con la ejecución mas larga presentada en este trabajo. Se corrobora, pues, que el algoritmo suele presentar una buena evolución hasta que queda estancado en algún optimo local.

En definitiva, son muchos los factores que hay que tener en cuenta en la construcción de un algoritmo que arroja soluciones para este problema concreto, si bien la mayoría de ellos han podido ser probados y descartados mediante la experimentación.

En el capítulo siguiente se presentarán las conclusiones del trabajo, así como sugerencias para futuras investigaciones al respecto.

5 CONCLUSIONES

En la gestión de operaciones portuarias, la asignación óptima del personal desempeña un papel crítico en la eficiencia y el rendimiento general del puerto. La tarea de programar adecuadamente los horarios y las asignaciones de trabajo para un equipo diverso y en constante movimiento de trabajadores portuarios es un desafío complejo que requiere una atención meticulosa a los detalles y la consideración de numerosos factores, debido no solo a la limitación de recursos que suponen los prácticos por la especialización del trabajo que realizan; sino también a la enorme dependencia que existe a este sistema de transporte, las regulaciones estrictas a las que está sometido por su peligrosidad y el gran impacto que generarían retrasos o daños.

Como hemos visto a lo largo de este trabajo, existen ejemplos en la literatura que tratan de utilizar diversos métodos de resolución para este problema en concreto y el de planificación de personal en términos generales. Esto involucra una serie de variables como la demanda de trabajo, las restricciones laborales y la disponibilidad de recursos.

Además, nuestro caso incluye la diferencia integral de la inclusión de la satisfacción de los empleados como métrica de optimización del problema, lo que supone un enfoque innovador que cada vez tiene más auge en el ámbito de la planificación de personal debido a los beneficios que supone para la producción y los costes de la empresa. Tradicionalmente, se han utilizado métodos exactos, heurísticas y otras aproximaciones para tratar de acercarse al problema de planificación, con resultados divergentes en función de los muchos parámetros que este puede adoptar.

Sin embargo, el que tenemos entre manos es un problema complejo precisamente debido a todas estas consideraciones, que suponen que un gran porcentaje de las soluciones que puedan obtenerse serán no factibles, complicando así la evolución hacia los resultados deseados.

Este trabajo se enfoca en dos aspectos clave. Primero, se presenta la implementación del algoritmo genético diseñado específicamente para abordar la optimización de la planificación del personal en el contexto portuario. Se han descrito las consideraciones previas, los operadores genéticos y las variables de decisión incorporadas en nuestro enfoque. Después, se han presentado y discutido los resultados obtenidos a través de la implementación de este algoritmo con un caso de prueba real; y se ha analizado la eficiencia de este en numerosos escenarios, comparándolos con los resultados con los obtenidos mediante la optimización exacta.

Si bien, como hemos dicho, estos resultados no eran los esperados en el planteamiento, sí que hemos podido obtener conclusiones sobre los pros y contras de implementar el algoritmo genético para este problema en concreto. En definitiva, se cree que este tipo de método de resolución es potente en cuanto al número de recursos computacionales utilizados, pero su buena construcción lleva aparejada una gran carga de trabajo por la necesidad de implementar mecanismos muy específicos que favorezcan la convergencia.

Ya hemos hablado de las limitaciones a la hora de realizar esta experimentación, pero cabe destacar los supuestos de los que partimos a la hora de elaborar este trabajo. En primer lugar, la limitación de recursos de tiempo y de computación, de forma que no ha sido posible ser tan exhaustivos en la ejecución, pudiendo quizás haber obtenido convergencia con mayor número de iteraciones. Por otro lado, el diseño de la investigación se ha hecho de forma experimental, variando de forma manual los datos y observando y analizando los resultados para sacar conclusiones acerca de la idoneidad de cada uno de los parámetros seleccionados. La metodología escogida, a su vez, aunque apropiada en principio para este tipo de problemas, quizás limite el alcance de los resultados obtenidos ya que, si bien los algoritmos genéticos resultan eficaces a la hora de explorar espacios de soluciones muy grandes, no lo son tanto cuando este espacio de soluciones es, en su mayoría, no factible.

Con todo esto, y habiendo hecho una reflexión sobre los posibles aspectos a mejorar en la elaboración de este algoritmo, se proponen las siguientes implementaciones de cara a futuras investigaciones:

1. En primer lugar, la utilización de algoritmos de selección más avanzados, como la selección por ruleta

o clasificación, evitando así estancamientos y manteniendo la diversidad. El torneo que hemos utilizado es útil en la mayoría de casos, pero convendría explorar la posibilidad de que una modificación en este operador genético pudiera resultar beneficiosa para la consecución de una solución óptima.

2. Por otro lado, en cuanto a la inicialización de la población, se sugiere una heurística u otra métrica que favorezca una buena solución desde el principio de las iteraciones, evitando así una búsqueda inútil en el frente de soluciones. Los beneficios de la utilización de un algoritmo híbrido- combinado con cualquier otro método de resolución- han sido ampliamente discutidos en este trabajo, y es por ello por lo que se entiende que su aplicación sería interesante para el campo.
3. Mecanismos de reinicio: En las ocasiones en las que el algoritmo se estanca, podría valorarse implementar mecanismos de reinicio periódico que vuelvan a generar la población, desde cero o desde otra previamente encontrada con la suficiente diversidad. Esto es especialmente interesante para el caso que nos ocupa, ya que hemos encontrado numerosas ejecuciones en las que la solución se ha quedado estancada en un valor concreto, sin llegar este a ser óptimo.
4. Análisis de convergencia: Implementación de métricas que nos permitan observar la evolución más detallada de las generaciones, como gráficos de la evolución de la FO con respecto al tiempo. Estos ya han sido implementados en este trabajo, pero simplemente de forma representativa. Un estudio más exhaustivo de las mismas podría proporcionar pistas sobre las razones que llevan al estancamiento que hemos venido observando en la evolución de las soluciones, y por tanto sería fundamental de cara a futuros avances en esta dirección.

En resumen, el presente trabajo brinda una visión integral de la creación, implementación y resultados de un algoritmo genético diseñado para la optimización de la planificación del personal. La investigación a veces supone lanzar hipótesis que posteriormente sean descartadas, y en esta exploración también avanza la ciencia.

A través de la combinación entre pruebas prácticas y el ajuste de parámetros, este trabajo busca contribuir al avance de la gestión eficiente del personal; y aportar una herramienta que permita implementar la justicia, la equitatividad y la felicidad de los trabajadores como máximas en la toma de decisiones.

REFERENCIAS

1. Aardal, K., & Ari, A. (1987). Decomposition principles applied to the dynamic production and work-force scheduling problem. Engineering Costs and Production Economics, 12(1–4), 39–49.
2. Aggarwal, S. (1982a). "A Focused Review of Scheduling in Services." European Journal of Operational Research, 9(2), 114–121.
3. Agnihothri, S., & Taylor, P. (1991). Staffing a centralized appointment scheduling department in Lourdes Hospital. Interfaces, 21(5), 1-11.
4. Alam, T., Qamar, S., Dixit, A., Benaida, M. J. a. p. a. (2020). Genetic algorithm: Reviews, implementations, and applications.
5. Alfares, H. K. (2004). Survey, categorization, and comparison of recent tour scheduling literature. Annals of Operations Research, 127, 145–175. https://doi.org/10.1023/B:ANOR.0000019088.98647.e2
6. Alvarez-Valdes, R., Crespo, E., & Tamarit, J. (1999). Labour scheduling at an airport refuelling installation. Journal of the Operational Research Society, 50(3), 211-218.
7. Al-Yakoob, S. M., & Sherali, H. D. (2008). A column generation approach for the project scheduling problem with discounted cash flows. Journal of Scheduling, 11(5), 313-327.
8. Al-Zubaidi, H., Christer, A. (1997). Maintenance manpower modelling for a hospital building complex. European Journal of Operational Research, 99, 603-618.
9. Anbari, F. (1987). Train and engine crew management system. In Computers in Railway Operations (pp. 267-284). WIT Press.
10. Anbil, R., Gelman, E., Patty, B., & Tanga, R. (1991). Recent advances in crew-pairing optimization at American airlines. Interfaces, 21(1), 62-74.
11. Anwaar, A. S., Bhuiyan, I. A., Arani, M., Billal, M. M. (2020). Multi-Objective Optimization for Sustainable Closed-Loop Supply Chain Network Under Demand Uncertainty: A Genetic Algorithm. 2020 International Conference on Data Analytics for Business and Industry (ICDABI).
12. Ashley, D. (1995). A spreadsheet optimization system for library staff scheduling. Computers and Operations Research, 22(6), 615-624.
13. Awad, R., & Chinneck, J. (1998). Proctor assignment at Carleton University. Interfaces, 28(2), 58-71.
14. Aykin, T. (1996). Optimal Shift Scheduling with Multiple Break Windows. Management Science, 42(4), 591–602. https://doi.org/10.1287/mnsc.42.4.591
15. Aykin, T. (2000). Comparative evaluation of modeling approaches to the labor shift scheduling problem. European Journal of Operational Research, 125(2), 381–397. https://doi.org/10.1016/S0377-2217(99)00413-0
16. Azaiez, M. N., & Al Sharif, A. A. (2005). An interactive goal programming approach for project scheduling under uncertainty. Journal of Construction Engineering and Management, 131(1), 46-54.
17. Azarmi, N., Abdulhameed, W. (1995). Workforce scheduling with constraint logic programming. BT Technology Journal, 13(1), 81-94.
18. Baker, E., Bodin, L., Finnegan, W., & Ponder, R. (1979). Efficient heuristic solutions to an airline crew scheduling problem. IIE Transactions, 11(2), 79-85.

19. Baker, K. R. (1976). Workforce allocation in Cyclical Scheduling Problems: A survey. Journal of the Operational Research Society, 27(1), 155-167. https://doi.org/10.1057/jors.1976.30
20. Ball, M., Roberts, A., & A graph partitioning approach to airline crew scheduling. Transportation Science, 19(2), 107-126.
21. Bard, J. F., & Purnomo, H. (2007). Lagrangean relaxation-based heuristics for the resource-constrained project scheduling problem. European Journal of Operational Research, 183(3), 1239-1253.
22. Bechtold, S. E., & Jacobs, L. W. (1990). Implicit Modeling of Flexible Break Assignments in Optimal Shift Scheduling. Management Science, 36(11), 1339–1351. https://doi. org/10.1287/mnsc.36.11.1339
23. Bechtold, S. E., Brusco, M. J., & Showalter, M. J. (1991). A comparative evaluation of labor tour scheduling methods. Decision Sciences, 22(4), 683-699. https://doi.org/10.1111/j.1540-5915.1991.tb00359.x
24. Beliën, J., & Demeulemeester, E. (2007). A branch-and-bound procedure for the multiple resource-constrained project scheduling problem. European Journal of Operational Research, 180(2), 679-697.
25. Bellanti, F., Carello, G., Della Croce, F., & Tadei, R. (2004). A greedy based neighborhood search approach to a nurse rostering problem. European Journal of Operational Research, 153(1), 28-40.
26. Bodin, L., Golden, B., Assad, A., & Ball, M. (1983). Routing and scheduling of vehicles and crews--the state of the art. Computers and Operations Research, 10(2), 63-211.
27. Burke, E., Causmaecker, P. D., Berghe, G. V., & A hybrid tabu search algorithm for the nurse rostering problem. Lecture Notes in Computers Science, 1585, 187-194.
28. Cappanera, P., & Gallo, G. (2004). A Multicommodity Flow Approach to the Crew Rostering Problem. Operations Research, 52(4), 583–596. https://doi.org/10.1287/ opre.1040.0110
29. Caprara, A., Fischetti, M., Guida, P., Toth, P., & Vigo, D. (1999). Solution of large-scale railway crew planning problems: The Italian experience. In Computer-Aided Transit Scheduling (pp. 1-18). Springer.
30. Carraresi, P., Nonato, M., & Girardi, L. (1995). Network models, Lagrangean relaxation and subgradient bundle approach in crew scheduling problem. In Computer-Aided Transit Scheduling (pp. 188-212). Springer.
31. Cavicchio DJ (1970) Adaptive search using simulated evolution. Unpublished doctoral dissertation. University of Michigan, Ann Arbor.
32. Cheng, B., Lee, J., & Wu, J. (1997). A nurse rostering system using constraint programming and redundant modeling. IEEE Transactions on Information Technology in Biomedicine, 1(1), 44–54.
33. Chu, S., & Chan, E. (1998). Crew scheduling of light rail transit in Hong Kong: From modeling to implementation. Computers and Operations Research, 25(11), 887-894.
34. Cowling, P., Kendall, G., Han, L. (2002). An investigation of a hyperheuristic genetic algorithm applied to a trainer scheduling problem. Proceedings of the 2002 Congress on Evolutionary Computation.
35. Crainic, T., & Rousseau, J. (1987). The column generation principle and the airline crew scheduling problem. INFOR, 25(2), 136-151.
36. Curtois, T., Ochoa, G., Hyde, M., & Vázquez-Rodríguez, J. A. (2010). A hyflex module for the personnel scheduling problem. School of Computer Science, University of Nottingham, Tech. Rep.
37. Dantzig, G. B. (1954). Letter to the Editor—A Comment on Edie's "Traffic Delays at Toll Booths". Journal of the Operations Research Society of America, 2(3), 339–341. https://doi.org/10.1287/opre.2.3.339
38. De Causmaecker, P., Demeester, P., Berghe, V., & Verbeke, B. (2004). Analysis of real-world personnel scheduling problems. In Proceedings of the 5th international conference on practice and theory of automated timetabling, PATAT 2004 (18th August - 20th August 2004) (pp. 183–198). Pittsburgh, PA, USA.

39. Dias, T. M., Ferber, D., de Souza, C., & Moura, A. V. (2003). Constructing nurse schedules at large hospitals. International Transactions in Operational Research, 10(3), 245-265.

40. Dowling, D., Mackenzie, H., Krishnamoorthy, M., & Sier, D. (1997). Staff rostering at a large international airport. Annals of Operations Research, 72, 125-147.

41. Easton, F. F., Mansour, N. (1999). A distributed genetic algorithm for deterministic and stochastic labor scheduling problems. European Journal of Operational Research, 118(3), 505-523.

42. Edie, L. C. (1954). Traffic delays at toll booths. Journal of the Operations Research Society of America, 2(2), 107-138. https://doi.org/10.1287/opre.2.2.107

43. Eiselt, H. A., & Marianov, V. (2008). Project scheduling under uncertainty: Survey and research potentials. European Journal of Operational Research, 184(2), 416-432.

44. Eitzen, G. A., Panton, D. M., & Mills, G. H. (2004). An integer programming approach to the resource-constrained project scheduling problem. Journal of the Operational Research Society, 55(11), 1186-1192.

45. Elshafei, M., & Alfares, H. K. (2007). A dynamic programming algorithm for days-off scheduling with sequence dependent labor costs. Journal of Scheduling, 11(2), 85-93. https://doi.org/10.1007/s10951-007-0040-x

46. Ernst, A. T., Jiang, H., Krishnamoorthy, M., Owens, B., & Sier, D. (2004a). An annotated bibliography of personnel scheduling and rostering. Annals of Operations Research, 127, 21–144. https://doi.org/10.1023/B:ANOR.0000019087.46656.e2

47. Ernst, A., Hourigan, P., Krishnamoorthy, M., Mills, G., Nott, H., & Sier, D. (1999). Rostering ambulance officers. In Proceedings of the 15th National Conference of the Australian Society for Operations Research (pp. 470-481).

48. Evans, J. (1988). A microcomputer-based decision support system for scheduling umpires in the American baseball league. Interfaces, 18(6), 42-51.

49. Evans, J. (1988). A microcomputer-based decision support system for scheduling umpires in the American baseball league. Interfaces, 18(6), 42–51.

50. Faaland, B., Schmitt, T. (1993). Cost-based scheduling of workers and equipment in a fabrication and assembly shop. Operations Research, 41(2), 253-268.

51. Feiring, B. (1993). A model generation approach to the personnel assignment problem. Journal of the Operational Research Society, 44(5), 503-512.

52. Firat, A. F., & Hurkens, C. A. (2012). A mixed integer programming approach for the multi-mode resource-constrained project scheduling problem. European Journal of Operational Research, 219(3), 573-582.

53. Fowler, J. W., Wirojanagud, P., & Gel, E. S. (2008). Solving the resource-constrained project scheduling problem with a multiobjective genetic algorithm. IEEE Transactions on Evolutionary Computation, 12(4), 481-499.

54. Gharsalli, L. (2022). Hybrid Genetic Algorithms. IntechOpen. doi: 10.5772/intechopen.104735

55. Glen, J. (1975). A dynamic programming model for work scheduling in a shipyard. Operational Research Quarterly, 26(4), 787-799.

56. Goldberg, D. E. (1989). Genetic Algorithms in Search, Optimization, and Machine Learning. Addison-Wesley.

57. Gopalakrishnan, M., Gopalakrishnan, S., & Miller, D. (1993). A decision support system for scheduling personnel in a newspaper publishing environment. Interfaces, 23(4), 104-115.

58. Gopalan, J., Korkmaz, E., Alhajj, R., Barker, K. (2005). Effective data mining by integrating genetic algorithm into the data preprocessing phase. Fourth International Conference on Machine Learning and

Applications (ICMLA'05).

59. Grossman, T., D. Samuelson, S. Oh, and T. Rohleder. (1999). "Call Centers." Technical Report, Haskayne School of Business, University of Calgary.
60. He, F., & Qu, R. (2012). A constraint programming based column generation approach to nurse rostering problems. Computers & Operations Research, 39(12), 3331-3343. https://doi.org/10.1016/j.cor.2012.04.018
61. Hertz, A., Lahrichi, N., & Widmer, M. (2010). A mixed integer linear programming formulation for the resource-constrained project scheduling problem with discounted cash flows. Journal of Scheduling, 13(5), 495-506.
62. Hochbaum, D. S., & Levin, A. (2006). The complexity of the resource-constrained project scheduling problem. Operations Research Letters, 34(3), 265-270.
63. Hojati, M., & Patil, R. (2011). A new linear programming model for the resource-constrained project scheduling problem. Journal of Applied Mathematics and Computing, 36(1-2), 247-264.
64. Holland, J. H. (1975). Adaptation in Natural and Artificial Systems. University of Michigan Press.
65. Holland, J. H. (1992). Genetic algorithms. Scientific American, 267(1), 66-73.
66. Isken, M., & Hancock, W. (1991). A heuristic approach to nurse scheduling in hospital units with non-stationary, urgent demand, and a fixed staff size. Journal of the Society for Health Systems, 24-41.
67. Jaumard, B., Semet, F., & Vovor, T. (1998). A generalized linear programming model for nurse scheduling. European Journal of Operational Research, 107(1), 1–18.
68. Katoch, S., Chauhan, S. S., Kumar, V. (2021). A review on genetic algorithm: Past, present, and future. Multimedia Tools and Applications, 80(5), 8091-8126. https://doi.org/10.1007/s11042-020-10139-6
69. Lambora, A., Gupta, K., Chopra, K. (2019). Genetic Algorithm - A Literature Review. 2019 International Conference on Machine Learning, Big Data, Cloud and Parallel Computing (COMITCon).
70. Laporte, G., & Pesant, G. (2004). A branch-and-cut-and-price algorithm for the resource-constrained project scheduling problem. Management Science, 50(5), 714-722.
71. Lau, H. C. W., Chan, T. M., Tsui, W. T., Chan, F. T., Ho, G., & Choy, K. L. T. (2009). A fuzzy guided multi-objective evolutionary algorithm model for solving transportation problem. Expert Systems With Applications, 36(4), 8255-8268. https://doi.org/10.1016/j.eswa.2008.10.031
72. Lau, H. C. W., Chan, T. M., Tsui, W. T., Ho, G. T. S. (2009). Cost optimization of the supply chain network using genetic algorithms. IEEE Transactions on Knowledge and Data Engineering.
73. Li, X., Qian, X., Wang, Z. (2009). Classification rule mining using feature selection and genetic algorithm. 2009 Asia-Pacific Conference on Computational Intelligence and Industrial Applications (PACIIA).
74. Liang, T., & Buclatin, B. (1988). Improving the utilization of training resources through optimal assignment in the US navy. European Journal of Operational Research, 33, 183-190.
75. Lin, S., & Ying, K. (2014). Minimizing shifts for personnel task scheduling problems: a three-phase algorithm. European Journal of Operational Research, 237(1), 323-334. https://doi.org/10.1016/j.ejor.2014.01.035
76. Lin, S., & Ying, K. (2015). A multi-point simulated annealing heuristic for solving multiple objective unrelated parallel machine scheduling problems. International Journal of Production Research, 53(4), 1065-1076. https://doi.org/10.1080/00207543.2014.942011
77. Lin, Y. C., Chen, S. H., Chou, J. H., & Liao, C. N. (2012). A goal programming approach for the multi-mode resource-constrained project scheduling problem. Journal of the Operational Research Society, 63(11), 1602-1613.

78. Lorenzo-Espejo, A., Muñuzuri, J., Onieva, L., & Cortés, P. (2021). Scheduling consecutive days off: a case study of maritime pilots. Computers & Industrial Engineering, 155, 107192. https://doi.org/10.1016/j.cie.2021.107192

79. Lorenzo Espejo, A. (2023). Optimización de procesos logísticos: planificación de recursos materiales y humanos en una terminal portuaria. (Tesis Doctoral). Universidad de Sevilla, Sevilla.

80. Maier-Rothe, C., & Wolfe, H. B. (1973). Cyclical scheduling and allocation of nursing staff. Socio-economic Planning Sciences, 7(5), 471-487. https://doi.org/10.1016/0038-0121(73)90043-8

81. Mason, A., Ryan, D., & Panton, D. (1998). Integrated simulation, heuristic and optimization approaches to staff scheduling. Operations Research, 46(2), 161-175.

82. Mehrotra, V. (1997). "Ringing up Big Business." OR/MS Today, 24(4).

83. Mesquita, M., Moz, M., Paias, A., & Pato, M. (2015). A decompose-and-fix heuristic based on multi-commodity flow models for driver rostering with days-off pattern. European Journal of Operational Research, 245(2), 423–437. https://doi.org/10.1016/j. ejor.2015.03.030

84. Millar, H., & Kiragu, M. (1998). Cyclic and non-cyclic scheduling of 12 h shift nurses by network programming. European Journal of Operational Research, 104(3), 582–592.

85. Mould, G. (1996). Case study of manpower planning for clerical operations. Journal of the Operational Research Society, 47(3), 358-368.

86. Naso, D., Surico, M., Turchiano, B., & Kaymak, U. (2007). Genetic Algorithms for supply-chain scheduling: A case study in the distribution of ready-mixed concrete. European Journal of Operational Research, 177(3), 2069-2099. https://doi.org/10.1016/j.ejor.2005.12.019

87. Nooriafshar, M. (1995). A heuristic approach to improving the design of nurse training schedules. European Journal of Operational Research, 81(1), 50-61.

88. Norby, R., Freund, L., & Wagner, B. (1977). A nurse staffing system based on assignment difficulty. Journal of Nursing Administration, 7(9), 2–24.

89. Nordström, A. L., Tufekci, S. (1994). A genetic algorithm for the talent scheduling problem. Computers & Operations Research, 21(8), 803-816.

90. Oh, I., Lee, J., & Moon, B. (2004). Hybrid genetic algorithms for feature selection. IEEE Transactions on Pattern Analysis and Machine Intelligence, 26(11), 1424-1437. https://doi.org/10.1109/tpami.2004.105

91. Ozkarahan, I. (1991). A disaggregation model of a flexible nurse scheduling support system. Socio-economic Planning Sciences, 25(1), 9–26.

92. Pot, A. M., Bhulai, S., & Koole, G. M. (2008). Lagrangian relaxation for the resource-constrained project scheduling problem with discounted cash flows. European Journal of Operational Research, 190(1), 70-83.

93. Qu, R., & He, F. (2009). A constraint programming approach to the resource-constrained project scheduling problem. Journal of the Operational Research Society, 60(11), 1601-1611.

94. Radhakrishnan, P., Prasad, V. M., Gopalan, M. R. (2009). Genetic Algorithm Based Inventory Optimization Analysis in Supply Chain Management. 2009 IEEE International Advance Computing Conference.

95. Restrepo, J. I., Lozano, S., & Medaglia, A. L. (2012). A column generation approach for the resource-constrained project scheduling problem with cash flows and time-dependent discount rates. Journal of Scheduling, 15(3), 257-272.

96. Ritzman, L. P., Krajewski, L., & Showalter, M. (1976). The disaggregation of aggregate manpower plans. Management Science, 22(11), 1204-1214.

97. Rocha, M., Oliveira, J., & Carravilla, M. (2012). Quantitative Approaches on Staff Scheduling and Rostering in Hospitality Management: An Overview. American Journal of Operations Research, 2(1), 137-145.

98. Rong, A. (2010). Monthly tour scheduling models with mixed skills considering weekend off requirements. Computers & Industrial Engineering, 59(2), 334–343. https://doi.org/10.1016/j.cie.2010.05.005

99. Ryan, T., Barker, B., & Marciante, F. (1975). A system for determining appropriate nurse staffing. Journal of Nurse Administration, 5(5), 30–38.

100. Sarin, S., & Aggarwal, S. (2001). Modeling and algorithmic development of a staff scheduling problem. European Journal of Operational Research, 128, 558-569.

101. Seckiner, T., Gokcen, H., & Kurt, M. (2007). A new integer programming formulation for the resource-constrained project scheduling problem. Journal of the Operational Research Society, 58(5), 691-697.

102. Siferd, S., & Benton, W. (1994). A decision mode for shift scheduling of nurses. European Journal of Operational Research, 74(3), 519-527.

103. Sinuany-Stern, Z., & Teomi, Y. (1986). Multi-objective scheduling plans for security guards. Journal of the Operational Research Society, 37(1), 67-77.

104. Sinuany-Stern, Z., & Teomi, Y. (1986). Multi-objective scheduling plans for security guards. Journal of the Operational Research Society, 37(1), 67–77.

105. Sivanandam, S., Deepa, S. (2008). Genetic Algorithm Optimization Problems. In: Introduction to Genetic Algorithms. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-540-73190-0_7

106. Taghizadehalvandi, M., & Kamisli Ozturk, Z. (2019). Multi-objective solution approaches for employee shift scheduling problems in service sectors (research note). International Journal of Engineering, 32(9), 1312-1319.

107. Tang, H., & Lu, J. (2007). A Hybrid Algorithm Combined Genetic Algorithm with Information Entropy for Data Mining. In ICIEA 2007: 2nd IEEE Conference On Industrial Electronics And Applications, Vols 1-4, Proceedings. https://doi.org/10.1109/iciea.2007.4318508

108. Tanweer Alam, Shamimul Qamar, Amit Dixit, Mohamed Benaida. (2020). Genetic Algorithm: Reviews, Implementations, and Applications. International Journal of Engineering Pedagogy (iJEP).

109. Taylor, P., & Huxley, S. (1989). A break from tradition for the San Francisco police: Patrol officer scheduling using an optimization-based decision support system. Interfaces, 19(1), 4-24.

110. Tien, J. M., & Kamiyama, A. (1982). On Manpower Scheduling Algorithms. SIAM Review, 24(3), 275–287. https://doi.org/10.2307/2030177

111. Tiwari, A. K., Ramakrishna, G., Sharma, L. K., Kashyap, S. K. (2019). Neural Network and Genetic Algorithm based Hybrid Data Mining Algorithm. 2019 International Conference on Computing, Communication, and Intelligent Systems (ICCCIS).

112. Topaloglu, S., & Ozkarahan, I. (2003). A goal programming approach for the resource-constrained project scheduling problem. Journal of the Operational Research Society, 55(6), 591-600.

113. Topaloglu, S., & Ozkarahan, I. (2003). A goal programming approach for the resource-constrained project scheduling problem. Journal of the Operational Research Society, 55(6), 591-600.

114. Trivedi, V. (1974). Optimum allocation of float nurses using head nurses' perspectives (Tesis de doctorado). Universidad de Michigan.

115. Tsai, C. C., Li, S. H. A., Moon, B. R. (2002). An investigation of a hyperheuristic genetic algorithm applied to a trainer scheduling problem. Proceedings of the 2002 Congress on Evolutionary Computation.

116. Turhan, A. M., & Bilgen, B. (2020). A hybrid fix-and-optimize and simulated annealing approaches for nurse rostering problem. Computers & Industrial Engineering, 145, 106531. https://doi.org/10.1016/j.cie.2020.106531.

117. Van Den Bergh, J., Belien, J., De Bruecker, P., Demeulemeester, E., & De Boeck, L. (2013). Personnel Scheduling: A Literature review. European Journal of Operational Research, 226(3), 367-385. https://doi.org/10.1016/j.ejor.2012.11.029

118. Warner, D. (1976). Scheduling nursing personnel according to nursing preference: A mathematical programming approach. Operations Research, 24(5), 842–856.

119. Warner, D., & Prawda, J. (1972). A mathematical programming model for scheduling nursing personnel in a hospital. Management Science, 19(4), 411–422.

120. Weil, G., Heus, K., Francois, P., & Poujade, M. (1995). Constraint programming for nurse scheduling. IEEE Engineering in Medicine and Biology, 14(4), 417-422.

121. Wermus, M., & Pope, J. A. (1994). Scheduling Harbor pilots. Interfaces, 24(2), 44-52. https://doi.org/10.1287/inte.24.2.44

122. Wright, M. (1991). Scheduling English cricket umpires. Journal of the Operational Research Society, 42(6), 447–452.

123. Yilmaz, O. (2012). A mixed integer programming approach for the resource-constrained project scheduling problem with multiple execution modes. Journal of the Operational Research Society, 63(11), 1622-1635.

ANEXO 1: TABLA DE CONFIGURACIONES DE PARÁMETROS DE ENTRADA

Configuración	Solución inicial	Tamaño población	Nº de iteraciones	Peso penalizaciones	Probabilidad crossover	Tipo Crossover	Probabilidad mutación	Tipo de mutación	Elitismo
1	Aleatoria	300	100000	1	0.8	Un solo punto, por filas	0.3	Mantiene demanda	N
2	Aleatoria	300	100000	1	0.9	Un solo punto, por filas	0.3	Mantiene demanda	N
3	Aleatoria	300	100000	1	0.9	Un solo punto, por filas	0.5	Mantiene demanda	N
4	Aleatoria	300	100000	1	0.5	Un solo punto, por filas	0.5	Mantiene demanda	N
5	Aleatoria	300	100000	1	0.5	Un solo punto, por filas	0.5	Mantiene demanda	S
6	Aleatoria	300	100000	1	0.8	Un solo punto, por filas	0.3	Mantiene demanda	S
7	Aleatoria	300	100000	1	0.9	Un solo punto, por filas	0.3	Mantiene demanda	S
8	57% de 1s	300	100000	1	0.9	Un solo punto, por filas	0.3	Mantiene demanda	S
9	Nº de 1s	300	100000	1	0.9	Un solo punto, por filas	0.3	Mantiene demanda	S
10	Nº de 1s	300	100000	1	0.9	Varios puntos, por	0.3	Mantiene demanda	S

						filas			
11	Nº de 1s	300	100000	1	0.9	Varios puntos, por filas	0.3	Mantiene demanda	S
12	Nº de 1s	300	100000	1	0.9	Varios puntos, por filas	0.5	Mantiene demanda	S
13	Nº de 1s	300	300000	1	0.9	Varios puntos, por filas	0.5	Mantiene demanda	S
14	Nº de 1s	100	500000	1	0.9	Varios puntos, por filas	0.5	Mantiene demanda	S
15	Nº de 1s	100	500000	[2,1,1,2]	0.9	Varios puntos, por filas	0.5	Mantiene demanda	S
16	Nº de 1s	100	500000	[2,1,1,2]	0.8	Varios puntos, por filas	0.5	Mantiene demanda	S
17	Nº de 1s	300	100000	[2,1,1,2]	0.8	Varios puntos, por filas	0.3	Mantiene demanda	S
18	Nº de 1s	300	300000	[2,1,1,2]	0.8	Varios puntos, por filas	0.3	Mantiene demanda	S
19	Nº de 1s	300	300000	[2,1,1,3]	0.8	Varios puntos, por filas	0.5	Mantiene demanda	S
20	Nº de 1s	300	300000	[2,1,1,3]	0.8	Varios puntos, por filas	0.3	Mantiene demanda	S
21	Nº de 1s	300	300000	[1,1,1,2]	0.8	Varios puntos, por filas	0.3	Mantiene demanda	S
22	Nº de 1s	300	300000	[1,1,1,2]	0.8	Varios puntos, por filas	0.5	Mantiene demanda	S
23	Nº de 1s	300	300000	[1,1,1,2]	0.8	Varios puntos, por filas	Variable: [0.7-0.1]	Mantiene demanda, prob. Variable	S
2	Nº de 1s	300	300000	[1,1,1,2]	0.8	combinado, filas+column	Variable: [0.7-	Mantiene demanda, prob.	S

4						as	0.1]	Variable	
25	Nº de 1s	300	300000	[1,1,1,2]	0.8	combinado, filas+columnas	0.5	Mantiene demanda, prob. Variable	S
26	Nº de 1s	300	300000	[1,1,1,2]	0.9	combinado, filas+columnas	Variable: [0.7-0.1]	Mantiene demanda, prob. Variable	S
27	Nº de 1s	3000	30000	[1,1,1,2]	0.9	combinado, filas+columnas	Variable: [0.7-0.1]	Mantiene demanda, prob. Variable	S
28	Nº de 1s	3000	300000	[1,1,1,2]	0.9	combinado, filas+columnas	Variable: [0.7-0.1]	Mantiene demanda, prob. Variable	S
29	Nº de 1s	100	300000	[1,1,1,2]	0.9	combinado, filas+columnas	Variable: [0.7-0.1]	Mantiene demanda, prob. Variable	S
30	Nº de 1s	100	300000	[1,1,1,2]	0.6	combinado, filas+columnas	Variable: [0.7-0.1]	Mantiene demanda, prob. Variable	S
31	Aleatoria	100	100000	[1,1,1,1]	0.8	Un solo punto, por filas	0.3	aleatoria	N
32	Nº de 1s	100	500000	[1,1,1,2]	0.6	combinado, filas+columnas	Variable: [0.7-0.1]	Mantiene demanda, prob. Variable	S
33	Nº de 1s	100	300000	[1,1,1,2]	0.6	combinado, filas+columnas	0.5	Mantiene demanda	S
34	Nº de 1s	100	300000	[1,1,1,2]	0.6	combinado, filas+columnas	0.3	Mantiene demanda	S
35	Aleatoria	100	500000	[1,1,1,2]	0.8	Un solo punto, por filas	0.3	aleatoria	N
36	Aleatoria	100	500000	[1,1,1,4]	0.8	Un solo punto, por filas	0.3	aleatoria	N
37	Aleatoria	100	500000	[1,1,1,8]	0.8	Un solo punto, por filas	0.3	aleatoria	S

38	Num 1s	100	5000000	[1,1,1,2]	0.8	combinado, filas+columnas	0.3	Mantiene demanda	S
39	Demanda diaria	100	500000	[1,1,1,2]	0.8	combinado, filas+columnas	Variable: [0.7-0.1]	Mantiene demanda	S
40	Demanda diaria	300	10000	[1,1,1,1]	0.9	Un solo punto, filas	0.3	Mantiene demanda	S
41	Demanda diaria	100	1000000	[1,1,1,2]	0.9	Un solo punto, filas	0.5	Mantiene demanda	S
42	Demanda diaria	100	1000000	[1,1,1,2]	0.9	Un solo punto, filas	Variable: [0.9-0.2]	Mantiene demanda	S
43	Demanda diaria	100	1000000	[2,1,1,1]	0.9	Un solo punto, filas	Variable: [0.9-0.2]	Mantiene demanda	S
44	Demanda diaria	300	500000	[2,2,1,1]	0.9	Un solo punto, filas	Variable: [0.9-0.2]	Mantiene demanda	S

ANEXO 2: CÓDIGOS UTILIZADOS

[1]Experimento 1: Selección por torneo de 3, crossover por filas de 1 punto y mutación aleatoria. Sin elitismo.

```
# módulos de Python que vamos a utilizar
import random
import numpy as np
import time

import warnings
warnings.filterwarnings("ignore")



# paso1: función de evaluación / fitness function
def evalua(individuo):
    num_workers, num_days = np.shape (individuo)
    PDj = [7, 9, 8, 9, 10, 7, 6]  # Demanda para cada día de la semana
    from itertools import cycle
    penalties = {
        "descansos_insuficientes": 0,
        "trabajos_consecutivos": 0,
        "limite_dias_trabajados": 0,
        "demanda_diaria": 0
    }
    longest_breaks = []
    dias_trabajados = 0
    workforce_diario = 0
    #Cálculo de dias totales trabajados
    dias_trabajados = np.sum(individuo, axis = 1)
    #Cálculo de los trabajadores diarios
    workforce_diario = np.sum(individuo, axis = 0)
    #Calculo de descansos
    for i in range(num_workers):
        breaks = []
        count = 0
        consecutive_workdays = 0
        for j in range(num_days):
            if individuo[i][j] == 0:
                count += 1
                if consecutive_workdays > 5:
                    penalties["trabajos_consecutivos"] += 1
                consecutive_workdays = 0
            else:
                if count > 0:
                    breaks.append(count)
                count = 0
                consecutive_workdays += 1
        if count > 0:
            breaks.append(count)
        if breaks:
            longest_breaks.append(max(breaks))
```

```
            penalties["descansos_insuficientes"] += sum(1 for x in breaks if
x < 2)
        else:
            longest_breaks.append(0)

        if consecutive_workdays > 5:
            penalties["trabajos_consecutivos"] += 1

    if any(x>63 for x in dias_trabajados):
        penalties["limite_dias_trabajados"] +=1

    PDj_cycle = cycle(PDj)
    for i in range(num_days):
        demanda_diaria = next(PDj_cycle)
        if workforce_diario[i] != demanda_diaria:
            penalties["demanda_diaria"] += 1


    objective1 = sum (longest_breaks)
    objective2 = (max (longest_breaks) - min(longest_breaks))
    total_penalty = sum(penalties.values())
    fitness = int(objective1) - int(objective2) - int(total_penalty)
    return (fitness)


#imprime la matriz con los descansos marcados en verde (solo visualizacion)
def imprimir_matriz (matriz) :
    s = [[str(e) for e in row] for row in matriz]
    lens = [max(map(len, col)) for col in zip(*s)]
    formatted_rows = []
    for row in matriz:
        formatted_row = []
        for cell in row:
            # Si el valor de la celda es 0, cambia el color del texto a verde
            if cell == 0:
                formatted_cell = '\x1b[32m{}\x1b[0m'.format(cell)
            else:
                formatted_cell = str(cell)
            formatted_row.append(formatted_cell)
        formatted_rows.append(formatted_row)
    fmt = '\t'.join('{{:{}}}'.format(x) for x in lens)
    table = [fmt.format(*row) for row in formatted_rows]
    return table

#Generacion de una poblacion con un 65% de 1s
def generate_population(population_size, num_rows, num_cols):
    population = []
    random.seed(64)
    for _ in range(population_size):
        # Genera una matriz binaria aleatoria
        individual = np.random.randint(2, size=(num_rows, num_cols))
        population.append(individual)
    return population


# Operaciones genéticas


def main():
    random.seed(64) # semilla del generador de números aleatorios
    start_time = time.time()
    p_cross = 0.8
```

```
    p_mutation = 0.3
    n_iteration = 100000
    size_population = 300

    pop = generate_population(size_population,14,98)

    population_evaluated = list(zip(map(lambda gen: evalua(gen), pop), pop))


    best_solution_value_i, best_solution_i = max(population_evaluated,
key=lambda x: x[0])
    best_solution_value = best_solution_value_i
    best_solution = best_solution_i
    contador_mutacion = 0
    # Comenzamos la evolución:
    for i in range(n_iteration):
        # Selection - Tournament
        size_tournament = 3
        selec1 = np.random.permutation(size_population)[0:size_tournament]
        selec2 = np.random.permutation(size_population)[0:size_tournament]

        value1, ind1 = max((population_evaluated[selec1[i]][0], selec1[i])
for i in range(size_tournament))
        value2, ind2 = max((population_evaluated[selec2[i]][0], selec2[i])
for i in range(size_tournament))


        if np.random.rand() <= p_cross:
            # Generar un único punto de corte en la mitad de la matriz
            parent1 = pop[ind1]
            parent2 = pop[ind2]
            cross_point = len(parent1) // 2

            descendent1 = np.concatenate((parent1[:, :cross_point],
parent2[:, cross_point:]), axis=1)
            descendent2 = np.concatenate((parent2[:, :cross_point],
parent1[:, cross_point:]), axis=1)
        else:
            descendent1 = pop[ind1]
            descendent2 = pop[ind2]

        if np.random.rand() <= p_mutation:
            # Elegir un índice de fila y columna al azar para la mutación
            mutation_indices = np.random.randint(0, descendent1.size,
size=(3, 2))
            contador_mutacion +=1
            for row_idx, col_idx in mutation_indices:
                row = row_idx % descendent1.shape[0]
                col = col_idx % descendent1.shape[1]

                if descendent1[row, col] == 1:
                    # Cambiar de 1 a 0 en la posición seleccionada
                    descendent1[row, col] = 0

                    # Encontrar los índices de elementos 0 en la misma
columna
                    zero_indices = np.where(descendent1[:, col] == 0)[0]

                    # Elegir al azar un índice de elemento 0 para cambiar a 1
                    if zero_indices.size > 0:
```

```
                        idx_to_change = np.random.choice(zero_indices)
                        descendent1[idx_to_change, col] = 1

        if np.random.rand() <= p_mutation:
            # Elegir un índice de fila y columna al azar para la mutación
            mutation_indices = np.random.randint(0, descendent2.size,
size=(3, 2))
            contador_mutacion +=1
            for row_idx, col_idx in mutation_indices:
                row = row_idx % descendent2.shape[0]
                col = col_idx % descendent2.shape[1]

                if descendent2[row, col] == 1:
                    # Cambiar de 1 a 0 en la posición seleccionada
                    descendent2[row, col] = 0

                    # Encontrar los índices de elementos 0 en la misma
columna
                    zero_indices = np.where(descendent2[:, col] == 0)[0]

                    # Elegir al azar un índice de elemento 0 para cambiar a 1
                    if zero_indices.size > 0:
                        idx_to_change = np.random.choice(zero_indices)
                        descendent2[idx_to_change, col] = 1

        descendent1_value = evalua(descendent1)
        descendent2_value = evalua(descendent2)

        if descendent1_value > best_solution_value:
            best_solution_value = descendent1_value
            best_solution = descendent1

        if descendent2_value > best_solution_value:
            best_solution_value = descendent2_value
            best_solution = descendent2

            # Selection - Reject
        size_tournament = 3
        selec1 = np.random.permutation(size_population)[0:size_tournament]
        selec2 = np.random.permutation(size_population)[0:size_tournament]

        _, ind1 = min((population_evaluated[selec1[i]][0], selec1[i]) for i
in range(size_tournament))
        _, ind2 = min((population_evaluated[selec2[i]][0], selec2[i]) for i
in range(size_tournament))

        population_evaluated[ind1] = descendent1_value, descendent1
        population_evaluated[ind2] = descendent2_value, descendent2



        tiempo_iteracion = time.time()
        print("Iteración:", i)
        print("Mejor solución:", best_solution_value)
        print("Tiempo transcurrido:", tiempo_iteracion-start_time)
        print("Tamaño de la población:", len(population_evaluated))

    #Evaluacion de la solucion final:
    num_workers, num_days = np.shape(best_solution)
    PDj = [7, 9, 8, 9, 10, 7, 6]  # Demanda para cada día de la semana
    from itertools import cycle
    penalties = {
```

```
        "descansos_insuficientes": 0,
        "trabajos_consecutivos": 0,
        "limite_dias_trabajados": 0,
        "demanda_diaria": 0
    }
    longest_breaks = []
    dias_trabajados = 0
    workforce_diario = 0
    # Cálculo de dias totales trabajados
    dias_trabajados = np.sum(best_solution, axis=1)
    # Cálculo de los trabajadores diarios
    workforce_diario = np.sum(best_solution, axis=0)
    # Calculo de descansos
    for i in range(num_workers):
        breaks = []
        count = 0
        consecutive_workdays = 0
        for j in range(num_days):
            if best_solution[i][j] == 0:
                count += 1
                if consecutive_workdays > 5:
                    penalties["trabajos_consecutivos"] += 1
                consecutive_workdays = 0
            else:
                if count > 0:
                    breaks.append(count)
                count = 0
                consecutive_workdays += 1
        if count > 0:
            breaks.append(count)
        if breaks:
            longest_breaks.append(max(breaks))
            penalties["descansos_insuficientes"] += sum(1 for x in breaks if
x < 2)
        else:
            longest_breaks.append(0)
        if consecutive_workdays > 5:
            penalties["trabajos_consecutivos"] += 1
    if any(x > 63 for x in dias_trabajados):
        penalties["limite_dias_trabajados"] += 1
    PDj_cycle = cycle(PDj)
    for i in range(num_days):
        demanda_diaria = next(PDj_cycle)
        if workforce_diario[i] != demanda_diaria:
            penalties["demanda_diaria"] += 1

    print('El valor de la mejor solucion inicial era %d' %
best_solution_value_i)
    print('El valor de la mejor solucion es %d' %best_solution_value)
    end_time = time.time()
    tiempo_transcurrido = end_time - start_time
    print ('El numero de mutaciones ha sido ', contador_mutacion)
    print('\nEl proceso tardó %d segundos' % (tiempo_transcurrido), 'y la
mejor solucion es : \n ')
    print('\n'.join(imprimir_matriz(best_solution)))
    print('\nEl vector de descansos de cada trabajador es ',  longest_breaks)
    print('\nEl numero de penalizaciones incumplidas son:  ')
    for penalty_name, penalty_value in penalties.items():
        print(f"{penalty_name}: {penalty_value}")
    print('los trabajadores diarios son: ', workforce_diario)
```

```
    print('Los dias trabajados por cada empleado son:', dias_trabajados)
if __name__ == "__main__":
    main()
```

[2] Experimento 2:

```
# módulos de Python que vamos a utilizar
import random
import numpy as np
import time

import warnings
warnings.filterwarnings("ignore")



# paso1: función de evaluación / fitness function
def evalua(individuo):
    num_workers, num_days = np.shape (individuo)
    PDj = [7, 9, 8, 9, 10, 7, 6]  # Demanda para cada día de la semana
    from itertools import cycle
    penalties = {
        "descansos_insuficientes": 0,
        "trabajos_consecutivos": 0,
        "limite_dias_trabajados": 0,
        "demanda_diaria": 0
    }
    longest_breaks = []
    dias_trabajados = 0
    workforce_diario = 0
    #Cálculo de dias totales trabajados
    dias_trabajados = np.sum(individuo, axis = 1)
    #Cálculo de los trabajadores diarios
    workforce_diario = np.sum(individuo, axis = 0)
    #Calculo de descansos
    for i in range(num_workers):
        breaks = []
        count = 0
        consecutive_workdays = 0
        for j in range(num_days):
            if individuo[i][j] == 0:
                count += 1
                if consecutive_workdays > 5:
                    penalties["trabajos_consecutivos"] += 1
                consecutive_workdays = 0
            else:
                if count > 0:
                    breaks.append(count)
                count = 0
                consecutive_workdays += 1
        if count > 0:
            breaks.append(count)
        if breaks:
            longest_breaks.append(max(breaks))
            penalties["descansos_insuficientes"] += sum(1 for x in breaks if
x < 2)
        else:
            longest_breaks.append(0)
```

```
            if consecutive_workdays > 5:
                penalties["trabajos_consecutivos"] += 1

        if any(x>63 for x in dias_trabajados):
            penalties["limite_dias_trabajados"] +=1

        PDj_cycle = cycle(PDj)
        for i in range(num_days):
            demanda_diaria = next(PDj_cycle)
            if workforce_diario[i] != demanda_diaria:
                penalties["demanda_diaria"] += 1


        objective1 = sum (longest_breaks)
        objective2 = (max (longest_breaks) - min(longest_breaks))
        total_penalty = sum(penalties.values())
        fitness = int(objective1) - int(objective2) - int(total_penalty)
        return (fitness)


#imprime la matriz con los descansos marcados en verde (solo visualizacion)
def imprimir_matriz (matriz) :
    s = [[str(e) for e in row] for row in matriz]
    lens = [max(map(len, col)) for col in zip(*s)]
    formatted_rows = []
    for row in matriz:
        formatted_row = []
        for cell in row:
            # Si el valor de la celda es 0, cambia el color del texto a verde
            if cell == 0:
                formatted_cell = '\x1b[32m{}\x1b[0m'.format(cell)
            else:
                formatted_cell = str(cell)
            formatted_row.append(formatted_cell)
        formatted_rows.append(formatted_row)
    fmt = '\t'.join('{{:{}}}'.format(x) for x in lens)
    table = [fmt.format(*row) for row in formatted_rows]
    return table

#Generacion de una poblacion aleatoria
def generate_population(population_size, num_rows, num_cols):
    population = []
    random.seed(64)
    for _ in range(population_size):
        # Genera una matriz binaria aleatoria
        individual = np.random.randint(2, size=(num_rows, num_cols))
        population.append(individual)
    return population




def main():
    random.seed(64) # semilla del generador de números aleatorios
    start_time = time.time()
    p_cross = 0.8
    p_mutation = 0.3
    n_iteration = 100000
    size_population = 300
```

```
    pop = generate_population(size_population,14,98)

    population_evaluated = list(zip(map(lambda gen: evalua(gen), pop), pop))


    best_solution_value_i, best_solution_i = max(population_evaluated,
key=lambda x: x[0])
    best_solution_value = best_solution_value_i
    best_solution = best_solution_i
    contador_mutacion = 0
    # Comenzamos la evolución:
    for i in range(n_iteration):
        # Selection - Tournament
        size_tournament = 3
        selec1 = np.random.permutation(size_population)[0:size_tournament]
        selec2 = np.random.permutation(size_population)[0:size_tournament]

        value1, ind1 = max((population_evaluated[selec1[i]][0], selec1[i])
for i in range(size_tournament))
        value2, ind2 = max((population_evaluated[selec2[i]][0], selec2[i])
for i in range(size_tournament))


        if np.random.rand() <= p_cross:
            # Generar un único punto de corte en la mitad de la matriz
            parent1 = pop[ind1]
            parent2 = pop[ind2]
            cross_point = len(parent1) // 2

            descendent1 = np.concatenate((parent1[:, :cross_point],
parent2[:, cross_point:]), axis=1)
            descendent2 = np.concatenate((parent2[:, :cross_point],
parent1[:, cross_point:]), axis=1)
        else:
            descendent1 = pop[ind1]
            descendent2 = pop[ind2]

        if np.random.rand() <= p_mutation:
            # Elegir un índice de fila y columna al azar para la mutación
            mutation_indices = np.random.randint(0, descendent1.size,
size=(3, 2))
            contador_mutacion +=1
            for row_idx, col_idx in mutation_indices:
                row = row_idx % descendent1.shape[0]
                col = col_idx % descendent1.shape[1]

                if descendent1[row, col] == 1:
                    # Cambiar de 1 a 0 en la posición seleccionada
                    descendent1[row, col] = 0

                    # Encontrar los índices de elementos 0 en la misma
columna
                    zero_indices = np.where(descendent1[:, col] == 0)[0]

                    # Elegir al azar un índice de elemento 0 para cambiar a 1
                    if zero_indices.size > 0:
                        idx_to_change = np.random.choice(zero_indices)
                        descendent1[idx_to_change, col] = 1

        if np.random.rand() <= p_mutation:
            # Elegir un índice de fila y columna al azar para la mutación
```

```
            mutation_indices = np.random.randint(0, descendent2.size,
size=(3, 2))
            contador_mutacion +=1
            for row_idx, col_idx in mutation_indices:
                row = row_idx % descendent2.shape[0]
                col = col_idx % descendent2.shape[1]

                if descendent2[row, col] == 1:
                    # Cambiar de 1 a 0 en la posición seleccionada
                    descendent2[row, col] = 0

                    # Encontrar los índices de elementos 0 en la misma
columna
                    zero_indices = np.where(descendent2[:, col] == 0)[0]

                    # Elegir al azar un índice de elemento 0 para cambiar a 1
                    if zero_indices.size > 0:
                        idx_to_change = np.random.choice(zero_indices)
                        descendent2[idx_to_change, col] = 1

        descendent1_value = evalua(descendent1)
        descendent2_value = evalua(descendent2)

        if descendent1_value > best_solution_value:
            best_solution_value = descendent1_value
            best_solution = descendent1

        if descendent2_value > best_solution_value:
            best_solution_value = descendent2_value
            best_solution = descendent2

            # Selection - Reject
        size_tournament = 3
        selec1 = np.random.permutation(size_population)[0:size_tournament]
        selec2 = np.random.permutation(size_population)[0:size_tournament]

        _, ind1 = min((population_evaluated[selec1[i]][0], selec1[i]) for i
in range(size_tournament))
        _, ind2 = min((population_evaluated[selec2[i]][0], selec2[i]) for i
in range(size_tournament))

        population_evaluated[ind1] = descendent1_value, descendent1
        population_evaluated[ind2] = descendent2_value, descendent2



        tiempo_iteracion = time.time()
        print("Iteración:", i)
        print("Mejor solución:", best_solution_value)
        print("Tiempo transcurrido:", tiempo_iteracion-start_time)
        print("Tamaño de la población:", len(population_evaluated))

    #Evaluacion de la solucion final:
    num_workers, num_days = np.shape(best_solution)
    PDj = [7, 9, 8, 9, 10, 7, 6]  # Demanda para cada día de la semana
    from itertools import cycle
    penalties = {
        "descansos_insuficientes": 0,
        "trabajos_consecutivos": 0,
        "limite_dias_trabajados": 0,
```

```
        "demanda_diaria": 0
    }
    longest_breaks = []
    dias_trabajados = 0
    workforce_diario = 0
    # Cálculo de dias totales trabajados
    dias_trabajados = np.sum(best_solution, axis=1)
    # Cálculo de los trabajadores diarios
    workforce_diario = np.sum(best_solution, axis=0)
    # Calculo de descansos
    for i in range(num_workers):
        breaks = []
        count = 0
        consecutive_workdays = 0
        for j in range(num_days):
            if best_solution[i][j] == 0:
                count += 1
                if consecutive_workdays > 5:
                    penalties["trabajos_consecutivos"] += 1
                consecutive_workdays = 0
            else:
                if count > 0:
                    breaks.append(count)
                count = 0
                consecutive_workdays += 1
        if count > 0:
            breaks.append(count)
        if breaks:
            longest_breaks.append(max(breaks))
            penalties["descansos_insuficientes"] += sum(1 for x in breaks if
x < 2)
        else:
            longest_breaks.append(0)
        if consecutive_workdays > 5:
            penalties["trabajos_consecutivos"] += 1
    if any(x > 63 for x in dias_trabajados):
        penalties["limite_dias_trabajados"] += 1
    PDj_cycle = cycle(PDj)
    for i in range(num_days):
        demanda_diaria = next(PDj_cycle)
        if workforce_diario[i] != demanda_diaria:
            penalties["demanda_diaria"] += 1

    print('El valor de la mejor solucion inicial era %d' %
best_solution_value_i)
    print('El valor de la mejor solucion es %d' %best_solution_value)
    end_time = time.time()
    tiempo_transcurrido = end_time - start_time
    print ('El numero de mutaciones ha sido ', contador_mutacion)
    print('\nEl proceso tardó %d segundos' % (tiempo_transcurrido), 'y la
mejor solucion es : \n ')
    print('\n'.join(imprimir_matriz(best_solution)))
    print('\nEl vector de descansos de cada trabajador es ',  longest_breaks)
    print('\nEl numero de penalizaciones incumplidas son:  ')
    for penalty_name, penalty_value in penalties.items():
        print(f"{penalty_name}: {penalty_value}")
    print('los trabajadores diarios son: ', workforce_diario)
    print('Los dias trabajados por cada empleado son:', dias_trabajados)
if __name__ == "__main__":
    main()
```

[3] Experimento 2, solucion inicial no aleatoria

```
# módulos de Python que vamos a utilizar
import random
import array
import numpy as np
import time
from deap import base
from deap import creator
from deap import tools
from itertools import cycle

import warnings
warnings.filterwarnings("ignore")

# paso1: creación del problema
creator.create("Problema2", base.Fitness, weights=(1.0 ,))

# paso2: creación del individuo

creator.create("Individuo", array.array, typecode= "I",
fitness=creator.Problema2)

toolbox = base.Toolbox() # creamos la caja de herramientas


# paso3: función de evaluación / fitness function
def evalua(individuo):
    num_workers, num_days = np.shape (individuo)
    PDj = [7, 9, 8, 9, 10, 7, 6]  # Demanda para cada día de la semana
    from itertools import cycle
    penalties = {
        "descansos_insuficientes": 0,
        "trabajos_consecutivos": 0,
        "limite_dias_trabajados": 0,
        "demanda_diaria": 0
    }
    longest_breaks = []
    dias_trabajados = 0
    workforce_diario = 0
    #Cálculo de dias totales trabajados
    dias_trabajados = np.sum(individuo, axis = 1)
    #Cálculo de los trabajadores diarios
    workforce_diario = np.sum(individuo, axis = 0)
    #Calculo de descansos
    for i in range(num_workers):
        breaks = []
        count = 0
        consecutive_workdays = 0
        for j in range(num_days):
            if individuo[i][j] == 0:
                count += 1
                if consecutive_workdays > 5:
```

```
                    penalties["trabajos_consecutivos"] += 1
                consecutive_workdays = 0
            else:
                if count > 0:
                    breaks.append(count)
                count = 0
                consecutive_workdays += 1
        if count > 0:
            breaks.append(count)
        if breaks:
            longest_breaks.append(max(breaks))
            penalties["descansos_insuficientes"] += sum(1 for x in breaks if
x < 2)
        else:
            longest_breaks.append(0)

        if consecutive_workdays > 5:
            penalties["trabajos_consecutivos"] += 1

    if any(x>63 for x in dias_trabajados):
        penalties["limite_dias_trabajados"] +=1

    PDj_cycle = cycle(PDj)
    for i in range(num_days):
        demanda_diaria = next(PDj_cycle)
        if workforce_diario[i] != demanda_diaria:
            penalties["demanda_diaria"] += 2


    objective1 = sum (longest_breaks)
    objective2 = (max (longest_breaks) - min(longest_breaks))
    total_penalty = sum(penalties.values())
    fitness = int(objective1) - int(objective2) - int(total_penalty)
    return (fitness)


#imprime la matriz con los descansos marcados en verde (solo visualizacion)
def imprimir_matriz (matriz) :
    s = [[str(e) for e in row] for row in matriz]
    lens = [max(map(len, col)) for col in zip(*s)]
    formatted_rows = []
    for row in matriz:
        formatted_row = []
        for cell in row:
            # Si el valor de la celda es 0, cambia el color del texto a verde
            if cell == 0:
                formatted_cell = '\x1b[32m{}\x1b[0m'.format(cell)
            else:
                formatted_cell = str(cell)
            formatted_row.append(formatted_cell)
        formatted_rows.append(formatted_row)
    fmt = '\t'.join('{{:{}}}'.format(x) for x in lens)
    table = [fmt.format(*row) for row in formatted_rows]
    return table

#Generacion de una poblacion con exactamente el numero de 1s de la demanda
def generate_population(population_size, num_rows, num_cols):
    population = []
    num_ones_per_individual = 784  # Número deseado de 1s en cada individuo

    for _ in range(population_size):
        individual = np.zeros((num_rows, num_cols), dtype=int)
```

```
        # Establecer aleatoriamente los 1s en el individuo
        indices = np.random.choice(num_rows * num_cols,
num_ones_per_individual, replace=False)
        row_indices, col_indices = np.unravel_index(indices, (num_rows,
num_cols))
        individual[row_indices, col_indices] = 1

        population.append(individual)

    return population



# Operaciones genéticas


def main():
    random.seed(64) # semilla del generador de números aleatorios
    start_time = time.time()
    p_cross = 0.8
    p_mutation = 0.3
    n_iteration = 700000
    size_population = 300

    pop = generate_population(size_population,14,98)

    population_evaluated = list(zip(map(lambda gen: evalua(gen), pop), pop))


    best_solution_value_i, best_solution_i = max(population_evaluated,
key=lambda x: x[0])
    best_solution_value = best_solution_value_i
    best_solution = best_solution_i
    contador_mutacion = 0
    # Comenzamos la evolución:
    for i in range(n_iteration):
        # Selection - Tournament
        size_tournament = 3
        selec1 = np.random.permutation(size_population)[0:size_tournament]
        selec2 = np.random.permutation(size_population)[0:size_tournament]

        value1, ind1 = max((population_evaluated[selec1[i]][0], selec1[i])
for i in range(size_tournament))
        value2, ind2 = max((population_evaluated[selec2[i]][0], selec2[i])
for i in range(size_tournament))


        if np.random.rand() <= p_cross:
            # Generar un único punto de corte en la mitad de la matriz
            parent1 = pop[ind1]
            parent2 = pop[ind2]
            cross_point = len(parent1) // 2

            descendent1 = np.concatenate((parent1[:, :cross_point],
parent2[:, cross_point:]), axis=1)
            descendent2 = np.concatenate((parent2[:, :cross_point],
parent1[:, cross_point:]), axis=1)
        else:
            descendent1 = pop[ind1]
```

```
            descendent2 = pop[ind2]

        if np.random.rand() <= p_mutation:
            # Elegir un índice de fila y columna al azar para la mutación
            mutation_indices = np.random.randint(0, descendent1.size,
size=(3, 2))
            contador_mutacion +=1
            for row_idx, col_idx in mutation_indices:
                row = row_idx % descendent1.shape[0]
                col = col_idx % descendent1.shape[1]

                if descendent1[row, col] == 1:
                    # Cambiar de 1 a 0 en la posición seleccionada
                    descendent1[row, col] = 0

        if np.random.rand() <= p_mutation:
            # Elegir un índice de fila y columna al azar para la mutación
            mutation_indices = np.random.randint(0, descendent2.size,
size=(3, 2))
            contador_mutacion +=1
            for row_idx, col_idx in mutation_indices:
                row = row_idx % descendent2.shape[0]
                col = col_idx % descendent2.shape[1]


        descendent1_value = evalua(descendent1)
        descendent2_value = evalua(descendent2)

        if descendent1_value > best_solution_value:
            best_solution_value = descendent1_value
            best_solution = descendent1

        if descendent2_value > best_solution_value:
            best_solution_value = descendent2_value
            best_solution = descendent2

            # Selection - Reject
        size_tournament = 3
        selec1 = np.random.permutation(size_population)[0:size_tournament]
        selec2 = np.random.permutation(size_population)[0:size_tournament]

        _, ind1 = min((population_evaluated[selec1[i]][0], selec1[i]) for i
in range(size_tournament))
        _, ind2 = min((population_evaluated[selec2[i]][0], selec2[i]) for i
in range(size_tournament))

        population_evaluated[ind1] = descendent1_value, descendent1
        population_evaluated[ind2] = descendent2_value, descendent2

        # Elitismo: reemplazar los peores individuos por los mejores de la
generación anterior
        population_evaluated.sort(key=lambda x: x[0], reverse=True)
        population_evaluated[-2:] = [(best_solution_value, best_solution)] *
2


        tiempo_iteracion = time.time()
        print("Iteración:", i)
        print("Mejor solución:", best_solution_value)
        print("Tiempo transcurrido:", tiempo_iteracion-start_time)
        print("Tamaño de la población:", len(population_evaluated))
```

```
    #Evaluacion de la solucion final:
    num_workers, num_days = np.shape(best_solution)
    PDj = [7, 9, 8, 9, 10, 7, 6]  # Demanda para cada día de la semana
    from itertools import cycle
    penalties = {
        "descansos_insuficientes": 0,
        "trabajos_consecutivos": 0,
        "limite_dias_trabajados": 0,
        "demanda_diaria": 0
    }
    longest_breaks = []
    dias_trabajados = 0
    workforce_diario = 0
    # Cálculo de dias totales trabajados
    dias_trabajados = np.sum(best_solution, axis=1)
    # Cálculo de los trabajadores diarios
    workforce_diario = np.sum(best_solution, axis=0)
    # Calculo de descansos
    for i in range(num_workers):
        breaks = []
        count = 0
        consecutive_workdays = 0
        for j in range(num_days):
            if best_solution[i][j] == 0:
                count += 1
                if consecutive_workdays > 5:
                    penalties["trabajos_consecutivos"] += 1
                consecutive_workdays = 0
            else:
                if count > 0:
                    breaks.append(count)
                count = 0
                consecutive_workdays += 1
        if count > 0:
            breaks.append(count)
        if breaks:
            longest_breaks.append(max(breaks))
            penalties["descansos_insuficientes"] += sum(1 for x in breaks if
x < 2)
        else:
            longest_breaks.append(0)
        if consecutive_workdays > 5:
            penalties["trabajos_consecutivos"] += 1
    if any(x > 63 for x in dias_trabajados):
        penalties["limite_dias_trabajados"] += 1
    PDj_cycle = cycle(PDj)
    for i in range(num_days):
        demanda_diaria = next(PDj_cycle)
        if workforce_diario[i] != demanda_diaria:
            penalties["demanda_diaria"] += 2

    print('El valor de la mejor solucion inicial era %d' %
best_solution_value_i)
    print('El valor de la mejor solucion es %d' %best_solution_value)
    end_time = time.time()
    tiempo_transcurrido = end_time - start_time
    print ('El numero de mutaciones ha sido ', contador_mutacion)
    print('\nEl proceso tardó %d segundos' % (tiempo_transcurrido), 'y la
mejor solucion es : \n ')
    print('\n'.join(imprimir_matriz(best_solution)))
```

```
    print('\nEl vector de descansos de cada trabajador es ',  longest_breaks)
    print('\nEl numero de penalizaciones incumplidas son:  ')
    for penalty_name, penalty_value in penalties.items():
        print(f"{penalty_name}: {penalty_value}")
    print('los trabajadores diarios son: ', workforce_diario)
    print('Los dias trabajados por cada empleado son:', dias_trabajados)
if __name__ == "__main__":
    main()
```

[4] Experimento 3

```
# módulos de Python que vamos a utilizar
import random
import array
import numpy as np
import time
from deap import base
from deap import creator
from deap import tools
from itertools import cycle

import warnings
warnings.filterwarnings("ignore")

# paso1: creación del problema
creator.create("Problema2", base.Fitness, weights=(1.0 ,))

# paso2: creación del individuo

creator.create("Individuo", array.array, typecode= "I",
fitness=creator.Problema2)

toolbox = base.Toolbox() # creamos la caja de herramientas


# paso3: función de evaluación / fitness function
def evalua(individuo):
    num_workers, num_days = np.shape (individuo)
    PDj = [7, 9, 8, 9, 10, 7, 6]  # Demanda para cada día de la semana
    from itertools import cycle
    penalties = {
        "descansos_insuficientes": 0,
        "trabajos_consecutivos": 0,
        "limite_dias_trabajados": 0,
        "demanda_diaria": 0
    }
    longest_breaks = []
    dias_trabajados = 0
    workforce_diario = 0
    #Cálculo de dias totales trabajados
    dias_trabajados = np.sum(individuo, axis = 1)
    #Cálculo de los trabajadores diarios
    workforce_diario = np.sum(individuo, axis = 0)
    #Calculo de descansos
    for i in range(num_workers):
        breaks = []
        count = 0
        consecutive_workdays = 0
```

```
        for j in range(num_days):
            if individuo[i][j] == 0:
                count += 1
                if consecutive_workdays > 5:
                    penalties["trabajos_consecutivos"] += 1
                consecutive_workdays = 0
            else:
                if count > 0:
                    breaks.append(count)
                count = 0
                consecutive_workdays += 1
        if count > 0:
            breaks.append(count)
        if breaks:
            longest_breaks.append(max(breaks))
            penalties["descansos_insuficientes"] += sum(1 for x in breaks if
x < 2)
        else:
            longest_breaks.append(0)

        if consecutive_workdays > 5:
            penalties["trabajos_consecutivos"] += 1

    if any(x>63 for x in dias_trabajados):
        penalties["limite_dias_trabajados"] +=1

    PDj_cycle = cycle(PDj)
    for i in range(num_days):
        demanda_diaria = next(PDj_cycle)
        if workforce_diario[i] != demanda_diaria:
            penalties["demanda_diaria"] += 1


    objective1 = sum (longest_breaks)
    objective2 = (max (longest_breaks) - min(longest_breaks))
    total_penalty = sum(penalties.values())
    fitness = int(objective1) - int(objective2) - int(total_penalty)
    return (fitness)


#imprime la matriz con los descansos marcados en verde (solo visualizacion)
def imprimir_matriz (matriz) :
    s = [[str(e) for e in row] for row in matriz]
    lens = [max(map(len, col)) for col in zip(*s)]
    formatted_rows = []
    for row in matriz:
        formatted_row = []
        for cell in row:
            # Si el valor de la celda es 0, cambia el color del texto a verde
            if cell == 0:
                formatted_cell = '\x1b[32m{}\x1b[0m'.format(cell)
            else:
                formatted_cell = str(cell)
            formatted_row.append(formatted_cell)
        formatted_rows.append(formatted_row)
    fmt = '\t'.join('{{:{}}}'.format(x) for x in lens)
    table = [fmt.format(*row) for row in formatted_rows]
    return table

#Generacion de una poblacion con exactamente el numero de 1s de la demanda
```

```
def generate_population(population_size, num_rows, num_cols):
    population = []
    num_ones_per_individual = 784  # Número deseado de 1s en cada individuo

    for _ in range(population_size):
        individual = np.zeros((num_rows, num_cols), dtype=int)

        # Establecer aleatoriamente los 1s en el individuo
        indices = np.random.choice(num_rows * num_cols,
num_ones_per_individual, replace=False)
        row_indices, col_indices = np.unravel_index(indices, (num_rows,
num_cols))
        individual[row_indices, col_indices] = 1

        population.append(individual)

    return population



# Operaciones genéticas

def multi_point_crossover(parent1, parent2, num_points):
    # Asegúrate de que num_points no sea mayor que la longitud de los
cromosomas
    num_points = min(num_points, len(parent1) - 1)

    # Genera num_points puntos de corte distintos
    cut_points = sorted(random.sample(range(1, len(parent1)), num_points))

    child1 = np.copy(parent1)
    child2 = np.copy(parent2)
    last_cut = 0

    for cut in cut_points:
        if (cut - last_cut) % 2 == 0:
            # Si la distancia entre cortes es par, intercambia los segmentos
entre los cortes
            child1[last_cut:cut], child2[last_cut:cut] =
child2[last_cut:cut], child1[last_cut:cut]
        else:
            # Si la distancia entre cortes es impar, intercambia los
segmentos después del corte
            child1[cut], child2[cut] = child2[cut], child1[cut]

        last_cut = cut

    return child1, child2
def mutate_individual(matrix):

    # Choose 3 random elements in the matrix
    indices = random.sample(range(matrix.size), 3)
    row_indices, col_indices = np.unravel_index(indices, matrix.shape)

        # Transform the chosen elements into their contrary
    matrix[row_indices, col_indices] = 1 - matrix[row_indices, col_indices]

        # Find an element contrary to the permuted one in the column
    for col_index in col_indices:
        col = matrix[:, col_index]
        contrary_element = 1 - matrix[row_indices[0], col_index]
        contrary_indices = np.where(col == contrary_element)[0]
```

```
        # Choose a random contrary element and transform it
        if contrary_indices.size > 0:
            random_index = random.choice(contrary_indices)
            matrix[random_index, col_index] = 1 - matrix[random_index,
col_index]

    return matrix

def main():
    random.seed(64) # semilla del generador de números aleatorios
    start_time = time.time()
    p_cross = 0.8
    p_mutation = 0.5
    n_iteration = 10000
    size_population = 100

    pop = generate_population(size_population,14,98)

    population_evaluated = list(zip(map(lambda gen: evalua(gen), pop), pop))


    best_solution_value_i, best_solution_i = max(population_evaluated,
key=lambda x: x[0])
    best_solution_value = best_solution_value_i
    best_solution = best_solution_i
    contador_mutacion = 0
    # Comenzamos la evolución:
    for i in range(n_iteration):
        # Selection - Tournament
        size_tournament = 3
        selec1 = np.random.permutation(size_population)[0:size_tournament]
        selec2 = np.random.permutation(size_population)[0:size_tournament]

        value1, ind1 = max((population_evaluated[selec1[i]][0], selec1[i])
for i in range(size_tournament))
        value2, ind2 = max((population_evaluated[selec2[i]][0], selec2[i])
for i in range(size_tournament))

        if np.random.rand() <= p_cross:
            parent1 = pop[ind1]
            parent2 = pop[ind2]
            num_crossover_points = 3  # Puedes ajustar este valor según tus
necesidades
            descendent1, descendent2 = multi_point_crossover(parent1,
parent2, num_crossover_points)

        else:
            descendent1 = pop[ind1]
            descendent2 = pop[ind2]

        if np.random.rand() <= p_mutation:
            contador_mutacion += 1
            # Elegir un índice de individuo al azar para la mutación
            mutation_idx = ind1
            descendent1 = mutate_individual(pop[mutation_idx])
            pop[mutation_idx] = descendent1

        if np.random.rand() <= p_mutation:
            contador_mutacion += 1
```

```
            # Elegir un índice de individuo al azar para la mutación
            mutation_idx = ind2
            descendent2 = mutate_individual(pop[mutation_idx])
            pop[mutation_idx] = descendent2

        descendent1_value = evalua(descendent1)
        descendent2_value = evalua(descendent2)

        if descendent1_value > best_solution_value:
            best_solution_value = descendent1_value
            best_solution = descendent1

        if descendent2_value > best_solution_value:
            best_solution_value = descendent2_value
            best_solution = descendent2

            # Selection - Reject
        size_tournament = 3
        selec1 = np.random.permutation(size_population)[0:size_tournament]
        selec2 = np.random.permutation(size_population)[0:size_tournament]

        _, ind1 = min((population_evaluated[selec1[i]][0], selec1[i]) for i
in range(size_tournament))
        _, ind2 = min((population_evaluated[selec2[i]][0], selec2[i]) for i
in range(size_tournament))

        population_evaluated[ind1] = descendent1_value, descendent1
        population_evaluated[ind2] = descendent2_value, descendent2

        # Elitismo: reemplazar los peores individuos por los mejores de la
generación anterior
        population_evaluated.sort(key=lambda x: x[0], reverse=True)
        population_evaluated[-2:] = [(best_solution_value, best_solution)] *
2



        tiempo_iteracion = time.time()
        print("Iteración:", i)
        print("Mejor solución:", best_solution_value)
        print("Tiempo transcurrido:", tiempo_iteracion-start_time)
        print("Tamaño de la población:", len(population_evaluated))

    #Evaluacion de la solucion final:
    num_workers, num_days = np.shape(best_solution)
    PDj = [7, 9, 8, 9, 10, 7, 6]  # Demanda para cada día de la semana
    from itertools import cycle
    penalties = {
        "descansos_insuficientes": 0,
        "trabajos_consecutivos": 0,
        "limite_dias_trabajados": 0,
        "demanda_diaria": 0
    }
    longest_breaks = []
    dias_trabajados = 0
    workforce_diario = 0
    # Cálculo de dias totales trabajados
    dias_trabajados = np.sum(best_solution, axis=1)
    # Cálculo de los trabajadores diarios
    workforce_diario = np.sum(best_solution, axis=0)
    # Calculo de descansos
    for i in range(num_workers):
        breaks = []
```

```
        count = 0
        consecutive_workdays = 0
        for j in range(num_days):
            if best_solution[i][j] == 0:
                count += 1
                if consecutive_workdays > 5:
                    penalties["trabajos_consecutivos"] += 1
                consecutive_workdays = 0
            else:
                if count > 0:
                    breaks.append(count)
                count = 0
                consecutive_workdays += 1
        if count > 0:
            breaks.append(count)
        if breaks:
            longest_breaks.append(max(breaks))
            penalties["descansos_insuficientes"] += sum(1 for x in breaks if
x < 2)
        else:
            longest_breaks.append(0)
        if consecutive_workdays > 5:
            penalties["trabajos_consecutivos"] += 1
    if any(x > 63 for x in dias_trabajados):
        penalties["limite_dias_trabajados"] += 1
    PDj_cycle = cycle(PDj)
    for i in range(num_days):
        demanda_diaria = next(PDj_cycle)
        if workforce_diario[i] != demanda_diaria:
            penalties["demanda_diaria"] += 1

    print('El valor de la mejor solucion inicial era %d' %
best_solution_value_i)
    print('El valor de la mejor solucion es %d' %best_solution_value)
    end_time = time.time()
    tiempo_transcurrido = end_time - start_time
    print ('El numero de mutaciones ha sido ', contador_mutacion)
    print('\nEl proceso tardó %d segundos' % (tiempo_transcurrido), 'y la
mejor solucion es : \n ')
    print('\n'.join(imprimir_matriz(best_solution)))
    print('\nEl vector de descansos de cada trabajador es ',  longest_breaks)
    print('\nEl numero de penalizaciones incumplidas son:  ')
    for penalty_name, penalty_value in penalties.items():
        print(f"{penalty_name}: {penalty_value}")
    print('los trabajadores diarios son: ', workforce_diario)
    print('Los dias trabajados por cada empleado son:', dias_trabajados)
if __name__ == "__main__":
    main()
```

[4] Experimento 5

```
# módulos de Python que vamos a utilizar
import random
import array
import numpy as np
import time
from deap import base
from deap import creator
from deap import tools
from itertools import cycle

import warnings
warnings.filterwarnings("ignore")

# paso1: creación del problema
creator.create("Problema2", base.Fitness, weights=(1.0 ,))

# paso2: creación del individuo

creator.create("Individuo", array.array, typecode= "I",
fitness=creator.Problema2)

toolbox = base.Toolbox() # creamos la caja de herramientas


# paso3: función de evaluación / fitness function
def evalua(individuo):
    num_workers, num_days = np.shape (individuo)
    PDj = [7, 9, 8, 9, 10, 7, 6]  # Demanda para cada día de la semana
    from itertools import cycle
    penalties = {
        "descansos_insuficientes": 0,
        "trabajos_consecutivos": 0,
        "limite_dias_trabajados": 0,
        "demanda_diaria": 0
    }
    longest_breaks = []
    dias_trabajados = 0
    workforce_diario = 0
    #Cálculo de dias totales trabajados
    dias_trabajados = np.sum(individuo, axis = 1)
    #Cálculo de los trabajadores diarios
    workforce_diario = np.sum(individuo, axis = 0)
    #Calculo de descansos
    for i in range(num_workers):
        breaks = []
        count = 0
        consecutive_workdays = 0
        for j in range(num_days):
            if individuo[i][j] == 0:
                count += 1
                if consecutive_workdays > 5:
                    penalties["trabajos_consecutivos"] += 1
                consecutive_workdays = 0
```

```
            else:
                if count > 0:
                    breaks.append(count)
                count = 0
                consecutive_workdays += 1
        if count > 0:
            breaks.append(count)
        if breaks:
            longest_breaks.append(max(breaks))
            penalties["descansos_insuficientes"] += sum(1 for x in breaks if
x < 2)
        else:
            longest_breaks.append(0)

        if consecutive_workdays > 5:
            penalties["trabajos_consecutivos"] += 1

    if any(x>63 for x in dias_trabajados):
        penalties["limite_dias_trabajados"] +=1

    PDj_cycle = cycle(PDj)
    for i in range(num_days):
        demanda_diaria = next(PDj_cycle)
        if workforce_diario[i] != demanda_diaria:
            penalties["demanda_diaria"] += 2


    objective1 = sum (longest_breaks)
    objective2 = (max (longest_breaks) - min(longest_breaks))
    total_penalty = sum(penalties.values())
    fitness = objective1 - objective2 - total_penalty
    return (fitness)


#imprime la matriz con los descansos marcados en verde (solo visualizacion)
def imprimir_matriz (matriz) :
    s = [[str(e) for e in row] for row in matriz]
    lens = [max(map(len, col)) for col in zip(*s)]
    formatted_rows = []
    for row in matriz:
        formatted_row = []
        for cell in row:
            # Si el valor de la celda es 0, cambia el color del texto a verde
            if cell == 0:
                formatted_cell = '\x1b[32m{}\x1b[0m'.format(cell)
            else:
                formatted_cell = str(cell)
            formatted_row.append(formatted_cell)
        formatted_rows.append(formatted_row)
    fmt = '\t'.join('{{:{}}}'.format(x) for x in lens)
    table = [fmt.format(*row) for row in formatted_rows]
    return table

#Generacion de una poblacion con exactamente el numero de 1s de la demanda
def generate_population(population_size, num_rows, num_cols):
    population = []
    num_ones_per_individual = 784  # Número deseado de 1s en cada individuo

    for _ in range(population_size):
        individual = np.zeros((num_rows, num_cols), dtype=int)
```

```
        # Establecer aleatoriamente los 1s en el individuo
        indices = np.random.choice(num_rows * num_cols,
num_ones_per_individual, replace=False)
        row_indices, col_indices = np.unravel_index(indices, (num_rows,
num_cols))
        individual[row_indices, col_indices] = 1

        population.append(individual)

    return population



# Operaciones genéticas

def combine_crossovers(parent1, parent2, num_points, p3):
    if random.random() < p3:
        # Aplicar el crossover por filas
        return crossover_rows(parent1, parent2, num_points)
    else:
        # Aplicar el crossover por columnas
        return crossover_columns(parent1, parent2, num_points)

def crossover_rows(parent1, parent2, num_points):
    # Asegúrate de que num_points no sea mayor que la longitud de los
cromosomas
    num_points = min(num_points, len(parent1) - 1)

    # Genera num_points puntos de corte distintos
    cut_points = sorted(random.sample(range(1, len(parent1)), num_points))

    child1 = np.copy(parent1)
    child2 = np.copy(parent2)
    last_cut = 0

    for cut in cut_points:
        if (cut - last_cut) % 2 == 0:
            # Si la distancia entre cortes es par, intercambia los segmentos
entre los cortes
            child1[last_cut:cut], child2[last_cut:cut] =
child2[last_cut:cut], child1[last_cut:cut]
        else:
            # Si la distancia entre cortes es impar, intercambia los
segmentos después del corte
            child1[cut], child2[cut] = child2[cut], child1[cut]

        last_cut = cut

    return child1, child2


def crossover_columns(matrix1, matrix2, cut_points):
    # Get the number of columns in the matrices
    num_columns = matrix1.shape[1]

    # Generate random cut points
    cut_indices = np.random.choice(num_columns, size=cut_points,
replace=False)

    # Sort the cut points in ascending order
    cut_indices.sort()
```

```
    # Initialize child matrices
    child1 = np.zeros_like(matrix1)
    child2 = np.zeros_like(matrix2)

    # Perform crossover by columns
    for i in range(num_columns):
        if i in cut_indices:
            # Perform crossover at cut points
            child1[:, i] = matrix2[:, i]
            child2[:, i] = matrix1[:, i]
        else:
            # Copy columns from parent matrices
            child1[:, i] = matrix1[:, i]
            child2[:, i] = matrix2[:, i]

    return child1, child2

def mutate_individual(matrix):

    # Choose 3 random elements in the matrix
    indices = random.sample(range(matrix.size), 3)
    row_indices, col_indices = np.unravel_index(indices, matrix.shape)

        # Transform the chosen elements into their contrary
    matrix[row_indices, col_indices] = 1 - matrix[row_indices, col_indices]

        # Find an element contrary to the permuted one in the column
    for col_index in col_indices:
        col = matrix[:, col_index]
        contrary_element = 1 - matrix[row_indices[0], col_index]
        contrary_indices = np.where(col == contrary_element)[0]

        # Choose a random contrary element and transform it
        if contrary_indices.size > 0:
            random_index = random.choice(contrary_indices)
            matrix[random_index, col_index] = 1 - matrix[random_index,
col_index]

    return matrix


def prob_mutacion(iteration, max_iterations):
    # Define la probabilidad de mutación inicial y final
    initial_mutation_prob = 0.7  # Probabilidad de mutación alta al principio
    final_mutation_prob = 0.1    # Probabilidad de mutación baja al final

    # Calcula la probabilidad de mutación actual en función de la iteración
actual
    mutation_prob = initial_mutation_prob - (initial_mutation_prob -
final_mutation_prob) * (iteration / max_iterations)
    return mutation_prob

def main():
    random.seed(64) # semilla del generador de números aleatorios
    start_time = time.time()
    p_cross = 0.8
    p_mutation = 0.3
    n_iteration = 500000
    size_population = 100
```

```
    pop = generate_population(size_population,14,98)

    population_evaluated = list(zip(map(lambda gen: evalua(gen), pop), pop))


    best_solution_value_i, best_solution_i = max(population_evaluated,
key=lambda x: x[0])
    best_solution_value = best_solution_value_i
    best_solution = best_solution_i
    contador_mutacion = 0
    # Comenzamos la evolución:
    for i in range(n_iteration):
        # Selection - Tournament
        size_tournament = 3
        selec1 = np.random.permutation(size_population)[0:size_tournament]
        selec2 = np.random.permutation(size_population)[0:size_tournament]

        value1, ind1 = max((population_evaluated[selec1[i]][0], selec1[i])
for i in range(size_tournament))
        value2, ind2 = max((population_evaluated[selec2[i]][0], selec2[i])
for i in range(size_tournament))

        if np.random.rand() <= p_cross:
            parent1 = pop[ind1]
            parent2 = pop[ind2]
            num_crossover_points = 2  # Puedes ajustar este valor según tus
necesidades
            descendent1, descendent2 = combine_crossovers(parent1, parent2,
num_crossover_points, 0.5)

        else:
            descendent1 = pop[ind1]
            descendent2 = pop[ind2]

        if np.random.rand() <= prob_mutacion(i,n_iteration):
            contador_mutacion += 1
            # Elegir un índice de individuo al azar para la mutación
            mutation_idx = ind1
            descendent1 = mutate_individual(pop[mutation_idx])
            pop[mutation_idx] = descendent1

        if np.random.rand() <= prob_mutacion(i,n_iteration):
            contador_mutacion += 1
            # Elegir un índice de individuo al azar para la mutación
            mutation_idx = ind2
            descendent2 = mutate_individual(pop[mutation_idx])
            pop[mutation_idx] = descendent2

        descendent1_value = evalua(descendent1)
        descendent2_value = evalua(descendent2)

        if descendent1_value > best_solution_value:
            best_solution_value = descendent1_value
            best_solution = descendent1

        if descendent2_value > best_solution_value:
            best_solution_value = descendent2_value
            best_solution = descendent2

            # Selection - Reject
        size_tournament = 3
```

```
        selec1 = np.random.permutation(size_population)[0:size_tournament]
        selec2 = np.random.permutation(size_population)[0:size_tournament]

        _, ind1 = min((population_evaluated[selec1[i]][0], selec1[i]) for i
in range(size_tournament))
        _, ind2 = min((population_evaluated[selec2[i]][0], selec2[i]) for i
in range(size_tournament))

        population_evaluated[ind1] = descendent1_value, descendent1
        population_evaluated[ind2] = descendent2_value, descendent2

        # Elitismo: reemplazar los peores individuos por los mejores de la
generación anterior
        population_evaluated.sort(key=lambda x: x[0], reverse=True)
        population_evaluated[-2:] = [(best_solution_value, best_solution)] *
2



        tiempo_iteracion = time.time()
        print("Iteración:", i)
        print("Mejor solución:", best_solution_value)
        print("Tiempo transcurrido:", tiempo_iteracion-start_time)
        print("Tamaño de la población:", len(population_evaluated))

    #Evaluacion de la solucion final:
    num_workers, num_days = np.shape(best_solution)
    PDj = [7, 9, 8, 9, 10, 7, 6]  # Demanda para cada día de la semana
    from itertools import cycle
    penalties = {
        "descansos_insuficientes": 0,
        "trabajos_consecutivos": 0,
        "limite_dias_trabajados": 0,
        "demanda_diaria": 0
    }
    longest_breaks = []
    dias_trabajados = 0
    workforce_diario = 0
    # Cálculo de dias totales trabajados
    dias_trabajados = np.sum(best_solution, axis=1)
    # Cálculo de los trabajadores diarios
    workforce_diario = np.sum(best_solution, axis=0)
    # Calculo de descansos
    for i in range(num_workers):
        breaks = []
        count = 0
        consecutive_workdays = 0
        for j in range(num_days):
            if best_solution[i][j] == 0:
                count += 1
                if consecutive_workdays > 5:
                    penalties["trabajos_consecutivos"] += 1
                consecutive_workdays = 0
            else:
                if count > 0:
                    breaks.append(count)
                count = 0
                consecutive_workdays += 1
        if count > 0:
            breaks.append(count)
        if breaks:
```

```
            longest_breaks.append(max(breaks))
            penalties["descansos_insuficientes"] += sum(1 for x in breaks if
x < 2)
        else:
            longest_breaks.append(0)
        if consecutive_workdays > 5:
            penalties["trabajos_consecutivos"] += 1
    if any(x > 63 for x in dias_trabajados):
        penalties["limite_dias_trabajados"] += 1
    PDj_cycle = cycle(PDj)
    for i in range(num_days):
        demanda_diaria = next(PDj_cycle)
        if workforce_diario[i] != demanda_diaria:
            penalties["demanda_diaria"] += 2

    print('El valor de la mejor solucion inicial era %d' %
best_solution_value_i)
    print('El valor de la mejor solucion es %d' %best_solution_value)
    end_time = time.time()
    tiempo_transcurrido = end_time - start_time
    print ('El numero de mutaciones ha sido ', contador_mutacion)
    print('\nEl proceso tardó %d segundos' % (tiempo_transcurrido), 'y la
mejor solucion es : \n ')
    print('\n'.join(imprimir_matriz(best_solution)))
    print('\nEl vector de descansos de cada trabajador es ',  longest_breaks)
    print('La suma total de los descansos es de', sum(longest_breaks))
    print('La diferencia entre descansos maximos y minimos es de ',
max(longest_breaks) - min(longest_breaks))
    print('La penalizacion total es ', sum(penalties.values()))
    print('\nEl numero de penalizaciones incumplidas son:  ')
    for penalty_name, penalty_value in penalties.items():
        print(f"{penalty_name}: {penalty_value}")
    print('los trabajadores diarios son: ', workforce_diario)
    print('Los dias trabajados por cada empleado son:', dias_trabajados)
if __name__ == "__main__":
    main()
```

[5] Experimento 6

```
# módulos de Python que vamos a utilizar
import random
import array
import numpy as np
import time
import matplotlib.pyplot as plt

import warnings
warnings.filterwarnings("ignore")


# paso3: función de evaluación / fitness function
def evalua(individuo):
    num_workers, num_days = np.shape (individuo)
    PDj = [7, 9, 8, 9, 10, 7, 6]  # Demanda para cada día de la semana
    from itertools import cycle
    penalties = {
        "descansos_insuficientes": 0,
        "trabajos_consecutivos": 0,
        "limite_dias_trabajados": 0,
        "demanda_diaria": 0
```

```
    }
    longest_breaks = []
    dias_trabajados = 0
    workforce_diario = 0
    #Cálculo de dias totales trabajados
    dias_trabajados = np.sum(individuo, axis = 1)
    #Cálculo de los trabajadores diarios
    workforce_diario = np.sum(individuo, axis = 0)
    #Calculo de descansos
    for i in range(num_workers):
        breaks = []
        count = 0
        consecutive_workdays = 0
        for j in range(num_days):
            if individuo[i][j] == 0:
                count += 1
                if consecutive_workdays > 5:
                    penalties["trabajos_consecutivos"] += 1
                consecutive_workdays = 0
            else:
                if count > 0:
                    breaks.append(count)
                count = 0
                consecutive_workdays += 1
        if count > 0:
            breaks.append(count)
        if breaks:
            longest_breaks.append(max(breaks))
            penalties["descansos_insuficientes"] += sum(2 for x in breaks if
x < 2)
        else:
            longest_breaks.append(0)

        if consecutive_workdays > 5:
            penalties["trabajos_consecutivos"] += 2

    if any(x>63 for x in dias_trabajados):
        penalties["limite_dias_trabajados"] +=1

    PDj_cycle = cycle(PDj)
    for i in range(num_days):
        demanda_diaria = next(PDj_cycle)
        if workforce_diario[i] != demanda_diaria:
            penalties["demanda_diaria"] += 1


    objective1 = sum (longest_breaks)
    objective2 = (max (longest_breaks) - min(longest_breaks))
    total_penalty = sum(penalties.values())
    fitness = int(objective1) - int(objective2) - int(total_penalty)
    return (fitness)


#imprime la matriz con los descansos marcados en verde (solo visualizacion)
def imprimir_matriz (matriz) :
    s = [[str(e) for e in row] for row in matriz]
    lens = [max(map(len, col)) for col in zip(*s)]
    formatted_rows = []
    for row in matriz:
        formatted_row = []
```

```
        for cell in row:
            # Si el valor de la celda es 0, cambia el color del texto a verde
            if cell == 0:
                formatted_cell = '\x1b[32m{}\x1b[0m'.format(cell)
            else:
                formatted_cell = str(cell)
            formatted_row.append(formatted_cell)
        formatted_rows.append(formatted_row)
    fmt = '\t'.join('{{:{}}}'.format(x) for x in lens)
    table = [fmt.format(*row) for row in formatted_rows]
    return table

#Generacion de una poblacion con un 65% de 1s
def generate_population(population_size, num_rows, num_cols, demand_vector):
    population = []

    for _ in range(population_size):
        individual = np.zeros((num_rows, num_cols), dtype=int)

        for col in range(num_cols):
            demand = demand_vector[col % len(demand_vector)]
            indices = random.sample(range(num_rows), demand)
            individual[indices, col] = 1

        population.append(individual)

    return population

def prob_mutacion(iteration, max_iterations):
    # Define la probabilidad de mutación inicial y final
    initial_mutation_prob = 0.9  # Probabilidad de mutación alta al principio
    final_mutation_prob = 0.2    # Probabilidad de mutación baja al final

    # Calcula la probabilidad de mutación actual en función de la iteración
actual
    mutation_prob = initial_mutation_prob - (initial_mutation_prob -
final_mutation_prob) * (iteration / max_iterations)
    return mutation_prob



def main():
    random.seed(64) # semilla del generador de números aleatorios
    start_time = time.time()
    p_cross = 0.9
    p_mutation = 0.5
    n_iteration = 500000
    size_population = 300
    PDj_cycle = [7, 9, 8, 9, 10, 7, 6]
    iteraciones = []  # Para almacenar el número de iteración
    valores_fo = []  # Para almacenar los valores de la función objetivo
    tiempos = []

    pop = generate_population(size_population, 14, 98, PDj_cycle)

    population_evaluated = list(zip(map(lambda gen: evalua(gen), pop), pop))


    best_solution_value, best_solution = max(population_evaluated, key=lambda
x: x[0])
    contador_mutacion = 0
    # Comenzamos la evolución:
```

```
    for i in range(n_iteration):
        # Selection - Tournament
        size_tournament = 3
        selec1 = np.random.permutation(size_population)[0:size_tournament]
        selec2 = np.random.permutation(size_population)[0:size_tournament]

        value1, ind1 = max((population_evaluated[selec1[i]][0], selec1[i])
for i in range(size_tournament))
        value2, ind2 = max((population_evaluated[selec2[i]][0], selec2[i])
for i in range(size_tournament))


        if np.random.rand() <= p_cross:
            # Generar un único punto de corte en la mitad de la matriz
            parent1 = pop[ind1]
            parent2 = pop[ind2]
            cross_point = len(parent1) // 2

            descendent1 = np.concatenate((parent1[:, :cross_point],
parent2[:, cross_point:]), axis=1)
            descendent2 = np.concatenate((parent2[:, :cross_point],
parent1[:, cross_point:]), axis=1)
        else:
            descendent1 = pop[ind1]
            descendent2 = pop[ind2]

        if np.random.rand() <= prob_mutacion(i,n_iteration):
            # Elegir un índice de fila y columna al azar para la mutación
            mutation_indices = np.random.randint(0, descendent1.size,
size=(3, 2))
            contador_mutacion +=1
            for row_idx, col_idx in mutation_indices:
                row = row_idx % descendent1.shape[0]
                col = col_idx % descendent1.shape[1]

                if descendent1[row, col] == 1:
                    # Cambiar de 1 a 0 en la posición seleccionada
                    descendent1[row, col] = 0

                    # Encontrar los índices de elementos 0 en la misma
columna
                    zero_indices = np.where(descendent1[:, col] == 0)[0]

                    # Elegir al azar un índice de elemento 0 para cambiar a 1
                    if zero_indices.size > 0:
                        idx_to_change = np.random.choice(zero_indices)
                        descendent1[idx_to_change, col] = 1

        if np.random.rand() <= prob_mutacion(i,n_iteration):
            # Elegir un índice de fila y columna al azar para la mutación
            mutation_indices = np.random.randint(0, descendent2.size,
size=(3, 2))
            contador_mutacion +=1
            for row_idx, col_idx in mutation_indices:
                row = row_idx % descendent2.shape[0]
                col = col_idx % descendent2.shape[1]

                if descendent2[row, col] == 1:
                    # Cambiar de 1 a 0 en la posición seleccionada
                    descendent2[row, col] = 0
```

```
                    # Encontrar los índices de elementos 0 en la misma
columna
                    zero_indices = np.where(descendent2[:, col] == 0)[0]

                    # Elegir al azar un índice de elemento 0 para cambiar a 1
                    if zero_indices.size > 0:
                        idx_to_change = np.random.choice(zero_indices)
                        descendent2[idx_to_change, col] = 1

        descendent1_value = evalua(descendent1)
        descendent2_value = evalua(descendent2)

        if descendent1_value > best_solution_value:
            best_solution_value = descendent1_value
            best_solution = descendent1

        if descendent2_value > best_solution_value:
            best_solution_value = descendent2_value
            best_solution = descendent2

            # Selection - Reject
        size_tournament = 3
        selec1 = np.random.permutation(size_population)[0:size_tournament]
        selec2 = np.random.permutation(size_population)[0:size_tournament]

        _, ind1 = min((population_evaluated[selec1[i]][0], selec1[i]) for i
in range(size_tournament))
        _, ind2 = min((population_evaluated[selec2[i]][0], selec2[i]) for i
in range(size_tournament))

        population_evaluated[ind1] = descendent1_value, descendent1
        population_evaluated[ind2] = descendent2_value, descendent2

        # Elitismo: reemplazar los peores individuos por los mejores de la
generación anterior
        population_evaluated.sort(key=lambda x: x[0], reverse=True)
        population_evaluated[-2:] = [(best_solution_value, best_solution)] *
2

        tiempo_iteracion = time.time()
        print("Iteración:", i)
        print("Mejor solución:", best_solution_value)
        print("Tiempo transcurrido:", tiempo_iteracion-start_time)
        print("Tamaño de la población:", len(population_evaluated))
        iteraciones.append(i)
        valores_fo.append(best_solution_value)
        tiempos.append(tiempo_iteracion - start_time)

    #Evaluacion de la solucion final:
    num_workers, num_days = np.shape(best_solution)
    PDj = [7, 9, 8, 9, 10, 7, 6]  # Demanda para cada día de la semana
    from itertools import cycle
    penalties = {
        "descansos_insuficientes": 0,
        "trabajos_consecutivos": 0,
        "limite_dias_trabajados": 0,
        "demanda_diaria": 0
    }
    longest_breaks = []
    dias_trabajados = 0
    workforce_diario = 0
```

```
    # Cálculo de dias totales trabajados
    dias_trabajados = np.sum(best_solution, axis=1)
    # Cálculo de los trabajadores diarios
    workforce_diario = np.sum(best_solution, axis=0)
    # Calculo de descansos
    for i in range(num_workers):
        breaks = []
        count = 0
        consecutive_workdays = 0
        for j in range(num_days):
            if best_solution[i][j] == 0:
                count += 1
                if consecutive_workdays > 5:
                    penalties["trabajos_consecutivos"] += 1
                consecutive_workdays = 0
            else:
                if count > 0:
                    breaks.append(count)
                count = 0
                consecutive_workdays += 1
        if count > 0:
            breaks.append(count)
        if breaks:
            longest_breaks.append(max(breaks))
            penalties["descansos_insuficientes"] += sum(1 for x in breaks if
x < 2)
        else:
            longest_breaks.append(0)
        if consecutive_workdays > 5:
            penalties["trabajos_consecutivos"] += 1
    if any(x > 63 for x in dias_trabajados):
        penalties["limite_dias_trabajados"] += 1
    PDj_cycle = cycle(PDj)
    for i in range(num_days):
        demanda_diaria = next(PDj_cycle)
        if workforce_diario[i] != demanda_diaria:
            penalties["demanda_diaria"] += 1


    print('El valor de la mejor solucion es %d' %best_solution_value)
    end_time = time.time()
    tiempo_transcurrido = end_time - start_time
    print ('El numero de mutaciones ha sido ', contador_mutacion)
    print('\nEl proceso tardó %d segundos' % (tiempo_transcurrido), 'y la
mejor solucion es : \n ')
    print('\n'.join(imprimir_matriz(best_solution)))
    print('\nEl vector de descansos de cada trabajador es ',  longest_breaks)
    print('La suma total de los descansos es de', sum(longest_breaks))
    print('La diferencia entre descansos maximos y minimos es de ',
max(longest_breaks) - min(longest_breaks))
    print('La penalizacion total es ', sum(penalties.values()))
    print('\nEl numero de penalizaciones incumplidas son:  ')
    for penalty_name, penalty_value in penalties.items():
        print(f"{penalty_name}: {penalty_value}")
    print('los trabajadores diarios son: ', workforce_diario)
    print('Los dias trabajados por cada empleado son:', dias_trabajados)

    # Representación gráfica de FO vs. Iteración
    plt.figure(1)
    plt.plot(iteraciones, valores_fo)
```

```
    plt.title("FO vs. Iteración")
    plt.xlabel("Iteración")
    plt.ylabel("Valor de la FO")

    # Representación gráfica de FO vs. Tiempo
    plt.figure(2)
    plt.plot(tiempos, valores_fo)
    plt.title("FO vs. Tiempo")
    plt.xlabel("Tiempo (segundos)")
    plt.ylabel("Valor de la FO")

    plt.show()  # Muestra las gráficas


if __name__ == "__main__":
    main()
```